Escuela Técnica Superior de Ingenieros – Universidad de Sevilla

Departamento de Teoría de la Señal y Comunicaciones

Algoritmos para el Filtrado Eficiente de Señales Reales basados en la DFT

Proyecto Fin de Carrera

Septiembre 2015, Sevilla

Autora: Lara Martín Lagares

Tutor: Francisco José Simois Tirado

Agradecimientos a mi familia…

"Por su apoyo incondicional y su paciencia durante todos estos años.

Por inculcarme valores como la constancia y el sacrificio.

Por valorar siempre mi trabajo y esfuerzo.

En definitiva, porque sin ellos hoy no estaría escribiendo estas líneas…."

Agradecimientos, también, a mis compañeros y profesores de carrera…

Índice de Contenidos

Capítulo 1

Introducción

1. Introducción

La transformada discreta de Fourier (DFT, 'Discrete Fourier Transform') juega un papel muy importante en el diseño, análisis y realización de sistemas y algoritmos de tratamiento de señales en tiempo discreto. Las propiedades básicas de la transformada de Fourier (DTFT, 'Discrete Time Fourier Transform') y de la DFT hacen que analizar y diseñar sistemas en el dominio transformado sea conveniente y práctico. Este hecho, junto con la existencia de algoritmos eficientes para el cálculo explícito de la DFT (en el capítulo 2 del presente documento, se describen brevemente algunos de estos algoritmos, conocidos como algoritmos de transformada rápida de Fourier (FFT, 'Fast Fourier Transform')), la convierten en un componente importante de muchas aplicaciones prácticas de los sistemas en tiempo discreto.

Una de estas aplicaciones es el filtrado lineal (convolución lineal), considerada una de las operaciones fundamentales en el procesamiento digital de señales. En el capítulo 3 del presente proyecto, se expondrán los métodos empleados tradicionalmente para ello y que son conocidos como algoritmos de solapamiento y almacenamiento (OLS, 'OverLap-and-Save') y solapamiento y suma (OLA, 'OverLap-and-Add'), aplicados principalmente en el caso de señales de entrada de larga duración.

En la práctica, las señales para las que la DFT es computada son a menudo reales y existen multitud de algoritmos específicamente diseñados para su cálculo. La forma más común de abordar el filtrado de una señal real es manejar la DFT de las señales de entrada y de salida separadamente, sin embargo, estos métodos no se centran en el proceso de filtrado en sí mismo.

En el capítulo 4.1, se presentará un método que realiza el proceso completo de filtrado a partir de versiones complejas de señales reales, logrando así una considerable reducción del tiempo de ejecución con respecto a los métodos tradicionales OLS y OLA. El sacrificio es un incremento en la latencia que se duplica. Mientras que la latencia no es un problema en muchas aplicaciones, sí puede llegar a ser un inconveniente para otras, particularmente en sistemas de tiempo real.

En el capítulo 4.2, se desarrolla una técnica que hace uso de DFT's complejas para el filtrado de señales reales. Este método presenta la misma latencia que OLS y OLA y se basa directamente en la DFT tradicional. Además, a diferencia de los algoritmos descritos en los capítulos 3 y 4.1, este método no está limitado al filtrado por bloques, pudiendo ser usado en cualquier tipo de proceso de convolución. Esta técnica reduce el número total de operaciones aritméticas, siendo el hecho más sobresaliente que el tiempo de ejecución es inferior al obtenido con los métodos comentados previamente. Por lo tanto, este

algoritmo puede ser utilizado en aplicaciones que usen señales reales y, particularmente, en sistemas de tiempo real donde la latencia deba mantenerse pequeña.

En los capítulos 5, 6 y Anexo se implementan en lenguaje C los algoritmos anteriormente comentados, se comparan y comentan los resultados obtenidos y se extraen conclusiones.

Capítulo 2

Algoritmos FFT

2. Algoritmos FFT

La DFT $X[k]$ de N puntos de la secuencia $x[n]$ son las muestras de la transformada de Fourier $X(e^{jw})$ en N frecuencias equiespaciadas $w_k = \frac{2\pi k}{N}$, es decir, en N puntos de la circunferencia unidad del plano complejo. En este capítulo se describen varios métodos muy eficientes que permiten calcular la DFT. Estos algoritmos se denominan colectivamente **transformada rápida de Fourier (FFT, '*Fast Fourier Transform*')**.

2.1 Breve reseña histórica del desarrollo de la FFT

La primera aparición de un método eficiente para el cálculo de la DFT, como muchos otros algoritmos, se debe al matemático alemán Karl Friedrich Gauss (1777-1855) [1]. En 1977 Herman H. Goldstine publicó un libro titulado *A History of Numerical Analysis from the 16th Through the 19th Century* [2], en el que describía la existencia de un artículo de Gauss relacionado con algoritmos de interpolación trigonométrica, extendido a funciones periódicas, para lo cual recurría a la expansión en series de Fourier en su forma trigonométrica, y de acuerdo con Goldstine, las simplificaciones matemáticas del trabajo contenían las ideas básicas del algoritmo de la FFT. El algoritmo de Gauss pasó desapercibido ya que fue publicado en latín después de su muerte, incluido en la recopilación de todas sus obras.

En abril de 1965, la revista *Mathematics of Computation* publicaba el ahora ya famoso artículo de 5 páginas "An algorithm for the machine calculation of complex Fourier series" [3], por James W. Cooley, investigador especialista en computación y análisis numérico, y John W. Tukey, profesor de estadística de la Universidad de Princeton. A los pocos días, la atención de la comunidad científica involucrada en el área de Procesamiento Digital de Señales se había volcado sobre el algoritmo descrito, que abría las puertas a un terreno lleno de posibilidades de aplicación. Este artículo, junto con el proselitismo de Richard L. Garwin, investigador del Centro de Investigación Watson de la IBM y compañero de J.W. Cooley, hicieron que el algoritmo se difundiera muy rápidamente. Garwin requería como parte de uno de sus proyectos un programa que realizara el cálculo de la Transformada Discreta de Fourier en tres dimensiones. Sabiendo de la capacidad de Cooley, decidió invitarlo a participar en el desarrollo, comentándole las recientes ideas presentadas por el profesor Tukey. Después de unos cuantos meses, quedó lista una primera versión del programa. Los resultados eras buenos, por lo que a sugerencia del grupo de computación de la IBM, el algoritmo fue puesto a disposición del dominio público. Sin la tenacidad de Garwin, es posible que la FFT siguiese siendo desconocida hoy en día.

Unos meses después de que Cooley y Tukey publicaran aquel artículo, comenzaron a surgir reportes de otros investigadores que habían utilizado técnicas similares. Philip Rudnick, investigador del Instituto Scripps de Oceanografía de la Universidad de California en San Diego, sometía un breve comunicado de dos páginas [4] para su publicación en la revista *Mathematics of Computation*. Rundnick reportó que había estado usando una técnica similar, adaptada de un trabajo publicado por Danielson y Lanczos en 1942. Efectivamente, el artículo de Danielson y Lanczos [5] describía en esencia el algoritmo de desdoblamiento básico de la FFT. Es interesante el hecho de que el algoritmo se presentaba en referencia al tratamiento de problemas en imágenes de rayos X, un área en la que, muchos años después de 1942, el cálculo de la Transformada de Fourier había sido un cuello de botella. Estos autores mostraban cómo reducir una transformada de $2N$ puntos en dos transformadas de N puntos, utilizando solamente N operaciones adicionales. Hoy en día el tamaño y el tiempo de cálculo de su problema causan pavor. A continuación, se exponen unas líneas extraídas de su artículo:

> *"Adopting these improvements the approximate times for Fourier analysis are 10 minutes for 8 coefficients, 25 minutes for 16 coefficients, 60 minutes for 32 coefficients, and 140 minutes for 64 coefficients".*

Revisando este breve resumen histórico, resulta llamativo que los algoritmos anteriores al de Cooley y Tukey hayan desaparecido o permaneciesen ocultos, mientras que éste tuvo enorme repercusión. Una explicación posible es que el interés en los aspectos teóricos del procesamiento digital de señales fue creciendo junto con las mejoras tecnológicas en la industria de los semiconductores, ya que la disponibilidad de mayor potencia de cómputo a costo razonable permite desarrollar muchas nuevas aplicaciones.

A su vez, el avance de la tecnología ha influido en el diseño del algoritmo. Mientras que en los años 60 y principios de los 70 el interés se centraba en disminuir el número de multiplicaciones, el progreso reveló que también son de importancia fundamental el número de sumas y de accesos a memoria (desde el punto de vista del software) y los costos de comunicación (desde el punto de vista del hardware).

2.2 Aplicación de la Simetría y la Periodicidad

El cálculo directo de la DFT es básicamente ineficiente debido, fundamentalmente, a que no explota las propiedades de simetría y periodicidad del factor de fase $W_N = e^{\frac{-j2\pi}{N}}$. En particular estas dos propiedades son:

Propiedad de Simetría: $W_N^{k+\frac{N}{2}} = -W_N^k$ **(2.2.1)**

Propiedad de Periodicidad: $W_N^{k+N} = W_N^k$ **(2.2.2)**

La simetría y periodicidad del factor de fase W_N queda de manifiesto en la siguiente figura. El ejemplo es para $N = 8$:

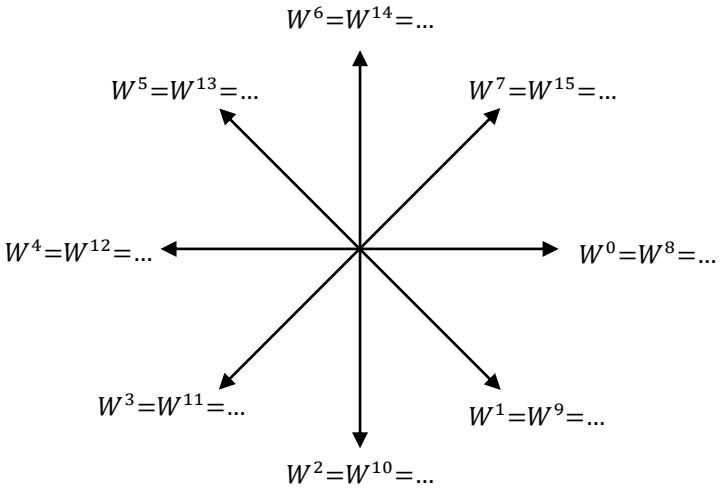


Figura 2.2.1: Periodicidad y simetría de los factores W

En cambio, los algoritmos FFT aprovechan las propiedades de periodicidad y simetría de la secuencia $W_N^{k \cdot n}$ para calcular la DFT de forma más eficiente.

2.3 Descripción de Algoritmos FFT

Los algoritmos de FFT se basan en el principio fundamental de descomponer el cálculo de una DFT de una secuencia de longitud N en DFTs sucesivamente más pequeñas que se combinan para formar la transformada de N puntos. Las transformadas de menor longitud se pueden evaluar por métodos directos o descomponerse a su vez en transformadas más pequeñas.

La forma en que se aplica este principio conduce a una variedad de algoritmos diferentes, todos ellos similares en cuanto a mejora de la velocidad de cómputo.

En este capítulo se presentan los algoritmos FFT más comunes y se realiza una comparativa entre ellos en cuanto a eficiencia ([6] y [7]). Se estudiará el caso en el que el número de puntos de la secuencia original (N) es potencia entera de 2, en cuyo caso son de aplicación dos tipos de algoritmos FFT. El primero de ellos, denominado de diezmado en tiempo (DET), se basa en descomponer la secuencia $x[n]$ en subsecuencias más pequeñas. En el segundo, la sucesión de los coeficientes $X[k]$ de la DFT se descompone en subsecuencias más cortas; de ahí su nombre de diezmado en frecuencia (DEF). Además de estos dos tipos de algoritmos, se estudiará un caso más general en el que N no es necesariamente potencia de 2, pero sí un número compuesto, es decir, producto de dos o más enteros.

2.3.1 Algoritmos FFT de diezmado en Tiempo

El cálculo de la DFT se puede hacer mucho más eficiente computando mayor cantidad de DFT de menor longitud (Duhamel y Vetterli, 1990). Este proceso explota tanto la simetría como la periodicidad de $e^{\frac{-j2\pi}{N}nk}$. Los algoritmos en los que la señal temporal $x[n]$ se descompone sucesivamente en secuencias más pequeñas se denominan algoritmos de diezmado en el tiempo (DET). Existe un caso especial en el que N es una potencia entera de 2, es decir, $N = 2^h$. Como N es un número entero par, se puede considerar el cálculo de $X[k]$ dividiendo $x[n]$ en dos secuencias de $N/2$ puntos correspondientes a las muestras[1] pares e impares de $x[n]$. Ambas secuencias se pueden expresar como sigue:

$$g_1[n] = x[2n] \qquad n = 0,1,\dots,\frac{N}{2} - 1. \qquad \textbf{(2.3.1.1)}$$

$$g_2[n] = x[2n+1] \qquad n = 0,1,\dots,\frac{N}{2} - 1. \qquad \textbf{(2.3.1.2)}$$

Por tanto, $g_1[n]$ y $g_2[n]$ se obtienen diezmando $x[n]$ por 2 y, en consecuencia, el algoritmo FFT resultante se denomina algoritmo de diezmado en el tiempo base 2.

A continuación, expresamos la DFT de N puntos de $x[n]$ en función de las DFTs de las secuencias diezmadas como sigue:

1 Las palabras *muestra* y *punto* se usan en este documento indistintamente para referirse a *"valor de una sucesión"*.

$$X[k] = \sum_{n=0}^{\frac{N}{2}-1} x[2n]W_N^{2nk} + \sum_{n=0}^{\frac{N}{2}-1} x[2n+1]W_N^{k(2n+1)}$$
$$= \sum_{n=0}^{\frac{N}{2}-1} g_1[n]W_{N/2}^{nk} + W_N^k \sum_{n=0}^{\frac{N}{2}-1} g_2[n]W_{N/2}^{kn} = G_1[k] + W_N^k G_2[k] \qquad (\mathbf{2.3.1.3})$$

Si a lo anterior añadimos el hecho de que $G_1[k]$ y $G_2[k]$ son periódicas de periodo $N/2$, por lo que $G_1[k+N/2] = G_1[k]$ y $G_2[k+N/2] = G_2[k]$, y además $W_N^{k+N/2} = -W_N^k$, la DFT de N puntos de la secuencia $x[n]$ se puede expresar como:

$$X[k] = F_1[k] + F_2[k] \qquad k = 0,1,\dots,\frac{N}{2}-1. \qquad \mathbf{(2.3.1.4)}$$

$$X[k+N/2] = F_1[k] - F_2[k] \qquad k = 0,1,\dots,\frac{N}{2}-1. \qquad \mathbf{(2.3.1.5)}$$

$donde\ \ F_1[k] = G_1[k]\ y\ F_2[k] = W_N^k G_2[k].$

A continuación, se representa el cálculo de una DFT de $N = 8$ puntos a partir de dos DFT de 4 puntos.

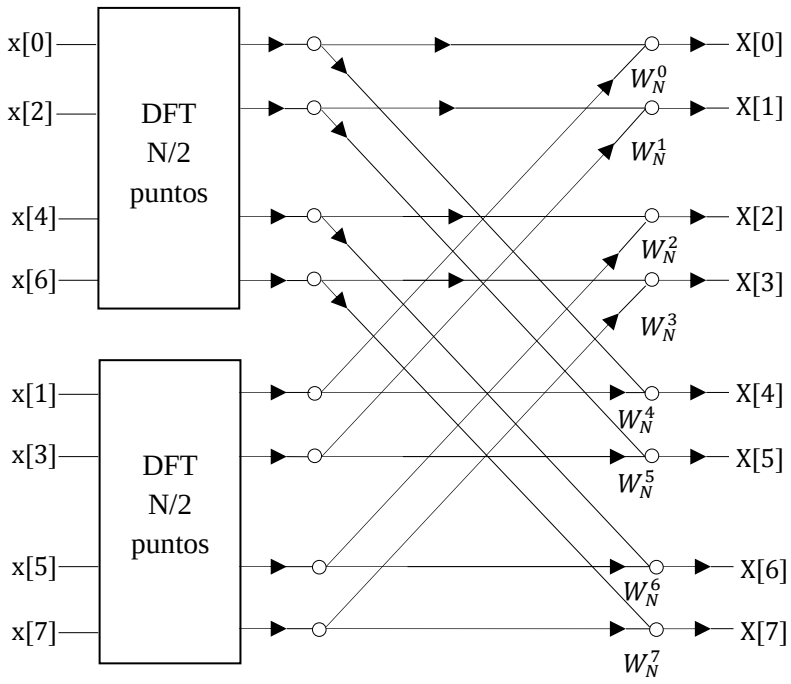


Figura 2.3.1.1: Descomposición de una DFT de N puntos en dos DFTs de N/2 puntos (N=8)

Resulta interesante comparar el número de multiplicaciones y sumas necesarias para calcular la DFT mediante este método con el que demanda el cálculo directo de la DFT. En este último caso, donde no se explota la simetría del factor $e^{\frac{j2\pi}{N}kn} = W_N^{kn}$, se necesitan aproximadamente N^2 multiplicaciones y sumas complejas. Si cada una de las transformadas de $N/2$ puntos, $G_1[k]$ y $G_2[k]$, se efectúa por cálculo directo, se deben realizar $2(N/2)^2$ multiplicaciones complejas y aproximadamente $2(N/2)^2$ sumas complejas. Además, para combinar las dos DFTs se requieren N productos complejos para multiplicar la segunda DFT de $N/2$ puntos, $G_2[k]$, por $e^{\frac{j2\pi}{N}kn} = W_N^{kn}$ y N sumas complejas para sumar este resultado con la primera DFT de $N/2$ puntos, $G_1[k]$. Por tanto, el cálculo de la DFT para todos los valores de k mediante este método requiere un máximo de $N/2 + 2(N/2)^2 = N/2 + N^2/2$ multiplicaciones y sumas complejas. Resulta sencillo comprobar que para $N > 2$, $N/2 + N^2/2$ será menor que N^2 (equivale, aproximadamente, a dividir por 2 el número de operaciones).

Si $N/2$ es par, lo que ocurre cuando N es potencia de 2, se puede considerar calcular cada una de las dos DFTs de $N/2$ puntos, $G_1[k]$ y $G_2[k]$, dividiéndolas a su vez en dos DFTs de $N/4$ puntos, que se combinarían para obtener las DFTs de $N/2$ puntos. Por tanto, $g_1[n]$ dará lugar a las dos subsecuencias de $N/4$ siguientes:

$$q_{11}[n] = g_1[2n] \qquad n = 0,1,\dots,\frac{N}{4} - 1. \qquad \textbf{(2.3.1.6)}$$

$$q_{12}[n] = g_1[2n+1] \qquad n = 0,1,\dots,\frac{N}{4} - 1. \qquad \textbf{(2.3.1.7)}$$

$$q_{21}[n] = g_2[2n] \qquad n = 0,1,\dots,\frac{N}{4} - 1. \qquad \textbf{(2.3.1.8)}$$

$$q_{22}[n] = g_2[2n+1] \qquad n = 0,1,\dots,\frac{N}{4} - 1. \qquad \textbf{(2.3.1.9)}$$

Calculando las DFTs de $N/4$ puntos obtendremos las DFTs de $N/2$ puntos, $G_1[k]$ y $G_2[k]$, a partir de las siguientes relaciones:

$$G_1[k] = Q_{11}[k] + W_{N/2}^k Q_{12}[k] \qquad k = 0,1,\dots,\frac{N}{4} - 1. \qquad \textbf{(2.3.1.10)}$$

$$G_1[k + N/4] = Q_{11}[k] - W_{N/2}^k Q_{12}[k] \qquad k = 0,1,\dots,\frac{N}{4} - 1. \qquad \textbf{(2.3.1.11)}$$

$$G_2[k] = Q_{21}[k] + W_{N/2}^{k} Q_{22}[k] \qquad k = 0,1,\ldots,\frac{N}{4} - 1. \qquad \textbf{(2.3.1.12)}$$

$$G_2[k + N/4] = Q_{21}[k] - W_{N/2}^{k} Q_{22}[k] \qquad k = 0,1,\ldots,\frac{N}{4} - 1. \qquad \textbf{(2.3.1.13)}$$

En la siguiente figura se representa la descomposición de una DFT de $N/2$ puntos en dos DFTs de $N/4$ puntos para el caso de $N = 8$:

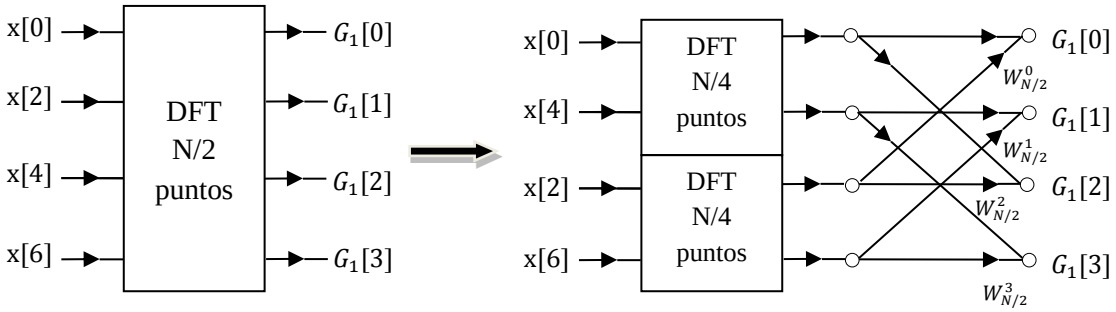


Figura 2.3.1.2: Descomposición de una DFT de N/2 puntos en dos DFTs de N/4 puntos (N = 8).

Si sustituimos esta estructura en la figura 2.3.1.1, obtenemos como resultado el siguiente diagrama:

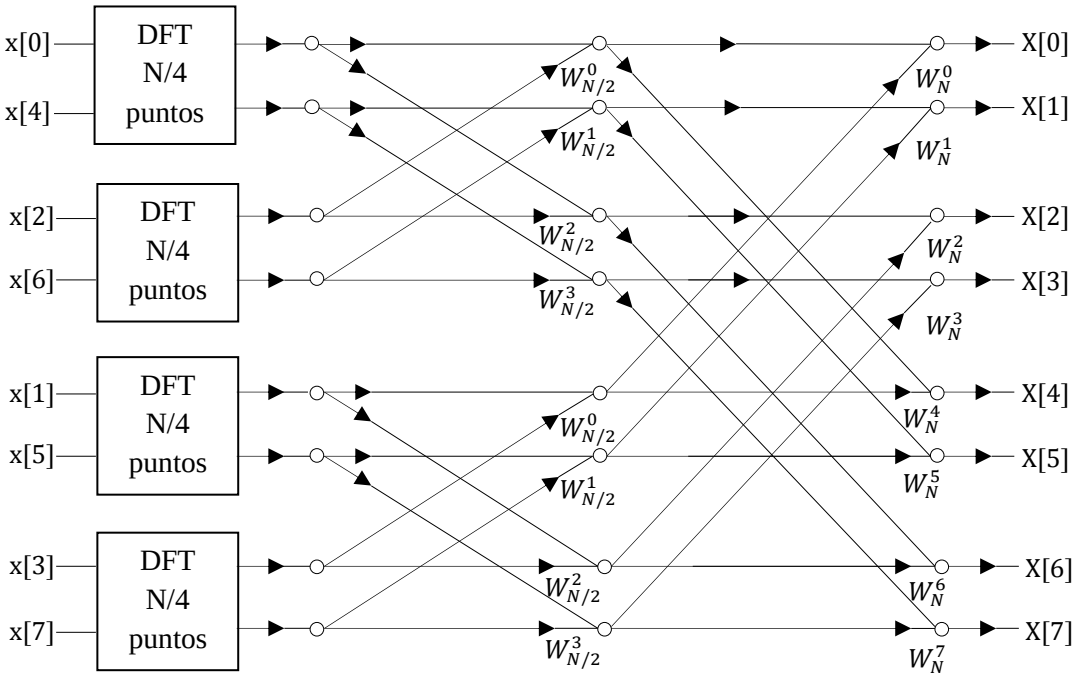


Figura 2.3.1.3: Descomposición de una DFT de N/2 puntos en dos DFTs de N/4 puntos (N = 8).

Las transformadas de $N/4$ puntos se pueden seguir descomponiendo en dos transformadas de $N/8$ puntos cada una. Esta descomposición puede continuar hasta llegar a transformadas de 2 puntos cuyo cálculo resulta muy sencillo: si $y[n]$ es una secuencia de dos puntos, la DFT $Y[k]$ viene dada por:

$$Y[k] = \sum_{n=0}^{N-1} y[n] e^{-j\frac{2\pi}{N}kn} = \sum_{n=0}^{1} y[n]\, e^{-j\frac{2\pi}{2}kn} = y[0] + (-1)^k y[1] \qquad (\mathbf{2.3.1.14})$$

De modo que:

$$Y[0] = y[0] + y[1]$$

$$Y[1] = y[0] - y[1]$$

Por tanto, la DFT de 2 puntos sólo necesita una suma y una resta.

A continuación se muestra el diagrama de cálculo de la DFT de 2 puntos de la subsecuencia formada por $x[0]$ y $x[4]$ para el caso de una secuencia de $N = 8$ puntos:

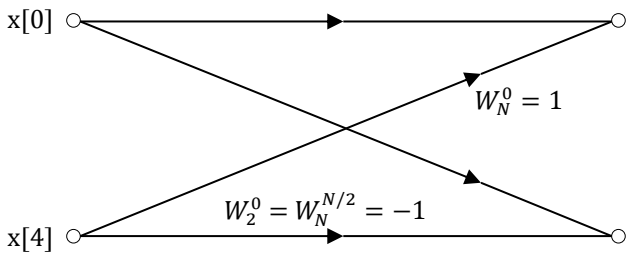


Figura 2.3.1.4: Diagrama de cálculo de una DFT de 2 puntos.

Para $N = 2^h$, el diezmado puede realizarse $h = \log_2 N$ veces. Como en la descomposición original de una transformada de N puntos en dos transformadas de $N/2$ puntos se requerían $N + 2(N/2)^2$ multiplicaciones y sumas, cuando las transformadas de $N/2$ puntos se descomponen en transformadas de $N/4$ puntos, el factor $(N/2)^2$ se debe sustituir por $N/2 + 2(N/4)^2$, por lo que el número total de operaciones es $N + N + 4(N/4)^2$ multiplicaciones y sumas complejas. Por tanto, tras realizar la descomposición tantas veces como es posible, es decir, $h = \log_2 N$ etapas de cómputo, el número de multiplicaciones y sumas es igual a $Nh = N\log_2 N$. A continuación, se muestra a modo ilustrativo las $h = 3$ etapas en las que se divide el cálculo de una DFT de $N = 8$ puntos, comenzando con el cálculo de cuatro DFTs de dos puntos, después dos de cuatro puntos, y finalmente, una de ocho puntos:

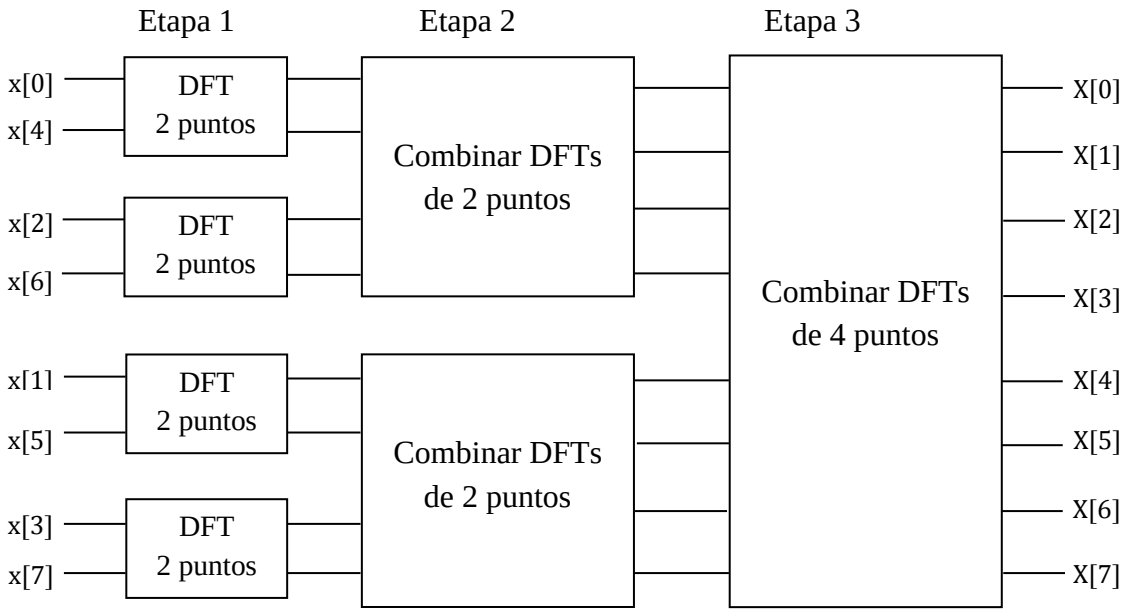


Figura 2.3.1.5: Etapas para el cálculo de una DFT de 8 puntos.

Sustituyendo en cada una de las etapas el diagrama de cálculo correspondiente, obtenemos el siguiente esquema:

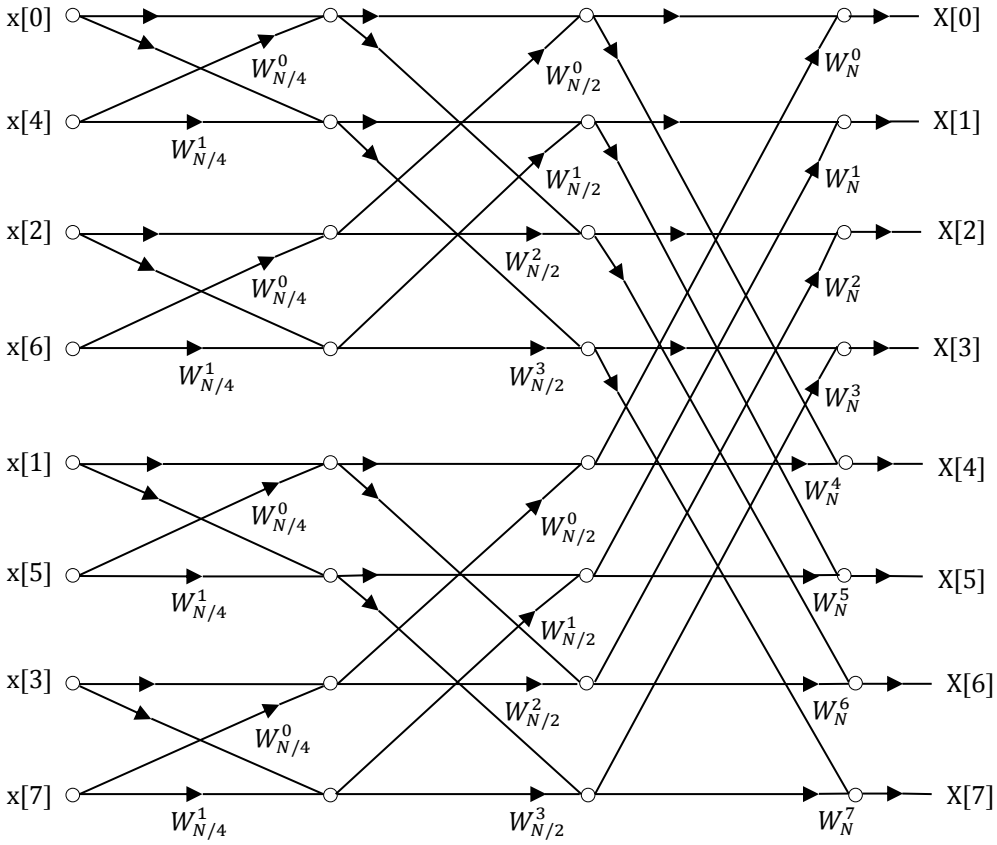


Figura 2.3.1.6: Diagrama completo de una DFT de 8 puntos.

El número de cálculos necesarios para computar la DFT de la figura 2.3.1.6 se puede reducir aún más explotando la simetría y la periodicidad de los coeficientes $W_N^p = e^{\frac{-j2\pi}{N}p}$. En el diagrama de la figura 2.3.1.6 para pasar de una etapa a la siguiente se debe efectuar el cálculo básico que se muestra en la figura 2.3.1.7, que por la forma de representación se denomina *mariposa*, donde los coeficientes son siempre potencias de $W_N = e^{\frac{j2\pi}{N}}$ y los exponentes difieren en $N/2$.

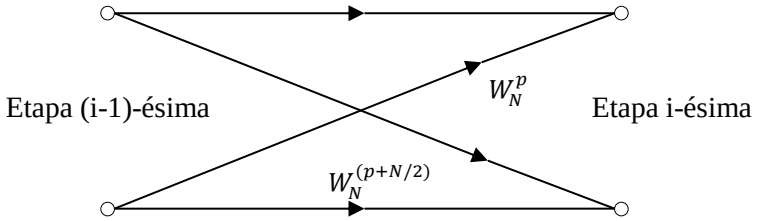


Figura 2.3.1.7: Diagrama básico mariposa.

Además, teniendo en cuenta que $W_N^{N/2} = e^{\frac{-j2\pi}{N}\cdot\frac{N}{2}} = e^{-j\pi} = -1$, el factor $W_N^{(p+N/2)} = e^{\frac{-j2\pi}{N}(p+N/2)}$ se puede escribir como:

$$W_N^{(p+N/2)} = e^{\frac{-j2\pi}{N}(p+N/2)} = e^{\frac{-j2\pi}{N}\cdot\frac{N}{2}} \cdot e^{\frac{j2\pi}{N}p} = -e^{\frac{j2\pi}{N}p} = -W_N^p \qquad \textbf{(2.3.1.15)}$$

Teniendo en cuenta el resultado anterior, el diagrama mariposa representado en la figura 2.3.1.7 se puede simplificar como se muestra en la figura 2.3.1.8, en la que sólo se requiere una multiplicación compleja en vez de dos.

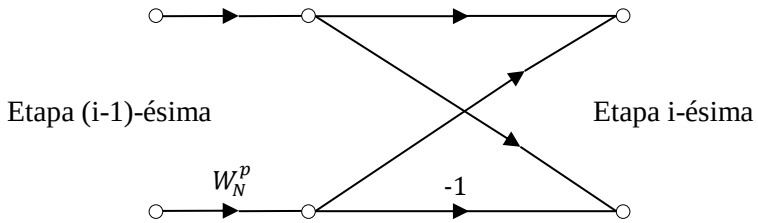


Figura 2.3.1.8: Diagrama básico mariposa con un solo producto complejo.

Introduciendo esta simplificación en la figura 2.3.1.6 se obtiene el diagrama de la siguiente figura 2.3.1.9, en el que el número de multiplicaciones complejas se reduce a $\frac{N}{2}\log_2 N$ (reducción en un factor de 2).

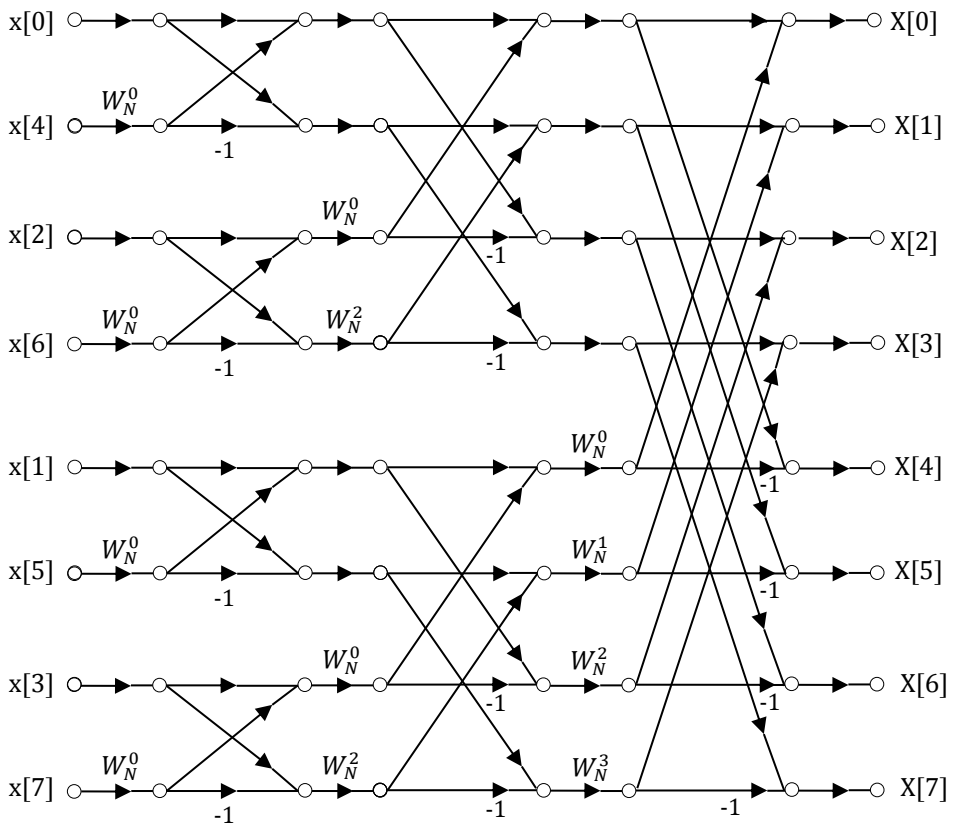


Figura 2.3.1.9: Diagrama completo de una DTF de 8 puntos usando mariposa de un solo producto.

2.3.2 Algoritmos FFT de diezmado en Frecuencia

Los algoritmos de FFT por DET se basan en calcular la DFT dividiendo la sucesión temporal $x[n]$ en sucesiones de menor longitud. Una forma equivalente se obtiene si se divide la sucesión frecuencial $X[k]$ en sucesiones más pequeñas. Estos algoritmos se denominan de decimación en frecuencia (DEF).

Al igual que hicimos para el caso DET, estudiaremos la situación especial en la que N es una potencia entera de 2, es decir, $N = 2^h$. Como N es un número entero par, distinguiremos entre las muestras pares e impares de $X[k]$:

$$V_1[k] = X[2k] \qquad k = 0,1,\dots,\frac{N}{2} - 1. \qquad \textbf{(2.3.2.1)}$$

$$V_2[k] = X[2k+1] \qquad k = 0,1,\dots,\frac{N}{2} - 1. \qquad \textbf{(2.3.2.2)}$$

Por tanto, $V_1[k]$ y $V_2[k]$ se obtienen diezmando $X[k]$ por 2 y, en consecuencia, el algoritmo FFT resultante se denomina algoritmo de diezmado en la frecuencia base 2.

En primer lugar, calcularemos una expresión para las muestras pares de $X[k]$:

$$V_1[k] = \sum_{n=0}^{\frac{N}{2}-1} x[n]W_N^{2nk} + \sum_{n=\frac{N}{2}}^{N-1} x[n]W_N^{2nk} \quad =$$

$$= \sum_{n=0}^{\frac{N}{2}-1} x[n]W_N^{2nk} + \sum_{n=0}^{\frac{N}{2}-1} x\left[n+\frac{N}{2}\right] W_N^{2\left(n+\frac{N}{2}\right)k} \qquad k = 0,1,\dots,\frac{N}{2}-1. \qquad (\mathbf{2.3.2.1})$$

Teniendo en cuenta las siguientes equivalencias:

$$e^{-j\frac{2\pi}{N}2(n+\frac{N}{2})k} = e^{-j\frac{2\pi}{N}2nk} \cdot e^{-j2\pi k} = e^{-j\frac{2\pi}{N}2nk} = e^{-j\frac{2\pi}{N/2}nk} = W_{N/2}^{nk} \qquad (\mathbf{2.3.2.2})$$

La ecuación 2.3.2.1 se puede expresar como sigue:

$$V_1[k] = \sum_{n=0}^{\frac{N}{2}-1} \{x[n] + x\left[n+\frac{N}{2}\right]\} W_{\frac{N}{2}}^{nk} \qquad k = 0,1,\dots,\frac{N}{2}-1. \qquad (\mathbf{2.3.2.3})$$

A continuación, hallaremos una expresión para las muestras impares de $X[k]$:

$$V_2[k] = \sum_{n=0}^{\frac{N}{2}-1} x[n]W_N^{(2k+1)n} + \sum_{n=\frac{N}{2}}^{N-1} x[n]W_N^{(2k+1)n} \qquad k = 0,1,\dots,\frac{N}{2}-1. \qquad (\mathbf{2.3.2.4})$$

Teniendo en cuenta las siguientes equivalencias:

$$\sum_{n=\frac{N}{2}}^{N-1} x[n]W_N^{(2k+1)n} = \sum_{n=0}^{\frac{N}{2}-1} x\left[n+\frac{N}{2}\right] W_N^{(2k+1)\left(n+\frac{N}{2}\right)} = e^{-j\frac{2\pi}{N}\left(\frac{N}{2}\right)(2k+1)} \sum_{n=0}^{\frac{N}{2}-1} x\left[n+\frac{N}{2}\right] W_N^{(2k+1)n}$$

$$= e^{-j2\pi k} e^{-j\pi} \sum_{n=0}^{\frac{N}{2}-1} x\left[n+\frac{N}{2}\right] W_N^{(2k+1)n}$$

$$= -\sum_{n=0}^{\frac{N}{2}-1} x\left[n+\frac{N}{2}\right] W_N^{(2k+1)n}$$

$$k = 0,1,\dots,\frac{N}{2}-1. \qquad (\mathbf{2.3.2.5})$$

La ecuación 2.3.2.4 se puede expresar como sigue:

$$V_2[k] = \sum_{n=0}^{\frac{N}{2}-1} x[n]W_N^{(2k+1)n} - \sum_{n=0}^{\frac{N}{2}-1} x\left[n+\frac{N}{2}\right]W_N^{(2k+1)n} = \sum_{n=0}^{\frac{N}{2}-1} \{x[n] - x\left[n+\frac{N}{2}\right]\}W_N^{(2k+1)n}$$

$$= \sum_{n=0}^{\frac{N}{2}-1} \{(x[n] - x\left[n+\frac{N}{2}\right])e^{-j\frac{2\pi}{N}n}\}W_{\frac{N}{2}}^{kn} \qquad k = 0,1,\dots,\frac{N}{2}-1. \qquad (\mathbf{2.3.2.6})$$

Definiendo las secuencias de $N/2$ puntos $s_1(n)$ y $s_2(n)$ como:

$$s_1(n) = x[n] + x[n+N/2] \qquad n = 0,1,2,\dots,\frac{N}{2}-1 \qquad \mathbf{(2.3.2.7)}$$

$$s_2(n) = (x[n] - x[n+N/2])\,W_N^n \qquad n = 0,1,2,\dots,\frac{N}{2}-1 \qquad \mathbf{(2.3.2.8)}$$

Entonces:

$$V_1[k] = \sum_{n=0}^{\frac{N}{2}-1} s_1(n)\,W_{\frac{N}{2}}^{kn} \qquad k = 0,1,\dots,\frac{N}{2}-1. \qquad (\mathbf{2.3.2.9})$$

$$V_2[k] = \sum_{n=0}^{\frac{N}{2}-1} s_2(n)\,W_{\frac{N}{2}}^{kn} \qquad k = 0,1,\dots,\frac{N}{2}-1. \qquad (\mathbf{2.3.2.10})$$

Por tanto, $V_1[k]$ y $V_2[k]$ se corresponden con las DFT de $N/2$ puntos de las secuencias $s_1(n)$ y $s_2(n)$, respectivamente.

La figura 2.3.2.1 muestra el procedimiento sugerido por las ecuaciones 2.3.2.3 y 2.3.2.6 para el cálculo de una DFT de $N = 8$ puntos:

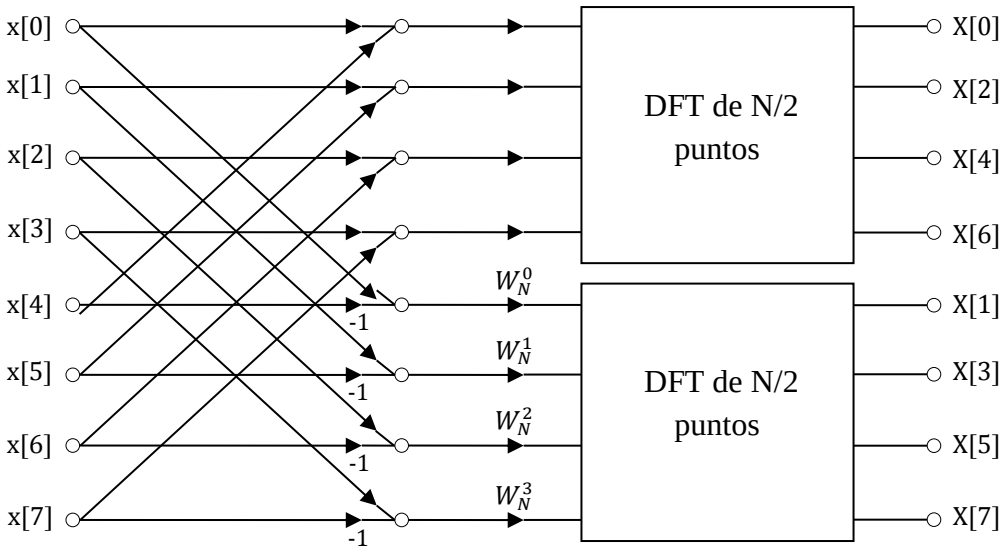


Figura 2.3.2.1: Descomposición de una DFT de N puntos en DFTs de N/2 puntos (N=8).

En este caso podemos proceder de manera similar al desarrollo realizado para obtener el algoritmo mediante diezmado en el tiempo. Como N es potencia de 2, $N/2$ es divisible por 2, por tanto, en las DFTs de $N/2$ puntos podemos calcular separadamente las muestras de índice par e impar. Esto se realiza combinando la primera y la segunda mitad de los puntos de entrada para cada una de las DFT de $N/2$ puntos, y calculando posteriormente las DFTs de $N/4$ puntos. La figura 2.3.2.2 muestra el diagrama resultante de aplicar este procedimiento:

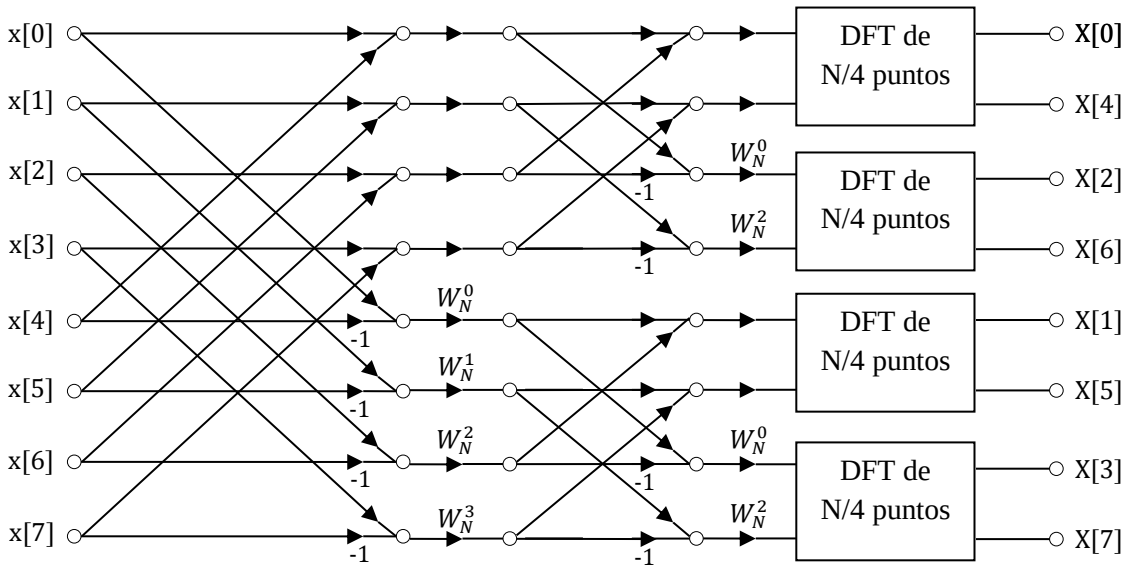


Figura 2.3.2.2: Descomposición de una DFT de N puntos en DFTs de N/4 puntos (N=8).

En el diagrama de la figura 2.3.2.2 para pasar de una etapa a la siguiente se emplea el cálculo básico que se muestra en la siguiente figura, y que por su forma de representación se denomina *mariposa*:

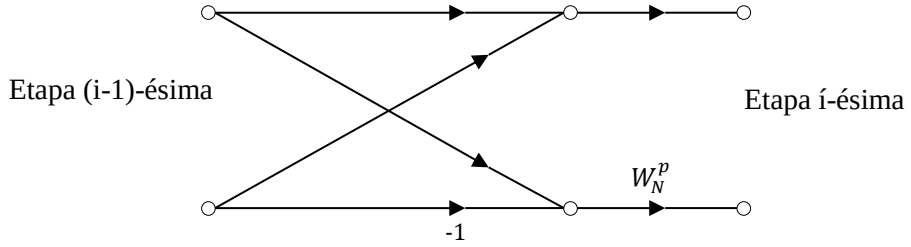


Figura 2.3.2.3: Diagrama básico mariposa.

Sustituyendo la estructura de la figura 2.3.2.3 en el diagrama de la figura 2.3.2.2, obtenemos el siguiente resultado:

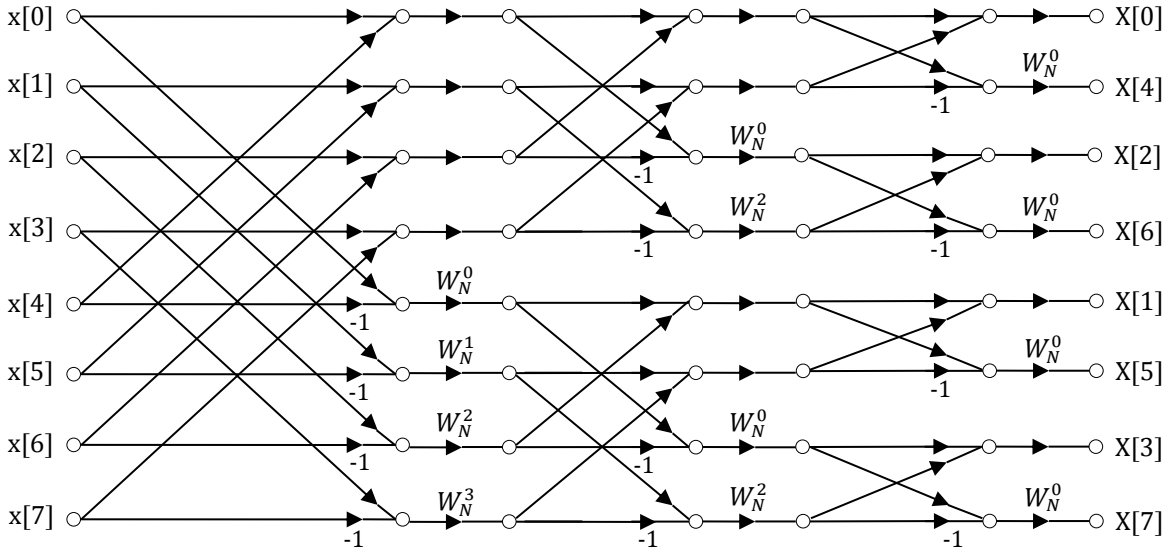


Figura 2.3.2.4: Diagrama completo de una DFT de 8 puntos.

Contando las operaciones aritméticas que se emplean en la figura 2.3.2.4 y generalizando para $N = 2^h$, podemos comprobar que se necesitan $(N/2)log_2 N$ multiplicaciones complejas y $Nlog_2 N$ sumas complejas para el cálculo de los coeficientes de una DFT de N puntos. Por tanto, en los algoritmos de diezmado en el tiempo y diezmado en la frecuencia, el número de operaciones necesarias es el mismo.

2.3.3 Algoritmos FFT para N Factorizable (Cooley – Tukey)

Los mismos principios aplicados cuando N es una potencia de 2 para obtener los algoritmos de diezmado en el tiempo y en la frecuencia se pueden aplicar al caso en que N sea un entero compuesto, es decir, el producto de dos o más factores enteros y, por tanto, un número no primo. Por ejemplo, si $N = N_1 \cdot N_2$, es posible expresar una DFT de N puntos como una combinación de N_1 transformadas DFT de N_2 puntos o como una combinación de N_2 transformadas de N_1 puntos, obteniendo así una considerable reducción del número de operaciones.

A continuación detallaremos los pasos a seguir en el desarrollo de este algoritmo:

Se toma la secuencia original $x[n]$ de N puntos ($0 \leq n \leq N$-1) y se almacena en una matriz bidimensional indexada por n_1 y n_2, donde $0 \leq n_1 \leq N_1$-1 y $0 \leq n_2 \leq N_2$-1, obteniéndose así la matriz $x[n_1, n_2]$. Este almacenamiento se puede realizar de diferentes maneras, en función de la correspondencia entre el índice n y los índices n_1 y n_2. A continuación se muestran las distintas opciones:

- **Almacenamiento de $x[n]$ por filas ($n = N_2 \cdot n_1 + n_2$)**

	0	1	2	.. ..	N_2-1
0	$x[0]$	$x[1]$	$x[2]$	.. ..	$x[N_2\text{-}1]$
1	$x[N_2]$	$x[N_2+1]$	$x[N_2+2]$	.. ..	$x[2N_2\text{-}1]$
2	$x[2N_2]$	$x[2N_2+1]$	$x[2N_2+2]$	.. ..	$x[3N_2\text{-}1]$
:	:	:	:	.. ..	:
:	:	:	:		:
N_1-1	$x[(N_1\text{-}1)N_2]$	$x[(N_1\text{-}1)N_2+1]$	$x[(N_1\text{-}1)N_2+2]$	.. ..	$x[N_1N_2\text{-}1]$

Esto conlleva a una disposición en la que la primera fila contiene los N_2 primeros elementos de $x[n]$, la segunda fila contiene los siguientes N_2 elementos, y así sucesivamente.

- **Almacenamiento de $\mathbf{x[n]}$ por columnas ($\mathbf{n = n_1 + n_2 \cdot N_1}$):**

	0	1	2	.. ..	N_2-1
0	$x[0]$	$x[N_1]$	$x[2N_1]$	.. ..	$x[(N_2\text{-}1)N_1]$
1	$x[1]$	$x[N_1\text{+}1]$	$x[2N_1\text{+}1]$	.. ..	$x[(N_2\text{-}1)N_1\text{+}1]$
2	$x[2]$	$x[N_1\text{+}2]$	$x[2N_2\text{+}2]$	.. ..	$x[(N_2\text{-}1)N_1\text{+}2]$
:	:	:	:	.. ..	:
:	:	:	:		:
N_1-1	$x[N_1\text{-}1]$	$x[2N_1\text{-}1]$	$x[3N_1\text{-}1]$	.. ..	$x[N_1N_2\text{-}1]$

En este caso, obtenemos una disposición en la que la primera columna contiene los N_1 primeros elementos de $x[n]$, la segunda columna contiene los siguientes N_1 elementos, y así sucesivamente.

El resultado obtenido en el cálculo de la DFT de la secuencia $x[n]$ también será almacenado en una matriz rectangular $X[k_1, k_2]$, donde $0 \leq k_1 \leq N_1\text{-}1$ y $0 \leq k_2 \leq N_2\text{-}1$. Al igual que para el caso de $x[n]$, tenemos dos posibles disposiciones para esta matriz; almacenamiento por filas ($k = N_2 \cdot k_1 + k_2$) o almacenamiento por columnas ($k = k_2 \cdot N_1 + k_1$).

Como se ha comentado anteriormente, $x[n]$ se dispone en la matriz rectangular $x[n_1, n_2]$ y $X[k]$ en la matriz $X[k_1, k_2]$. De este modo, la matriz $X[k_1, k_2]$ se puede expresar como el sumatorio doble de los elementos de la matriz rectangular $x[n_1, n_2]$ multiplicados por los factores de fase correspondientes.

Existen dos posibilidades para el cálculo de los elementos de la matriz $X[k_1, k_2]$, en función de la correspondencia establecida entre el índice n y los índices n_1 y n_2, y entre el índice k y los índices k_1 y k_2. A continuación, se representan estas dos opciones:

- **Almacenamiento $\mathbf{x[n]}$ por filas y $\mathbf{X[k]}$ por columnas → $\mathbf{n = N_2 \cdot n_1 + n_2 / k = k_2 \cdot N_1 + k_1}$**

$$X[k_1, k_2] = \sum_{n_2=0}^{N_2-1} \sum_{n_1=0}^{N_1-1} x[n_1, n_2] \cdot W_N^{(N_2 \cdot n_1 + n_2) \cdot (k_2 \cdot N_1 + k_1)} \quad \mathbf{(2.3.3.1)}$$

Teniendo en cuenta que: $W_N^{(N_2\cdot n_1+n_2)\cdot(k_2\cdot N_1+k_1)} = W_N^{N_2\cdot N_1\cdot n_1\cdot k_2}\cdot W_N^{N_2\cdot n_1\cdot k_1}\cdot W_N^{N_1\cdot n_2\cdot k_2}\cdot W_N^{n_2\cdot k_1}$ y que: $W_N^{N_2\cdot N_1\cdot n_1\cdot k_2} = W_N^{N\cdot n_1\cdot k_2} = e^{-j2\pi\cdot n_1\cdot k_2} = 1$; $W_N^{N_2\cdot n_1\cdot k_1} = e^{\frac{-j2\pi\cdot N_2\cdot n_1\cdot k_1}{N}} = e^{\frac{-j2\pi\cdot n_1\cdot k_1}{N_1}} = W_{N_1}^{n_1\cdot k_1}$; $W_N^{N_1\cdot n_2\cdot k_2} = e^{\frac{-j2\pi\cdot N_1\cdot n_2\cdot k_2}{N}} = W_{N_2}^{n_2\cdot k_2}$.

Obtenemos el siguiente resultado:

$$X[k_1,k_2] = \sum_{n_2=0}^{N_2-1}\sum_{n_1=0}^{N_1-1} x[n_1,n_2]\cdot W_{N_1}^{n_1\cdot k_1}\cdot W_{N_2}^{n_2\cdot k_2}\cdot W_N^{n_2\cdot k_1}$$

$$= \sum_{n_2=0}^{N_2-1}\left(W_N^{n_2\cdot k_1}\cdot\sum_{n_1=0}^{N_1-1} x[n_1,n_2]\cdot W_{N_1}^{n_1\cdot k_1}\right)\cdot W_{N_2}^{n_2\cdot k_2} \quad \mathbf{(2.3.3.2)}$$

El algoritmo al que da lugar la expresión anterior implica los siguientes cálculos:

- DFT de N_1 puntos de cada columna de la matriz $x[n_1,n_2]$
- Multiplicación de la matriz resultante por los factores $W_N^{n_2\cdot k_1}$.
- DFT de N_2 puntos de cada fila de la matriz obtenida en el paso anterior.

De este modo, la secuencia $X[k]$ se obtiene leyendo fila por fila la matriz $X[k_1,k_2]$.

- **Almacenamiento $x(n)$ por columnas y $X(k)$ por filas** → $n = n_1 + n_2\cdot N_1 /\ k = N_2\cdot k_1 + k_2$

$$X(p,q) = \sum_{n_2=0}^{N_2-1}\sum_{n_1=0}^{N_1-1} x(n_1,n_2)\cdot W_N^{(n_1+n_2\cdot N_1)\cdot(N_2\cdot k_1+k_2)} \quad \mathbf{(2.3.3.3)}$$

Teniendo en cuenta que: $W_N^{(n_1+n_2\cdot N_1)\cdot(N_2\cdot k_1+k_2)} = W_N^{n_1\cdot N_2\cdot k_1}\cdot W_N^{n_1\cdot k_2}\cdot W_N^{n_2\cdot N_1\cdot N_2\cdot k_1}\cdot W_N^{n_2\cdot N_1\cdot k_2}$ y que: $W_N^{n_1\cdot N_2\cdot k_1} = e^{\frac{-j2\pi\cdot N_2\cdot n_1\cdot k_1}{N}} = e^{\frac{-j2\pi\cdot n_1\cdot k_1}{N_1}} = W_{N_1}^{n_1\cdot k_1}$; $W_N^{n_2\cdot N_1\cdot N_2\cdot k_1} = e^{-j2\pi\cdot n_2\cdot k_1} = 1$; $W_N^{n_2\cdot N_1\cdot k_2} = e^{\frac{-j2\pi\cdot N_1\cdot n_2\cdot k_2}{N}} = W_{N_2}^{n_2\cdot k_2}$ .Obtenemos el siguiente resultado:

$$X(p,q) = \sum_{n_2=0}^{N_2-1} \sum_{n_1=0}^{N_1-1} x(n_1,n_2) \cdot W_{N_2}^{n_2 \cdot k_2} \cdot W_{N_1}^{n_1 \cdot k_1} \cdot W_N^{n_1 \cdot k_2}$$

$$= \sum_{n_1=0}^{N_1-1} [W_N^{n_1 \cdot k_2} \cdot \sum_{n_2=0}^{N_2-1} x(n_1,n_2) \cdot W_{N_2}^{n_2 \cdot k_2}] \cdot W_{N_1}^{n_1 \cdot k_1} \quad (\mathbf{2.3.3.4})$$

El algoritmo que da lugar a la expresión anterior implica los siguientes cálculos:

- DFT de N_2 puntos de cada fila de la matriz $x[n_1, n_2]$.
- Multiplicación de la matriz resultante por los factores $W_N^{n_1 \cdot k_2}$.
- DFT de N_1 puntos de cada columna de la matriz resultado del paso anterior.

A simple vista pudiera dar la impresión de que el cálculo computacional de los dos algoritmos descritos anteriormente es mayor que el cálculo directo de la DFT. Sin embargo, analizando el número de operaciones requeridas en ambos procesos confirmamos que no es así:

- **Almacenamiento $x[n]$ por filas y $X[k]$ por columnas → $n = N_2 \cdot n_1 + n_2 / k = k_2 \cdot N_1 + k_1$**
  - Cálculo de N_2 DFTs, cada una de N_1 puntos → $N_2 N_1^2$ multiplicaciones complejas y $N_2 N_1 (N_1 - 1)$ sumas complejas.
  - Multiplicación por los factores $W_N^{n_2 k_1}$ → $N_1 N_2$ multiplicaciones complejas.
  - Cálculo de N_1 DFTs, cada una de N_2 puntos → $N_1 N_2^2$ multiplicaciones complejas y $N_1 N_2 (N_2 - 1)$ sumas complejas.

- **Almacenamiento $x(n)$ por columnas y $X(k)$ por filas → $n = n_1 + n_2 \cdot N_1 / \ k = N_2 \cdot k_1 + k_2$**
  - Cálculo de N_1 DFTs, cada una de N_2 puntos → $N_1 N_2^2$ multiplicaciones complejas y $N_1 N_2 (N_2 - 1)$ sumas complejas.
  - Multiplicación por los factores $W_N^{n_1 k_2}$ → $N_1 N_2$ multiplicaciones complejas.
  - Cálculo de N_2 DFTs, cada una de N_1 puntos → $N_2 N_1^2$ multiplicaciones complejas y $N_2 N_1 (N_1 - 1)$ sumas complejas.

Ambos algoritmos tienen la misma complejidad, distinguiéndose en el orden de los cálculos. En resumen, el número de operaciones necesarias en ambos casos sería:

Multiplicaciones complejas: $\boldsymbol{N\,(N_1 + N_2 + 1)}$

Sumas complejas: $\boldsymbol{N\,(N_2 + N_1 - 2)}$

2.4 Consideraciones Adicionales sobre la FFT

Vistos en los apartados anteriores dos algoritmos para el cálculo de la FFT; diezmado en tiempo y diezmado en frecuencia, en este apartado detallaremos una serie de consideraciones prácticas a tener en cuenta a la hora de implementarlos:

1. Cantidad de memoria utilizada en la implementación del algoritmo: se minimiza usando *algoritmos de cómputo en el mismo lugar*.
2. Ordenación de los datos de entrada y salida al algoritmo.
3. Formas alternativas de realizar el algoritmo FFT.
4. Minimización de los cálculos de las exponenciales involucradas en el proceso utilizando distintas estrategias.

2.4.1 Algoritmos de Cómputo en el Mismo Lugar

Los grafos de flujo obtenidos como resultado de aplicar los algoritmos de diezmado en tiempo y diezmado en frecuencia presentan algunas similitudes y también algunas diferencias. Por supuesto, en ambos casos, estos diagramas se corresponden al cálculo de la DFT, independientemente de cómo se dibujen. Las características esenciales de estos grafos son los arcos que conectan los nodos, así como las ganancias de dichos arcos. La disposición de los nodos es indistinta, pues distintos arreglos representan el mismo cálculo si se mantienen las conexiones entre los nodos y sus ganancias respectivas.

Estos esquemas no sólo describen un procedimiento eficiente para calcular la DFT, sino que también sugieren una forma de almacenar los datos originales y los resultados de los cálculos intermedios en vectores.

Para ver el procedimiento, tomaremos como ejemplo el diagrama de la figura 2.3.1.9 que describe el cálculo de una DFT de $N = 8$ puntos mediante el algoritmo FFT de diezmado en el tiempo base 2. Dentro de este grafo aparecen una serie de nodos que representan las variables necesarias para realizar el programa, es decir, la cantidad de almacenamiento necesario. El algoritmo se divide en $h = log_2 N$ etapas. En el caso concreto que estamos estudiando el número de etapas es 3. Cada etapa obtiene N valores complejos de salida, a partir de otros N valores complejos de entrada.

Una forma de almacenar los datos y resultados necesarios para realizar el programa que implemente este algoritmo es utilizar vectores que permitan almacenar los valores complejos de las entradas y las salidas de las distintas etapas. Para tener almacenados N complejos de entrada y otros N de salida sólo son necesarios 2 vectores de registros de almacenamiento complejos para todo el algoritmo, ya que el vector de salida de una etapa se convierte en el vector de entrada de la siguiente, y por tanto, en un principio, sólo se necesitarían el vector con los resultados de la etapa anterior y el correspondiente a los resultados que se estén calculando en la etapa actual.

Los elementos de cada vector de salida se obtienen como resultado de aplicar estructuras mariposa como la de la figura 2.4.1.1 sobre los elementos del vector de entrada. Denominaremos $X_i\,[\,l]$ $(l = 0,1,\dots,N-1)$, a la secuencia de números complejos resultantes de la i-ésima etapa de cálculo $(i = 1,2,\dots,h)$. Por tanto, en la primera etapa, $X_0\,[l]$ será la secuencia de entrada $x[n]$, mientras que en la última etapa, $X_h\,[l]$ se corresponderá con los coeficientes de la DFT.

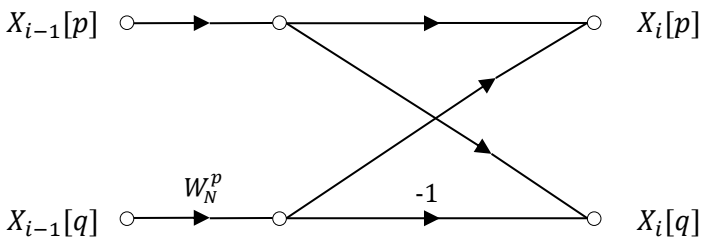


Figura 2.4.1.1: Diagrama mariposa

En la figura 2.4.1.1 puede observarse que para obtener los valores de cualquier par de elementos de una etapa $(X_i[p]\ y\ X_i[q])$ sólo se necesitan los valores de los elementos que están en la misma posición en el vector de la etapa anterior $(X_{i-1}[p]\ y\ X_{i-1}[q])$. Por tanto, si $X_i[p]\ y\ X_i[q]$ se almacenan en los mismos registros que $X_{i-1}[p]\ y\ X_{i-1}[q]$, respectivamente, sólo se necesitan N registros complejos de

almacenamiento para realizar el cálculo completo, es decir, un único vector de registros de almacenamiento complejo. Los cálculos realizados de esta forma reciben el nombre de cómputo en el mismo lugar, ya que los resultados se almacenan en el mismo vector que contenía los datos.

Por último, analizaremos el diagrama de la figura 2.4.1.2 que describe el cálculo de una DFT de $N = 8$ puntos mediante el algoritmo FFT de diezmado en frecuencia base 2. En este caso, el cálculo básico tiene de nuevo la estructura de una mariposa, aunque de forma distinta a la que surge en los algoritmos de diezmado en tiempo. Sin embargo, tal y como se aprecia en la figura 2.4.1.3, por la naturaleza del cálculo de la mariposa, los grafos de flujo que se basan en esta estructura se pueden interpretar como algoritmos de cómputo en el mismo lugar.

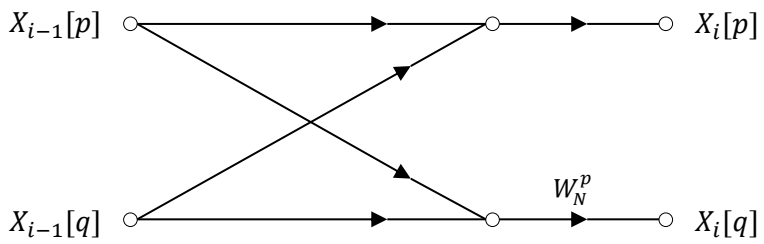


Figura 2.4.1.3: Diagrama mariposa

2.4.2 Orden de la Secuencia de Entrada y Salida

Para que se pueda realizar el cómputo en el mismo lugar es necesario que la secuencia de entrada se almacene en orden bit inverso y la secuencia de salida en orden natural, o viceversa.

Para explicar el significado del término bit inverso, tomaremos como ejemplo una secuencia de 8 muestras. En este caso, es suficiente con 3 dígitos binarios para indexar cada una de las muestras $x[n_2 n_1 n_0]$, donde n_0 es el bit LSB[2], de tal forma que el orden de cada muestra viene dado por una combinación de estos tres bits.

La forma de ordenación más normal es en orden natural, tal y como se muestra en la figura 2.4.2.1. En este caso, si el bit más significativo del índice de la muestra es cero, $x[0\, n_1 n_0]$, ésta pertenecerá a la mitad superior del vector ordenado. Si por el contrario, el índice de la muestra es uno, $x[1\, n_1 n_0]$,

[2] LSB: siglas en inglés de Bit Menos Significativo.

pertenecerá a la mitad inferior. Seguidamente, las subsecuencias de las mitades superior e inferior se ordenan examinando el segundo bit más significativo, y así sucesivamente.

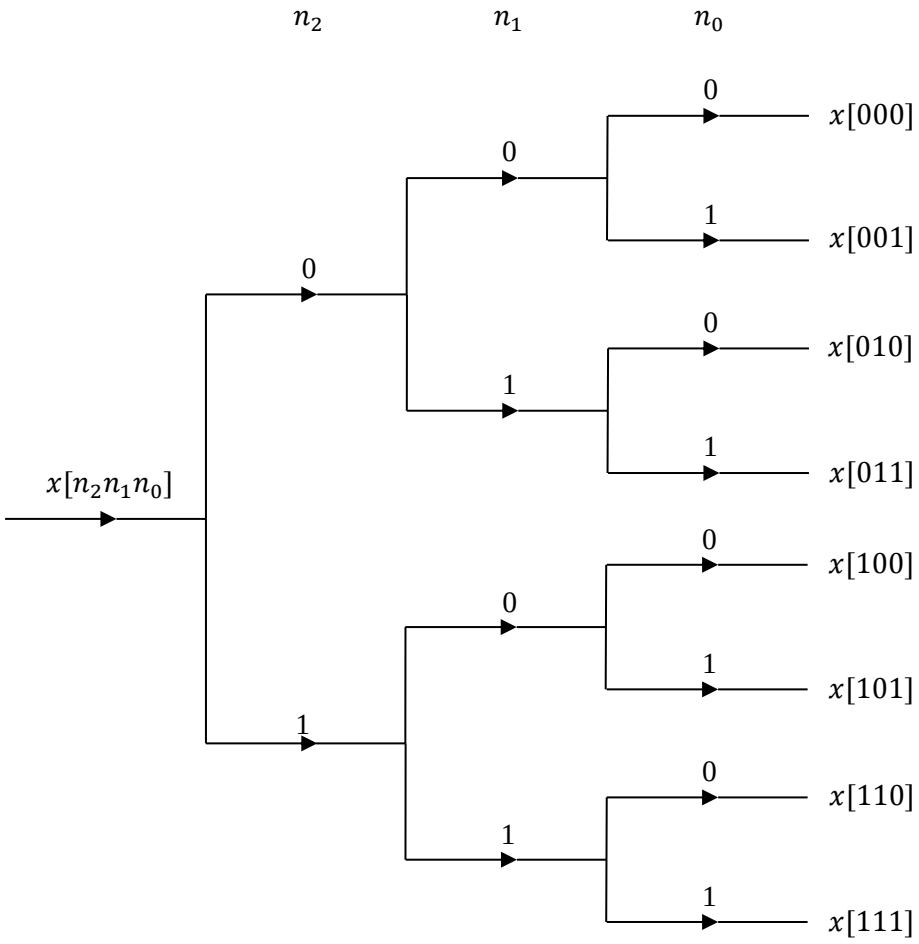


Figura 2.4.2.1: Diagrama en árbol para ordenamiento natural

Otra forma de ordenar la secuencia es en orden bit inverso, como se muestra en la figura 2.4.2.1, donde la filosofía de ordenación es la misma que en el orden natural pero el sentido de la misma es del bit menos significativo (n_0) al más significativo (n_2).

Analizando el diagrama de la figura 2.3.1.9, (DFT de 8 puntos mediante diezmado en tiempo base 2), comprobamos que la entrada se almacena en orden bit inverso, las mariposas se realizan por cómputo en el mismo lugar y, por tanto, la salida se almacena en orden natural.

En el caso del diagrama de la figura 2.3.2.4 (DFT de 8 puntos mediante diezmado en frecuencia base 2), la secuencia de entrada se almacena en orden natural, el cálculo de las mariposas se realiza por cómputo en el mismo lugar y, consecuentemente, la salida se almacena en orden bit inverso.

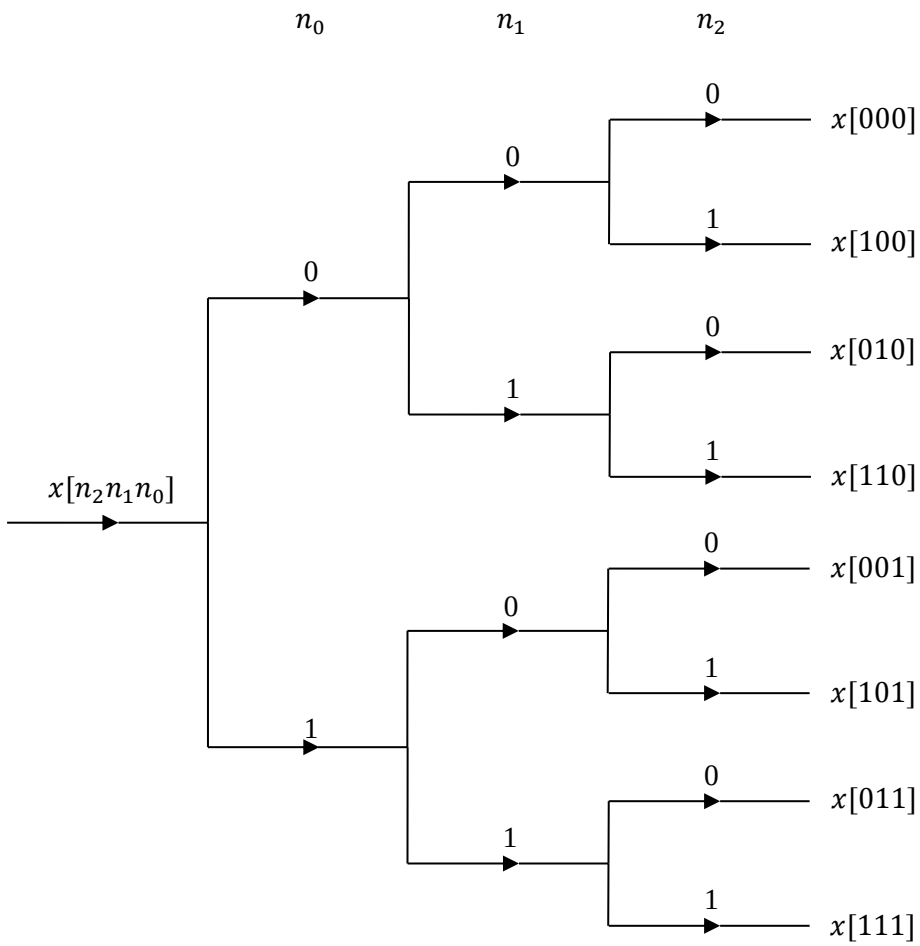


Figura 2.4.2.2: Diagrama en árbol para ordenamiento bit inverso

2.4.3 Formas Alternativas

En los apartados anteriores se han descrito los algoritmos FFT para un orden concreto de las secuencias de entrada y salida

- DET: entrada en orden bit inverso, salida en orden natural.
- DEF: entrada en orden natural, salida en orden bit inverso.

Sin embargo, es posible modificar el orden de las secuencias de entrada y salida, lo que dará lugar a formas alternativas, presentando cada una de ellas diferentes ventajas e inconvenientes.

Retomamos como ejemplo el diagrama de cálculo de una DFT de 8 puntos mediante diezmado en tiempo base 2 representado en figura 2.3.1.9, en el que la secuencia de entrada se dispone en orden bit inverso y secuencia de salida en orden natural. Pues bien, realizando una serie de modificaciones sobre la ubicación de los nodos del grafo podemos conseguir que la secuencia de entrada esté en orden natural y la secuencia de salida en orden correspondiente a inversión de bits, tal y como se muestra en la figura 2.4.3.1:

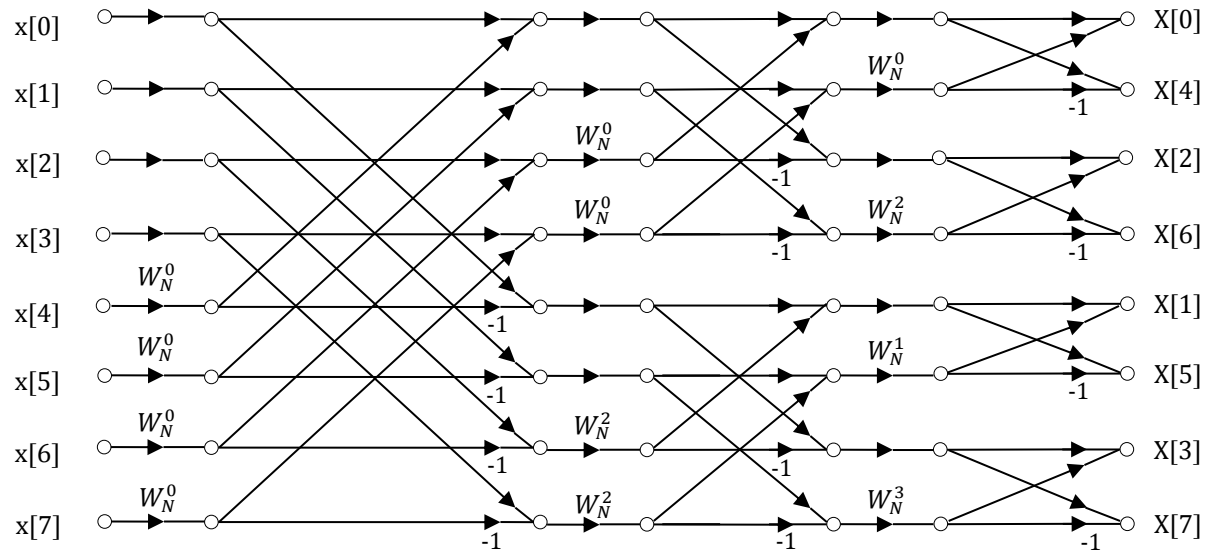


Figura 2.4.3.1: Reestructuración figura 2.3.1.9 con entrada en orden natural y salida en orden bit inverso

Es posible obtener la figura 2.4.3.1 a partir de la figura 2.3.1.9. Para ello, todos los nodos horizontalmente adyacentes a $x[4]$ en la figura 2.3.1.9 se intercambian con todos los nodos adyacentes horizontalmente a $x[1]$. De forma similar, todos los nodos horizontalmente adyacentes a $x[6]$ en la figura 2.3.1.9 se intercambian con los adyacentes horizontalmente a $x[3]$. Los nodos horizontalmente adyacentes a $x[0]$, $x[2]$, $x[5]$ y $x[7]$ no se modifican. El grafo de flujo resultante de la figura 2.4.3.1 corresponde a la forma del algoritmo de diezmado en el tiempo dada originalmente por Cooley y Tukey (1965). Como se puede apreciar, se trata de un algoritmo de cómputo en el mismo lugar, ya que la disposición de los nodos es tal que los nodos de entrada y de salida de cada mariposa son adyacentes horizontalmente. Por tanto, en este caso, también será suficiente con un único vector de registros de almacenamiento complejo.

Las implementaciones con las muestras de entrada en orden natural son muy útiles en procesos en los que haya que concatenar FFT e IFFT3, como es el caso de un filtrado (figura 2.4.3.2). De esta forma, se pueden evitar varias ordenaciones de la secuencia, proceso que puede resultar cada vez más lento a medida que N crece.

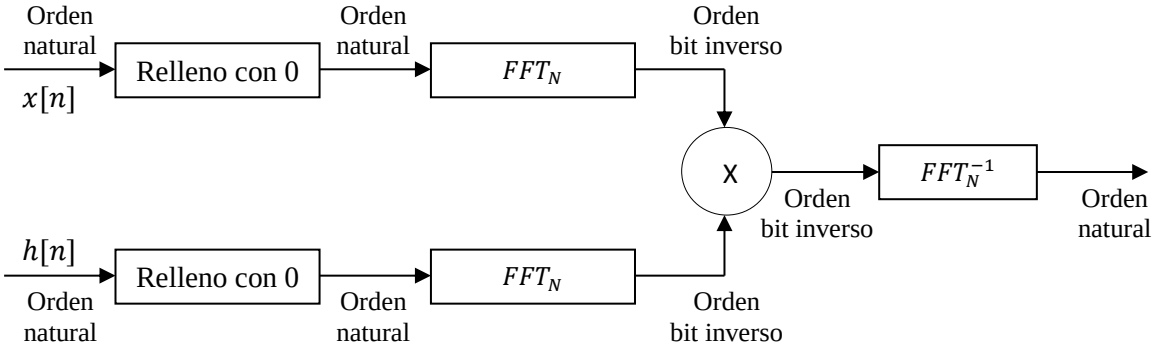


Figura 2.4.3.2: Filtrado usando FFTs.

Existe, por supuesto, una amplia variedad de formas alternativas. Sin embargo, la mayoría de ellas no tienen mucho sentido desde el punto de vista computacional. Por ejemplo, podríamos ordenar los nodos de forma que tanto la secuencia de entrada como la de salida presentaran un orden natural, tal y como se aprecia en la figura 2.4.3.3.

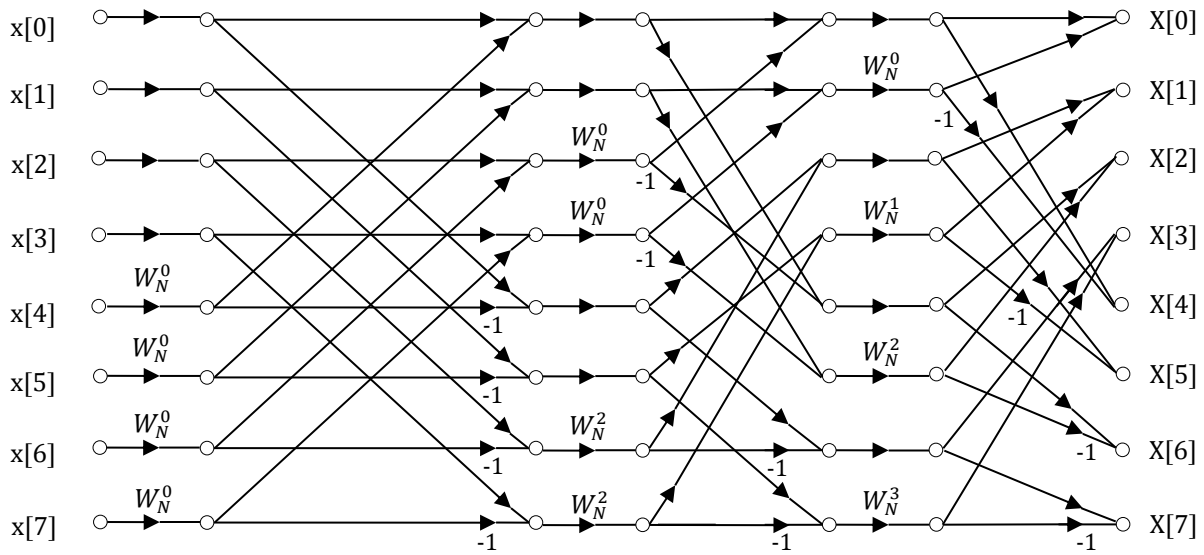


Figura 2.4.3.3: Reconstrucción de la figura 2.3.1.9 con entrada y salida en orden natural

[3] iFFT: siglas en inglés de Transformada Rápida de Fourier Inversa.

En este caso, no sería posible realizar el cómputo en el mismo lugar ya que la estructura en mariposa no se mantiene tras la primera etapa, por lo que se necesitarían dos vectores de registros de almacenamiento complejos de longitud N. Por tanto, esta estructura no presenta ventajas aparentes.

Sin embargo, algunas formas presentan ventajas incluso cuando no es posible el cómputo en el mismo lugar, como es el caso de la estructura de la figura 2.4.3.4. La característica importante de este grafo de flujo es que la geometría es la misma en cada etapa y sólo las ganancias de los arcos cambian de etapa en etapa, lo que permite una simplificación en el acceso a los datos. Este grafo de flujo representa el algoritmo de diezmado en el tiempo dado originalmente por Singleton (1969).

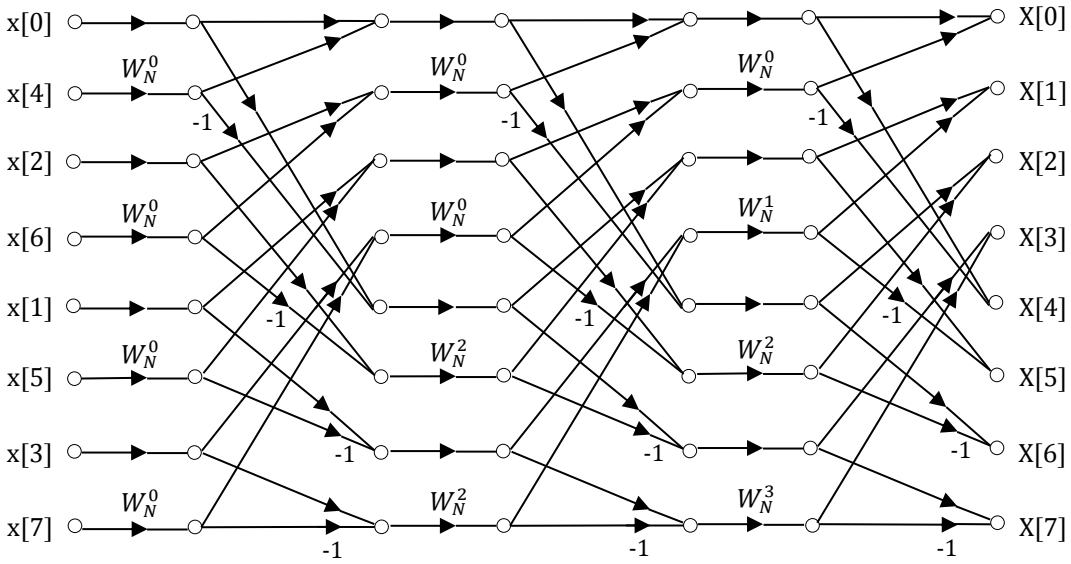


Figura 2.4.3.4: Reconstrucción de la figura 2.3.1.9 con la misma geometría en cada etapa.

Se pueden obtener formas alternativas para el algoritmo de diezmado en frecuencia mediante la trasposición de las formas de diezmado en el tiempo, es decir, invirtiendo la dirección del flujo de las señales e intercambiando la entrada y la salida. A continuación, se muestran varios ejemplos:

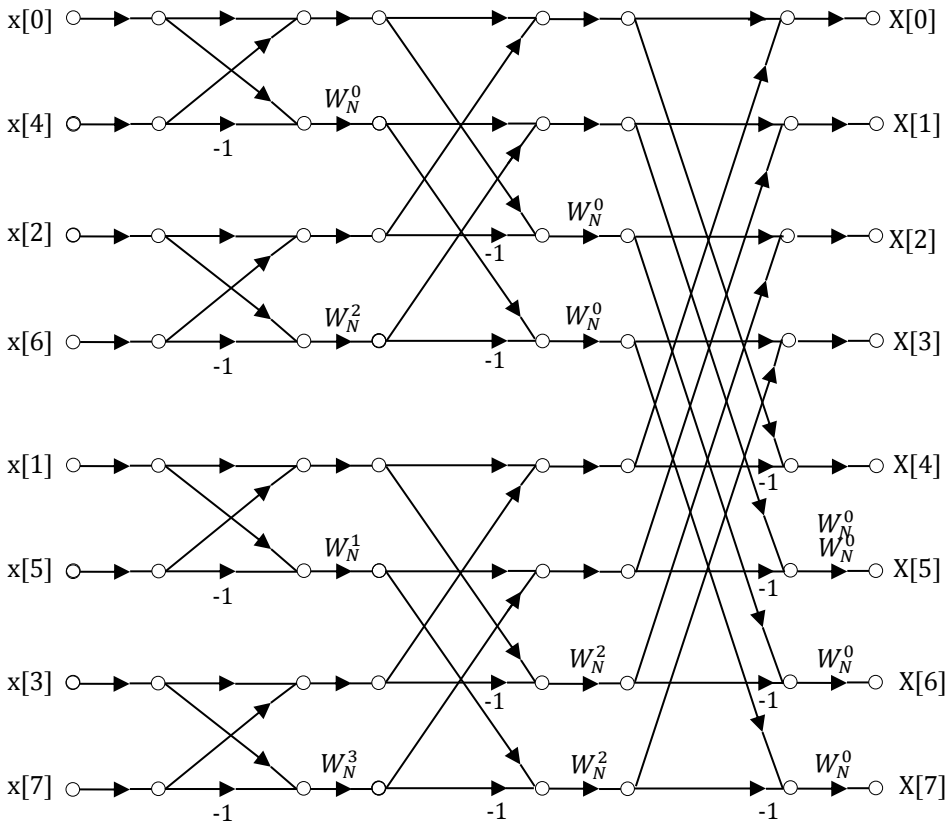


Figura 2.4.3.5: DEF con entrada invertida y salida en orden natural.

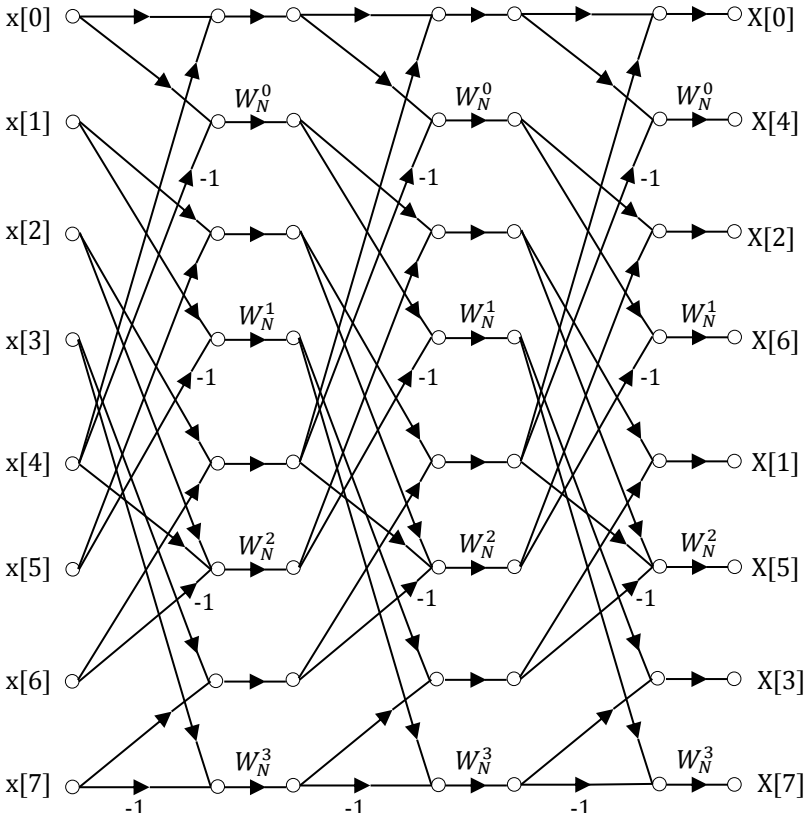


Figura 2.4.3.6: DEF con misma geometría en cada etapa.

2.4.4 Cálculo de Coeficientes Exponenciales

Como hemos visto en apartados anteriores, hacen falta una serie de coeficientes exponenciales W_N^p para $p = 0,1,\dots,\frac{N}{2}-1$. Estos coeficientes se pueden requerir en orden normal o en orden de inversión de bits. Su cálculo es un proceso que debe ser optimizado, para lo que se proponen dos alternativas:

1. **Uso de una tabla:** consiste en calcular previamente todos los coeficientes necesarios en el algoritmo y almacenarlos en una tabla. Esta opción tiene la ventaja de la velocidad, sin embargo, es necesario espacio de almacenamiento adicional; son necesarios N/2 registros de almacenamiento complejos para guardar la tabla completa de valores de W_N^p.

2. **Cálculo recursivo:** consiste en calcular los coeficientes cuando sean necesarios. En una etapa dada todos los coeficientes requeridos son múltiplos de W_N. Si se requieren los coeficientes en orden normal, se puede utilizar la fórmula recursiva 2.4.4.1 para obtener el l-ésimo coeficiente a partir del (l-1)-ésimo coeficiente:

$$W_N^l = W_N \cdot W_N^{l-1} \qquad \textbf{(2.4.4.1)}$$

Puede comprobarse claramente que los algoritmos que requieren los coeficientes en orden de inversión de bits no se adaptan bien a esta solución.

Este método minimiza la memoria necesaria optimizando también el tiempo de cálculo. Sin embargo, hay que tener en cuenta que los coeficientes sufren un error de cuantificación al almacenarlos, el cual se irá acumulando al ir aplicando la recursión, haciéndose inaceptable si N crece mucho. Este problema tiene un pequeño arreglo que consiste en introducir puntos de inicialización en la recursión, por ejemplo, $W_N^{N/4} = -j$, de forma que los errores no crezcan hasta hacerse inaceptables.

Capítulo 3

Algoritmos Tradicionales OLS y OLA

3. Algoritmos Tradicionales

En aplicaciones prácticas en las que es necesario el filtrado de señales, la secuencia de entrada $x[n]$ suele ser a menudo de muy larga duración.

Debido a las limitaciones de espacio de memoria y tiempo de respuesta de los ordenadores, el filtrado lineal que se realiza mediante la DFT implica la utilización de operaciones con bloques de datos que necesariamente deben ser de tamaño limitado.

En este capítulo del proyecto se analizarán dos métodos que dividen la secuencia de entrada, $x[n]$, en bloques de tamaño fijo, L, antes de ser procesada. Como el filtrado es lineal, los bloques sucesivos se pueden procesar uno a uno mediante la DFT y la IDFT para obtener el bloque de datos de salida. Finalmente, los bloques de salida se unen para formar la secuencia de salida global que será idéntica a la que se habría obtenido si el bloque largo de entrada se hubiese procesado mediante la convolución en el dominio del tiempo. Gracias a los conocidos algoritmos FFT, estas técnicas suponen un gran ahorro computacional, comparado con el cálculo directo del filtrado.

Los dos métodos se denominan **método de solapamiento y almacenamiento (OLS, Overlap-and-Save)** y **método de solapamiento y suma (OLA, Overlap-and-Add)** [8]. En ambos casos suponemos que el filtro es de longitud finita M y, sin pérdida de generalidad, se asume que la longitud de los bloques en los que se fragmenta la señal de entrada es mucho mayor que M, es decir, $L >> M$.

3.1 Método de Solapamiento y Almacenamiento (OLS)

En el método de solapamiento y almacenamiento la señal de entrada $x[n]$ se divide en fragmentos de longitud L. Cada bloque de entrada $x_m[n]$ se forma a partir de $M-1$ muestras del bloque anterior (para evitar el aliasing) seguidas por L muestras nuevas, de tal modo que cada bloque de entrada contendrá $N = L + M - 1$ muestras. Las $M-1$ muestras del primero de los bloques de entrada se considerarán iguales a cero.

$$x_1[n] = \{\underbrace{0, 0, \cdots, 0}_{M-1\ muestras}, x[0], x[1], \cdots, x[L-1]\} \qquad (\mathbf{3.1.1})$$

$$x_2[n] = \{\underbrace{x[L-M+1], \cdots, x[L-1]}_{M-1\ muestras\ de\ x_1[n]}, \underbrace{x[L], \cdots, x[2L-1]}_{L\ muestras\ nuevas}\} \qquad (\mathbf{3.1.2})$$

$$x_3[n] = \{\underbrace{x[2L-M+1],\cdots,x[2L-1]}_{M-1\ muestras\ de\ x_2[n]},\underbrace{x[2L],\cdots,x[3L-1]}_{L\ muestras\ nuevas}\} \qquad (\mathbf{3.1.3})$$

$$\vdots$$

A continuación, se calcula la DFT de N puntos de cada uno de estos bloques de datos. Por otra parte, la respuesta impulsional del filtro $h[n]$ se aumenta en longitud añadiendo $L-1$ ceros, se calcula su DFT de N puntos y se almacena.

El bloque de salida $\hat{y}_m[n]$ para cada bloque de entrada se calcula a través del producto de las dos DFTs de N puntos, $H[k]$ y $X_m[k]$:

$$\hat{Y}_m[k] = H[k]\,X_m[k], \quad k = 0,1,\dots,N-1. \qquad (\mathbf{3.1.4})$$

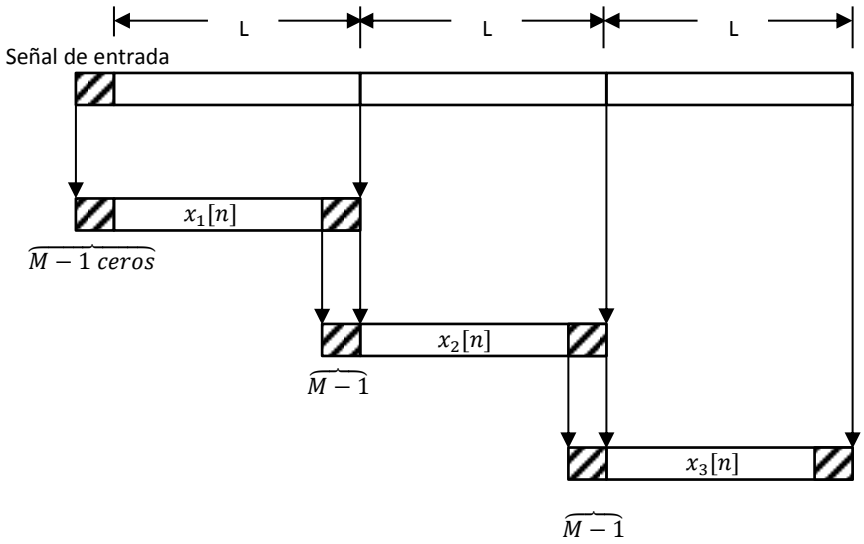
Aplicando a continuación la IDFT de N puntos, se obtiene como resultado:

$$\hat{y}_m[n] = \{\hat{y}_m[0],\hat{y}_m[1],\cdots,\hat{y}_m[M-1],\hat{y}_m[M],\cdots,\hat{y}_m[N-1]\} \qquad (\mathbf{3.1.5})$$

Los primeros $M-1$ puntos de $\hat{y}_m[n]$ están distorsionados por el aliasing y deben ser desechados. Por tanto, los últimos L puntos de $\hat{y}_m[n]$ son exactamente los mismos que se obtendrían a partir de la convolución lineal:

$$y_m[n] = \hat{y}_m[n], \quad n = M, M+1,\dots,N-1. \qquad (\mathbf{3.1.6})$$

Por último, concatenando los bloques de salida $y_m[n]$ se obtiene la secuencia de salida total $y[n]$.



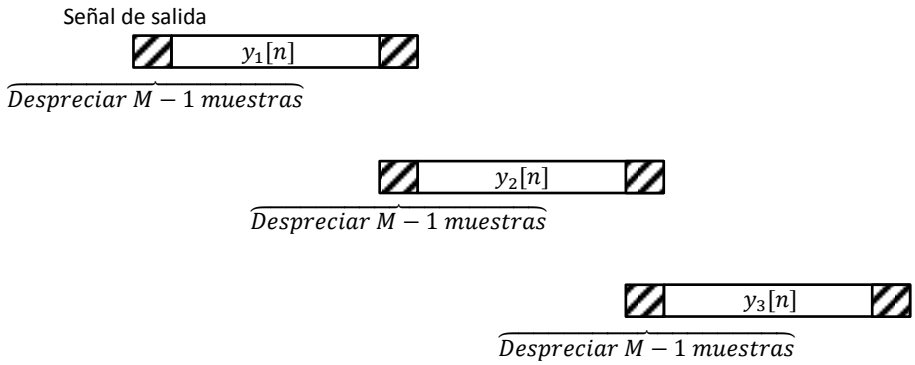


Figura 3.1.1: Esquema Método Solapamiento y Almacenamiento (OLS)

3.2 Método de Solapamiento y Suma (OLA)

En el método de solapamiento y suma la señal de entrada $x[n]$ se divide en fragmentos de longitud L. A cada bloque de datos se le añade $M-1$ muestras de valor cero, de modo que cada bloque de entrada $x_m[n]$ estará formado por $N = L + M - 1$ muestras.

$$x_1[n] = \{x[0], x[1], \cdots, x[L-1], \underbrace{0, 0, \cdots, 0}_{M-1\ muestras}\} \qquad (\mathbf{3.2.1})$$

$$x_2[n] = \{\underbrace{x[L], x[L+1], \cdots, x[2L-1],}_{L\ muestras} \underbrace{0, 0, \cdots, 0}_{M-1\ muestras}\} \qquad (\mathbf{3.2.2})$$

$$x_3[n] = \{\underbrace{x[2L], \cdots, x[3L-1],}_{L\ muestras} \underbrace{0, 0, \cdots, 0}_{M-1\ muestras}\} \qquad (\mathbf{3.2.3})$$

$$\vdots$$

A continuación, se calcula la DFT de N puntos de cada uno de estos bloques de datos. Por otra parte, la respuesta impulsional del filtro $h[n]$ se aumenta en longitud añadiendo $L-1$ ceros, se calcula su DFT de N puntos y se almacena.

Las dos DFTs anteriores se multiplican para formar el bloque de datos de salida:

$$Y_m[k] = H[k]\, X_m[k], \quad k = 0,1, \ldots, N-1. \qquad (\mathbf{3.2.4})$$

La IDFT da como resultado bloques de salida de longitud N que no están afectados por aliasing, debido a que tanto el tamaño de las DFTs como el de las IDFTs es $N = L + M - 1$ muestras, y los

bloques de datos de entrada $x_m[n]$ se aumentaron de longitud, añadiendo ceros, hasta un valor de N muestras.

La secuencia de salida total $y[n]$ se obtiene concatenando los bloques de salida $y_m[n]$, de modo que las $M-1$ últimas muestras de un bloque se solapen y se sumen a las $M-1$ primeras muestras del bloque siguiente. De aquí, que este método sea conocido como de solapamiento y suma:

$$y[n] = \{y_1[0], y_1[1], \cdots, y_1[L-1], y_1[L] + y_2[0], y_1[L+1] + y_2[1], \cdots, y_2[M-1], y_2[M], \cdots\} \quad (\mathbf{3.2.5})$$

La fragmentación de los datos de entrada y la unión de los bloques de datos de salida para formar la secuencia de salida total se muestra gráficamente a continuación:

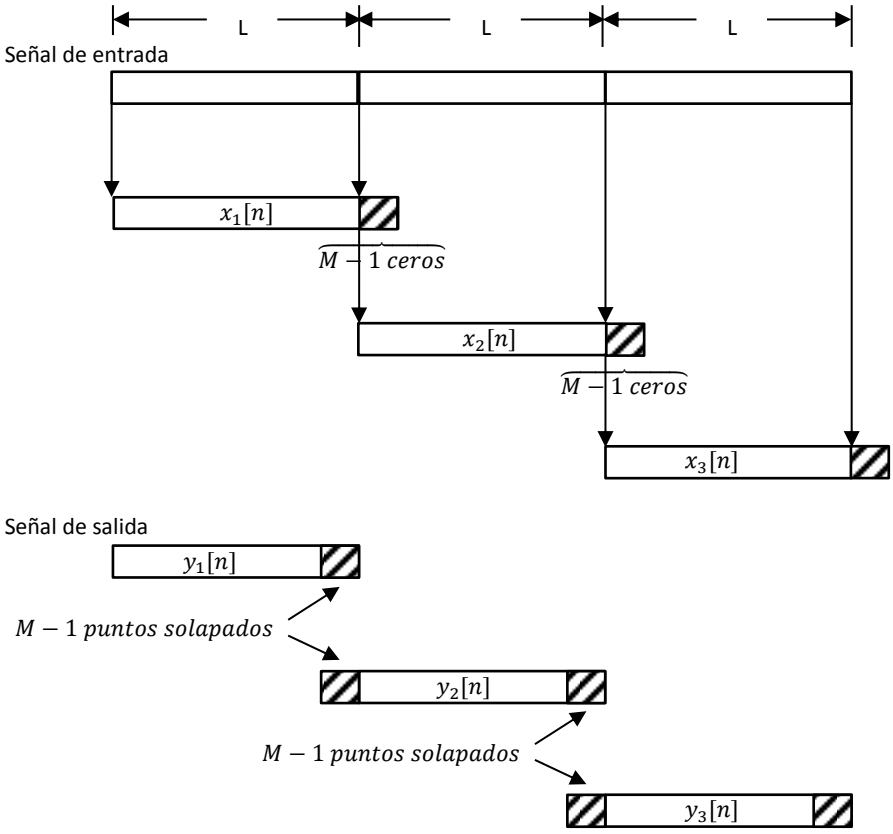


Figura 3.2.1: Esquema Método Solapamiento y Suma (OLA)

3.3 Formulación de Métodos Tradicionales OLS y OLA

En este apartado se formularán los tradicionales métodos OLS y OLA como cálculos de producto matriz-vector, donde la matriz es circulante y el vector está compuesto por bloques de entrada. Esto nos conducirá, naturalmente, a algoritmos modificados descritos en el capítulo siguiente.

Consideremos el problema del filtrado de una secuencia de longitud infinita, $x[n]$, con un filtro FIR de longitud N, definido por los coeficientes $h_n,\ n = 0,1,\dots,N-1$. Asumiendo que la secuencia de salida, $y[n]$, se compone por bloques, podemos escribir la siguiente expresión para la operación de filtrado por bloque:

$$\boldsymbol{y_k} = [\boldsymbol{U} \quad \boldsymbol{L}]\begin{bmatrix}\boldsymbol{x_k}\\ \boldsymbol{x_{k-1}}\end{bmatrix} = \boldsymbol{U x_k} + \boldsymbol{L x_{k-1}} \qquad \textbf{(3.3.1)}$$

Dónde k es el índice de bloque, x_k y x_{k-1} son los bloques de entrada de longitud N actual y previo, e y_k se corresponde con el bloque de salida actual:

$$\boldsymbol{x_k} \triangleq [x[kN], x[kN-1], \dots, x[kN-N+1]]^T \qquad \textbf{(3.3.2)}$$

$$\boldsymbol{x_{k-1}} \triangleq [x[kN-N], x[kN-N-1], \dots, x[kN-2N+1]]^T \qquad \textbf{(3.3.3)}$$

$$\boldsymbol{y_k} \triangleq [y[kN], x[kN-1], \dots, x[kN-N+1]]^T \qquad \textbf{(3.3.4)}$$

$\boldsymbol{U}$ y $\boldsymbol{L}$ denotan las matrices triangulares superior e inferior $N \times N$, constituidas por los coeficientes del filtro:

$$[\boldsymbol{U}|\boldsymbol{L}] \triangleq \left[\begin{array}{cccc|cccc} h_0 & h_1 & \cdots & h_{N-1} & 0 & \cdots & 0 & 0 \\ 0 & h_0 & \cdots & h_{N-2} & h_{N-1} & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & h_0 & h_1 & \cdots & h_{N-1} & 0 \end{array}\right] \qquad \textbf{(3.3.5)}$$

Se añade una columna de ceros a la derecha de la matriz para igualar las dimensiones de ambas matrices. En el método tradicional OLS se implementa este filtrado por bloque extendiendo las dimensiones $N \times 2N$ de la matriz $[\boldsymbol{U}|\boldsymbol{L}]$, obteniendo como resultado una matriz $\boldsymbol{C}$, circulante, de dimensiones $2N \times 2N$. A partir de esto, podemos escribir la ecuación 3.3.1 como sigue:

$$\begin{bmatrix} \boldsymbol{y_k} \\ \times \end{bmatrix} = \begin{bmatrix} \boldsymbol{U} & \boldsymbol{L} \\ \boldsymbol{L} & \boldsymbol{U} \end{bmatrix} \begin{bmatrix} \boldsymbol{x_k} \\ \boldsymbol{x_{k-1}} \end{bmatrix} = \boldsymbol{C_X} \qquad \textbf{(3.3.6)}$$

Donde la × denota las N salidas que son descartadas.

El producto de la matriz circulante $\boldsymbol{C}$ y el vector de entrada $\boldsymbol{x}$, formado por los bloques actual y previo, es calculado mediante técnicas FFT.

Por otra parte, en el método tradicional OLA, en vez de usar dos bloques consecutivos como vector de entrada, se anexa un bloque de N ceros al bloque actual y se evalúa el producto matriz-vector, tal y como se muestra a continuación:

$$\begin{bmatrix} \boldsymbol{p_k} \\ \boldsymbol{q_k} \end{bmatrix} = \begin{bmatrix} \boldsymbol{U} & \boldsymbol{L} \\ \boldsymbol{L} & \boldsymbol{U} \end{bmatrix} \begin{bmatrix} \boldsymbol{x_k} \\ \boldsymbol{0} \end{bmatrix} = \begin{bmatrix} \boldsymbol{Ux_k} \\ \boldsymbol{Lx_k} \end{bmatrix} \qquad \textbf{(3.3.7)}$$

Como se puede comprobar, la matriz circulante es la misma que para el caso OLS y el producto matriz-vector se calcula también mediante métodos FFT.

En este caso, la salida actual del filtro se obtiene mediante la suma de $\boldsymbol{p_k}$ y la versión retrasada de $\boldsymbol{q_k}$:

$$\boldsymbol{y_k} = \boldsymbol{Ux_k} + \boldsymbol{Lx_{k-1}} = \boldsymbol{p_k} + z^{-1}\boldsymbol{q_k} \qquad \textbf{(3.3.8)}$$

Donde z^{-1} denota un retraso de bloque unitario.

Capítulo 4

Nuevos Algoritmos

4. Nuevos Algoritmos

En la mayor parte de los problemas prácticos, la DFT se aplica sobre secuencias reales. Sin embargo, en la derivación de los algoritmos DFT a menudo se asume que la secuencia de entrada es compleja.

Aunque es usual encontrar algoritmos específicamente diseñados para el cálculo de la DFT de secuencias reales, la tendencia en estos métodos es manejar la DFT de la secuencia de entrada y de salida separadamente, y no atendiendo al proceso de filtrado en sí.

En este capítulo, se describen una serie de algoritmos que realizan el proceso de filtrado a partir de versiones complejas de la secuencia real de entrada ([9] y [10]). En los computadores de hoy día, el tiempo de cálculo con datos complejos sólo es un poco superior al tiempo de cálculo con datos reales. Aun así, embeber una DFT real en una DFT compleja equivalente es mucho más eficiente.

4.1 Algoritmo de Narasimha

El filtrado de señales reales de larga duración mediante un filtro de respuesta impulsiva finita (FIR, Finite Impulsive Response) con coeficientes reales es un problema común en el procesamiento de señales. Para filtros de larga duración, se suelen emplear métodos de convolución muy rápidos basados en los algoritmos OLS y OLA. Tal y como se describe en el capítulo 3, en estos métodos la señal de entrada es segmentada en bloques que son procesados mediante algoritmos FFT. En la mayoría de los casos, la longitud de estos bloques es la misma que la del filtro.

Está comprobado que la eficiencia de la FFT es mayor cuando las señales de entrada y salida son complejas. El objetivo de este apartado es el de presentar un algoritmo de este tipo, cuya clave es la posibilidad de organizar dos bloques de entrada consecutivos como una secuencia compleja antes del primer proceso FFT, ejecutar uno de los métodos tradicionales (OLS u OLA), y obtener dos bloques de salida consecutivos separando las partes real e imaginaria de la salida del segundo proceso FFT.

4.1.1. Métodos OLS y OLA Modificados

En la versión tradicional de ambos métodos, cuya formulación se detalla en el capítulo anterior, se considera la entrada segmentada en bloques de N muestras. Sin embargo, se pasan dos bloques de entrada al mismo tiempo y se generan $2N$ muestras de salida, empleando transformadas complejas de $2N$ puntos. Se asume que los coeficientes del filtro y los datos son reales.

En la versión modificada del método OLS, se consideran tres bloques de entrada de N muestas consecutivos en cada paso: el actual y los dos previos, denotados por $\boldsymbol{x_k}$, $\boldsymbol{x_{k-1}}$ y $\boldsymbol{x_{k-2}}$, respectivamente.

Para procesar estos bloques, se calcula el siguiente producto matriz-vector:

$$\begin{bmatrix} \boldsymbol{p_k} \\ \times \end{bmatrix} = \begin{bmatrix} \boldsymbol{U} & \boldsymbol{L} \\ \boldsymbol{L} & \boldsymbol{U} \end{bmatrix} \begin{bmatrix} \boldsymbol{x_k} + j\boldsymbol{x_{k-1}} \\ \boldsymbol{x_{k-1}} + j\boldsymbol{x_{k-2}} \end{bmatrix} \qquad \textbf{(4.1.1.1)}$$

En este caso, se combinan dos bloques de entrada consecutivos para formar una secuencia compleja, pero la matriz circulante sigue siendo la misma que la de los métodos tradicionales. El producto matriz-vector se calcula usando transformadas complejas de 2N puntos.

Ahora, las primeras N muestras de salida de la ecuación 4.1.1.1 vienen dadas por:

$$\boldsymbol{p_k} = \boldsymbol{U x_k} + \boldsymbol{L x_{k-1}} + j(\boldsymbol{U x_{k-1}} + \boldsymbol{L x_{k-2}}) \qquad \textbf{(4.1.1.2)}$$

Comparando esto con la ecuación 3.3.1, es evidente que las $2N$ muestras que se requieren para los bloques $\boldsymbol{y_k}$ e $\boldsymbol{y_{k-1}}$ pueden ser calculados por separación de las partes real e imaginaria del vector $\boldsymbol{p_k}$. En el siguiente paso, el procedimiento se repite después de saltar dos bloques de N muestras.

El método OLA modificado se deriva de una forma similar. Se combinan dos bloques de entrada consecutivos para formar una secuencia compleja, pero la matriz circulante sigue siendo la misma que la de los métodos tradicionales. El producto matriz-vector se calcula usando transformadas complejas de $2N$ puntos.

$$\begin{bmatrix} \boldsymbol{p_k} \\ \boldsymbol{q_k} \end{bmatrix} = \begin{bmatrix} \boldsymbol{U} & \boldsymbol{L} \\ \boldsymbol{L} & \boldsymbol{U} \end{bmatrix} \begin{bmatrix} \boldsymbol{x_k} + j\boldsymbol{x_{k-1}} \\ 0 \end{bmatrix} = \begin{bmatrix} \boldsymbol{U x_k} + j\boldsymbol{U x_{k-1}} \\ \boldsymbol{L x_{k-1}} + j\boldsymbol{L x_{k-2}} \end{bmatrix} \qquad \textbf{(4.1.1.3)}$$

A continuación, separamos las partes real e imaginaria de $\boldsymbol{p_k}$ y $\boldsymbol{q_k}$, para obtener $\boldsymbol{U x_k}$, $\boldsymbol{U x_{k-1}}$, $\boldsymbol{L x_k}$ y $\boldsymbol{L x_{k-1}}$. Los bloques de salida actual $\boldsymbol{y_k}$ y anterior $\boldsymbol{y_{k-1}}$, se obtienen combinando estos términos como sigue:

$$\boldsymbol{y_k = Ux_k + z^{-1}(Lx_k)} \qquad \textbf{(4.1.1.4)}$$

$$\boldsymbol{y_{k-1} = Ux_{k-1} + z^{-1}(Lx_{k-1})} \qquad \textbf{(4.1.1.5)}$$

De nuevo, z^{-1} denota un retraso de un bloque. Así, este procedimiento se repite después de saltar dos bloques de N muestras.

4.1.2. Reducción del Retardo de Procesamiento

Es evidente que el retraso en el procesamiento de los algoritmos modificados es de $2N$ muestras, mientras que en los algoritmos tradicionales este retraso se reduce a N muestras. La técnica estándar que permite disminuir este retraso, a expensas de introducir cálculos adicionales, consiste en usar la descomposición multifase enfocada al filtrado en bloques.

Para ilustrar esta técnica, asumiremos que se quiere reducir el retraso en el algoritmo OLS modificado a N muestras. Para lograr esto, se segmenta la entrada en bloques de M muestras, donde $M = N/2$, en vez de segmentarla en bloques de N muestras como se hacía anteriormente. En esta nueva situación, la longitud de los vectores complejos de entrada en la ecuación 4.1.1.1 será N, por lo que la dimensión de la matriz circulante formada por los coeficientes del filtro necesitará ser reducida acordemente. Esto se consigue combinando los coeficientes del filtro, dos en un tiempo, empleando para ello los componentes multifase, al igual que en la ecuación 3.3.8, como sigue a continuación:

$$g_k(z) \triangleq h_k + h_{M+k}z^{-1}, \quad k = 0,1,\dots,M-1 \qquad \textbf{(4.1.2.1)}$$

Con estos cambios, podemos reformular los cálculos para el algoritmo OLS descritos en la ecuación 4.1.1.1 como un producto de matrices $\boldsymbol{C}(z)\boldsymbol{X}(z)$, en el dominio z, tomando la transformada z del vector a ambos lados.

La matriz circulante $\boldsymbol{C}(z)$ en este caso es idéntica a la matriz circulante $\boldsymbol{C}$ anterior, con la excepción de que las entradas h_k son reemplazadas por $g_k(z)$. Aunque esta matriz es función de z, sus coeficientes continúan siendo reales. La evaluación del nuevo producto matriz-vector $\boldsymbol{C}(z)\boldsymbol{X}(z)$ consistirá en una DFT compleja de N puntos, seguido por N filtros complejos y una IDFT compleja de N puntos. Podemos, por tanto, obtener los dos bloques de salida de longitud M separando las partes real e imaginaria de la primera mitad del vector de salida, siendo la segunda mitad descartada.

4.2 Algoritmo Propuesto

En los problemas prácticos, las señales para las que las DFTs son calculadas a menudo son reales. Por el contrario, en la derivación de los algoritmos DFTs es normalmente asumido que los datos de entrada son complejos. Además, la aritmética compleja requiere sólo de un poco de más tiempo de procesamiento que la aritmética real, con los computadores de hoy en día. En definitiva, embeber una DFT real en una DFT compleja equivalente proporciona mejores resultados.

Es común encontrar algoritmos que son específicamente diseñados para el cálculo de DFTs de datos reales. Sin embargo, el enfoque típico a la hora de filtrar una señal real es manejar la DFT de las secuencias de entrada y salida separadamente y no prestando atención a eficientar el proceso de filtrado en sí mismo.

En el algoritmo de Narasimha desarrollado en el apartado anterior, haciéndose uso de versiones complejas de señales reales, se logra una considerable reducción del tiempo de ejecución con respecto a los métodos tradicionales de OLS y OLA. Sin embargo, el sacrificio está en un incremento de la latencia, que se hace dos veces superior. Mientras la latencia no es un problema en muchas aplicaciones, supone un gran inconveniente para muchas otras, particularmente en sistemas en tiempo real.

En este apartado, se desarrolla una técnica que emplea DFTs complejas para el filtrado de señales reales. Este método presenta la misma latencia que OLS y OLA y está basado directamente en la DFT tradicional. Además, a diferencia de los métodos OLS y OLA tracionales y modificados, el algoritmo propuesto no está limitado al filtrado en bloques y puede ser usado para cualquier tipo de proceso de convolución. Se demuestra que el número total de operaciones aritméticas se reduce, siendo el hecho más destacado que el tiempo de ejecución es inferior al de los métodos de apartados anteriores. Por tanto, el algoritmo que se propondrá puede resultar útil para aplicaciones que emplean señales reales y, particularmente, para sistemas en tiempo real como reverberaciones acústicas, donde la latencia debe ser pequeña.

En los subapartados siguientes se revisará el método tradicional para cálculo de la DFT de una secuencia real a partir de una versión compleja de esta, y se hará uso de este resultado básico para derivar el método propuesto.

4.2.1. Método Tradicional para Cálculo de la DFT de Señales Reales

Se considera una secuencia real $x[n]$ de longitud L, definida en $0 \leq n \leq L-1$. La intención es clacular $X[k]$, la DFT de Mpuntos de $x[n]$, donde $M \geq L$ (por simplicidad se asume que My L son impares, sin pérdida de generalidad). La forma en la que este método tradicional calcula este resultado es la siguiente:

En primer lugar, dividimos $x[n]$ en dos subsecuencias; $x_e[n]$ y $x_o[n]$, que contienen las muestras pares e impares de $x[n]$, respectivamente. A partir de estas subsecuencias, construimos la señal compleja:

$$x_c[n] = x_e[n] + jx_o[n] \qquad \textbf{(4.2.1.1)}$$

Donde $j = \sqrt{-1}$, y calculamos $X_c[k]$, la DFT de $M/2$puntos de $x_c[n]$.

A partir de $X_c[n]$ y las propiedades de simetría de la DFT, las DFTs de $M/2$ puntos de $x_e[n]$ y $x_o[n]$ se pueden calcular como sigue:

$$X_e[k] = \frac{1}{2} \cdot \left[X_c[k] + X_c{}^*[(-k)]_{M/2}\right] \qquad \textbf{(4.2.1.2)}$$

$$X_o[k] = -j \cdot \frac{1}{2} \cdot \left[X_c[k] - X_c{}^*[(-k)]_{M/2}\right] \qquad \textbf{(4.2.1.3)}$$

Donde $[(\cdot)]_{M/2}$ denota una trasposición circular módulo $M/2$.

Finalmente, $X[k]$ puede ser calculada como:

$$X[k] = X_e[k] + X_o[k] \cdot e^{-j\left(\frac{2\pi k}{M}\right)} \quad k = 0,1,\dots,M/2 \qquad \textbf{(4.2.1.4)}$$

Debido a la simetría conjugada de $X[k]$, el resto de muestras $k = \frac{M}{2} + 1,\dots,M-1$, no es necesario calcularlas explícitamente.

Como ya es conocido, la FFT y su inversa IFFT están estrechamente relacionadas. Por tanto, la técnica anterior es aconsejable para calcular la IFFT realizando tan sólo unas pequeñas modificaciones.

4.2.2. Convolución de Señales Reales usando DFTs

En este apartado, consideramos que queremos realizar un proceso de filtrado de datos reales. La longitud de la secuencia de entrada es L y la del filtro N. Con estos datos, DFTs de $M \geq L + N - 1$ puntos se necesitan para asegurar un cálculo correcto de la convolución líneal.

El método tradicional consiste en los siguientes pasos:

1. Evaluar $\mathrm{H[k]}$ y $\mathrm{X[k]}$, las FFTs de M puntos de la secuencia de entrada y del filtro, respectivamente, haciendo uso de las ecuaciones 4.2.1.1, 4.2.1.2, 4.2.1.3 y 4.2.1.4.
2. Calcular $\mathrm{Y[k]= X[k] \cdot H[k]}$.
3. Calcular la IFFT de M puntos de $\mathrm{Y[k]}$ para obtener la secuencia de salida, $\mathrm{y[n]}$, empleando de nuevo los resultados del apartado anterior.

Obviamente, este procedimiento es fácilmente aplicable en el filtrado en bloques, como OLS y OLA. Dado que $H[k]$ es usado en cada bloque, se podría precalcular para así reducir los requerimientos computacionales.

Si analizamos cuidadosamente este método, encontramos algunas operaciones innecesarias. Hasta ahora, necesitamos obtener $X[k]$ e $Y[k]$, siendo estos únicamente resultados intermedios. Por el contrario, sería posible completar el filtrado completo a partir de $X_c[k]$ e $Y_c[k]$, logrando una reducción en el procesamiento de las secuencias de entrada y salida. Como contrapartida, esto conlleva algunas operaciones adicionales con la respuesta impulsiva del sistema, sin embargo, pueden ser preprocesadas antes del filtrado, dejando de suponer un problema importante.

A partir de la ecuación 4.2.1.4, podemos afirmar lo siguiente:

$$Y_e[k] + Y_o[k] \cdot e^{-j\left(\frac{2\pi k}{M}\right)} = \left[X_e[k] + X_o[k] \cdot e^{-j\left(\frac{2\pi k}{M}\right)}\right] \cdot \left[H_e[k] + H_o[k] \cdot e^{-j\left(\frac{2\pi k}{M}\right)}\right] \quad \textbf{(4.2.2.1)}$$

Sustituyendo en la expresión anterior k por $k + M/2$, obtenemos:

$$Y_e[k + M/2] - Y_o[k + M/2] \cdot e^{-j\left(\frac{2\pi k}{M}\right)} = \left[X_e[k + M/2] - X_o[k + M/2] \cdot e^{-j\left(\frac{2\pi k}{M}\right)}\right]\left[H_e[k + M/2] - H_o[k + M/2] \cdot e^{-j\left(\frac{2\pi k}{M}\right)}\right] \quad \textbf{(4.2.2.2)}$$

Recordando que todas las DFTs de la ecuación 4.2.2.2 tienen longitud $M/2$, sabemos que $Y_e[k + M/2] = Y_e[k]$, $Y_o[k + M/2] = Y_o[k]$, $X_e[k + M/2] = X_e[k]$, $X_o[k + M/2] = X_o[k]$, $H_e[k + M/2] = H_e[k]$ y $H_o[k + M/2] = H_o[k]$. Por tanto:

$$Y_e[k] - Y_o[k] \cdot e^{-j\left(\frac{2\pi k}{M}\right)} = \left[X_e[k] - X_o[k] \cdot e^{-j\left(\frac{2\pi k}{M}\right)}\right]\left[H_e[k] - H_o[k] \cdot e^{-j\left(\frac{2\pi k}{M}\right)}\right] \qquad \textbf{(4.2.2.3)}$$

La solución a las ecuaciones anteriores es:

$$Y_e[k] = X_e[k] \cdot H_e[k] + X_o[k] \cdot H_o[k] \cdot e^{-j\left(\frac{4\pi k}{M}\right)} \qquad \textbf{(4.2.2.4)}$$

$$Y_o[k] = X_e[k] \cdot H_o[k] + X_o[k] \cdot H_e[k] \qquad \textbf{(4.2.2.5)}$$

Entonces, a partir de las ecuaciones 4.2.1.1, 4.2.2.4 y 4.2.2.5 tendremos que:

$$Y_c[k] = Y_e[k] + j \cdot Y_o[k] = \left[X_e[k] \cdot H_e[k] + X_o[k] \cdot H_o[k] \cdot e^{-j\left(\frac{4\pi k}{M}\right)}\right] + j \cdot \left[X_e[k] \cdot H_o[k] + X_o[k] \cdot H_e[k]\right]$$

(4.2.2.6)

A continuación, sustituyendo los resultados de las ecuaciones 4.2.1.2 y 4.2.1.3 en 4.2.2.6, obtenemos:

$$Y_c[k] = X_c[k] \cdot \left[H_e[k] + j \cdot \frac{1}{2} \cdot H_o[k] \cdot \left(1 - e^{-j\left(\frac{4\pi k}{M}\right)}\right)\right] + j \cdot \frac{1}{2} \cdot X_c{}^*[(-k)]_{M/2} \cdot H_o[k] \cdot \left(1 + e^{-j\left(\frac{4\pi k}{M}\right)}\right)$$

(4.2.2.7)

Para mayor claridad, definimos:

$$H^{(I)}[k] = H_e[k] + j \cdot \frac{1}{2} \cdot H_o[k] \cdot \left(1 - e^{-j\left(\frac{4\pi k}{M}\right)}\right) \qquad \textbf{(4.2.2.8)}$$

$$H^{(II)}[k] = j \cdot \frac{1}{2} \cdot H_o[k] \cdot \left(1 + e^{-j\left(\frac{4\pi k}{M}\right)}\right) \qquad \textbf{(4.2.2.9)}$$

Por tanto, podemos finalmente escribir:

$$Y_c[k] = X_c[k] \cdot H^{(I)}[k] + X_c{}^*[(-k)]_{M/2} \cdot H^{(II)}[k] \qquad \textbf{(4.2.2.10)}$$

De este modo, se completa el objetivo de obtener la secuencia de salida sólo a partir de $X_c[k]$, y no de $X[k]$.

En resumen, el método propuesto para el filtrado de señales reales contempla los siguientes pasos:

1. Calcular $H^{(I)}[k]$ y $H^{(II)}[k]$ a partir de las ecuaciones 4.2.2.8 y 4.2.2.9, respectivamente.
2. Obtener $X_c[k]$, la FFT de $M/2$ puntos de la secuencia compleja de la ecuación 4.2.1.1.
3. Determinar $Y_c[k]$ a partir de la Ecuación 4.2.2.10.
4. Calcular la IFFT de $M/2$ puntos de $Y_c[k]$ para conseguir $y_c[n]$.
5. Extraer las partes real e imaginaria de $y_c[n]$ y combinarlas para finalmente obtener $y[n]$.

Como se puede observar, las operaciones relacionadas con las señales de entrada y salida se reducen a expensas de un incremento en el procesamiento de la repuesta impulsiva del sistema.

4.2.3. Comparación de Requisitos Computacionales

En este apartado consideraremos ambos métodos, tradicional y propuesto, en el contexto de un filtrado en bloques. Además, los compararemos con el algoritmo Narasimha, descrito en el apartado 4.1. Asumiremos que la longitud de cada bloque es la misma que la longitud del filtro, es decir, $L = N$, y, por tanto, se calcularán FFTs de $M = 2N$ puntos. Aunque esta no es una condición necesaria, es la elección típica ya que da la mejor eficiencia sin incurrir en un retraso adicional innecesario.

Además, asumiremos que cada multiplicación compleja cuenta como cuatro multiplicaciones reales y dos sumas reales y que la DFT de Mpuntos calculada mediante una FFT radix-2 requiere de:

- Si la señal es real:
  - $M \cdot log_2(M) - \left(\frac{7}{2}\right) \cdot M + 6$ multiplicaciones reales.
  - $\left(\frac{3}{2}\right) \cdot M \cdot log_2(M) - M/2 + 2$ sumas reales.
- Si la señal es compleja:
  - $2 \cdot M \cdot log_2(M) - 7 \cdot M + 12$ multiplicaciones reales.
  - $3 \cdot M \cdot log_2(M) - 3 \cdot M + 4$ sumas reales.

En primer lugar, evaluaremos la complejidad computacional del algoritmo tradicional. Para computar un bloque de salida de N puntos, necesitaremos lo siguiente:

- Una FFT de $2N$ puntos de entrada real del bloque de entrada actual.
- Multiplicar la FFT anterior por la FFT de $2N$ puntos de la respuesta impulsiva del sistema. Puesto que todas las señales son reales, las DFTs son simétricas y sólo es necesario realizar la mitad de los cálculos.
- Una salida real a partir de la IFFT de $2N$ puntos de la secuencia resultante del paso anterior.

A partir del recuento anterior, el total de requerimientos computacionales para el método tradicional es:

- $4 \cdot N \cdot log_2(N) - 6 \cdot N + 10$ multiplicaciones reales.
- $6 \cdot N \cdot log_2(N) + 6 \cdot N + 2$ sumas reales.

Por otra parte, el método propuesto requiere de lo siguiente:

- Una FFT de N puntos de entrada compleja.
- Dos multiplicaciones complejas y suma compleja de secuencias de N puntos.

Según las necesidades anteriores, el total de requerimientos computacionales para el método propuesto es:

- $4 \cdot N \cdot log_2(N) - 6 \cdot N + 24$ multiplicaciones reales.
- $6 \cdot N \cdot log_2(N) + 8$ sumas reales.

Los resultados anteriores se resumen en la siguiente tabla, en la que se ha incluido en número de operaciones (adiciones y multiplicaciones) para el algoritmo de Narasimha:

	Multiplicaciones	**Sumas**
Tradicional OLS	$4Nlog_2(N) - 6N + 10$	$6Nlog_2(N) + 6N + 2$
Narasimha	$4Nlog_2(N) - 6N + 12$	$6Nlog_2(N) + 2N + 8$
Propuesto	$4Nlog_2(N) - 6N + 24$	$6Nlog_2(N) + 8$

Tabla 4.2.3.1: Número de operaciones aritméticas (multiplicaciones y sumas) por algoritmo.

A continuación, se lista el número total de multiplicaciones y sumas requeridas en cada uno de los algoritmos en función de diferentes valores de la longitud del filtro (N):

N	**Tradicional OLS**		**Narasimha**		**Propuesto**	
	Multiplic.	**Sumas**	**Multiplic.**	**Sumas**	**Multiplic.**	**Sumas**
16	170	482	172	424	184	392
32	458	1154	460	1032	472	968
64	1162	2690	1164	2440	1176	2312
128	2826	6146	2828	5640	2840	5384
256	6666	13826	6668	12808	6680	12296
512	15370	30722	15372	28680	15384	27656
1024	34826	67586	34828	63496	34840	61448
2048	77834	147458	77836	139272	77848	135176
4096	172042	319490	172044	303112	172056	294920
8192	376842	688130	376844	655368	376856	638984
16384	819210	1474562	819212	1409032	819224	1376264

Tabla 4.2.3.2: Complejidad computacional de OLS Tradicional, Narasimha y Propuesto.

A partir de la tabla anterior, podemos observar que el método Propuesto es el que requiere el menor número de adiciones con un mínimo incremento en el número de multiplicaciones.

Capítulo 5

Simulaciones

5. Simulaciones y Resultados

En este apartado se evaluará el tiempo de ejecución de programas para el cálculo de los algoritmos OLS Tradicional, Narasimha y Propuesto, desarrollados en lenguaje C++.

En los tiempos medidos sólo se han tenido en cuenta operaciones realmente involucradas en cada algoritmo. Es decir, no se han tenido en cuenta comandos como declaración de variables, extracción de bloques o los relacionados con cálculos de transformación del filtro.

Dado que el tiempo de ejecución depende altamente del software y hardware del PC, aparte de detallar resultados absolutos, se especificará gráficamente el ratio entre los tres algoritmos mencionados para compararlos.

Para la programación de los algoritmos se ha empleado la herramienta Dev-C++ (software de libre distribución). Se trata de un entorno de desarrollo integrado para programar en lenguaje C/C++. Usa MinGW, que es una versión de GCC (GNU Compiler Collection) como su compilador.

El cálculo de las FFTs implicadas en estos algoritmos se realiza mediante subrutinas implementadas en la librería "FFTW", que permite computar la DFT en una o más dimensiones, para secuencias de entrada, reales o complejas, de tamaño arbitrario. En concreto, se ha empleado la librería "FFTW versión 3.3.4" (última versión oficial de FFTW) [11].

Los códigos fuente correspondientes a la programación de los algoritmos se incluyen en un anexo al final de este documento.

En las siguientes tablas se detallan los resultados obtenidos de la ejecución de los algoritmos para diferentes valores de la longitud de la secuencia de entrada y de la longitud del filtro.

Cada resultado es el tiempo promedio de 100 realizaciones del algoritmo correspondiente.

		Nº de muestras de la secuencia de entrada				
		16384	**32768**	**65536**	**131072**	**262144**
Nº de muestras filtro	**4096**	0,0029	0,0052	0,0096	0,0180	0,0360
	8192	0,0036	0,0059	0,0100	0,0190	0,0370
	16384	N/A	0,0078	0,0130	0,0230	0,0420
	32768	N/A	N/A	0,0190	0,0300	0,0550
	65536	N/A	N/A	N/A	0,0510	0,0830
	131072	N/A	N/A	N/A	N/A	0,1480

Tabla 5.1: Tiempo de ejecución algoritmo OLS Tradicional

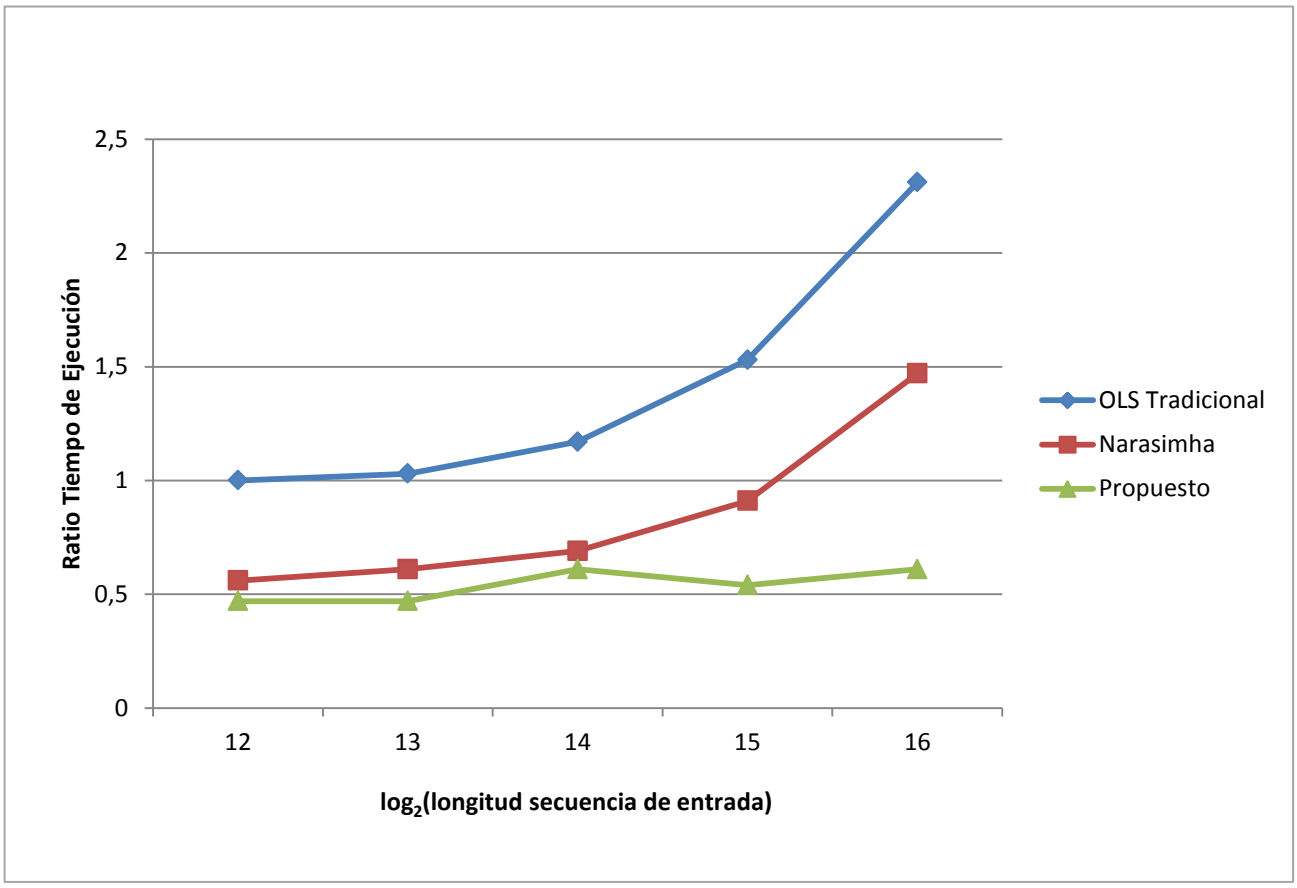
		Nº de muestras de la secuencia de entrada				
		16384	**32768**	**65536**	**131072**	**262144**
Nº de muestras filtro	**4096**	0,0019	0,0031	0,0055	0,0100	0,0200
	8192	0,0026	0,0039	0,0066	0,0110	0,0220
	16384	N/A	0,0058	0,0087	0,0140	0,0250
	32768	N/A	N/A	0,0130	0,0200	0,0330
	65536	N/A	N/A	N/A	0,0350	0,0530
	131072	N/A	N/A	N/A	N/A	0,1000

Tabla 5.2: Tiempo de ejecución algoritmo OLS Modificado (Narasimha)

		Nº de muestras de la secuencia de entrada				
		16384	**32768**	**65536**	**131072**	**262144**
Nº de muestras filtro	**4096**	0,0008	0,0017	0,0047	0,0077	0,0170
	8192	0,0010	0,0018	0,0037	0,0103	0,0170
	16384	N/A	0,0022	0,0041	0,0083	0,0220
	32768	N/A	N/A	0,0052	0,0095	0,0190
	65536	N/A	N/A	N/A	0,0115	0,0220
	131072	N/A	N/A	N/A	N/A	0,0290

Tabla 5.3: Tiempo de ejecución algoritmo Propuesto

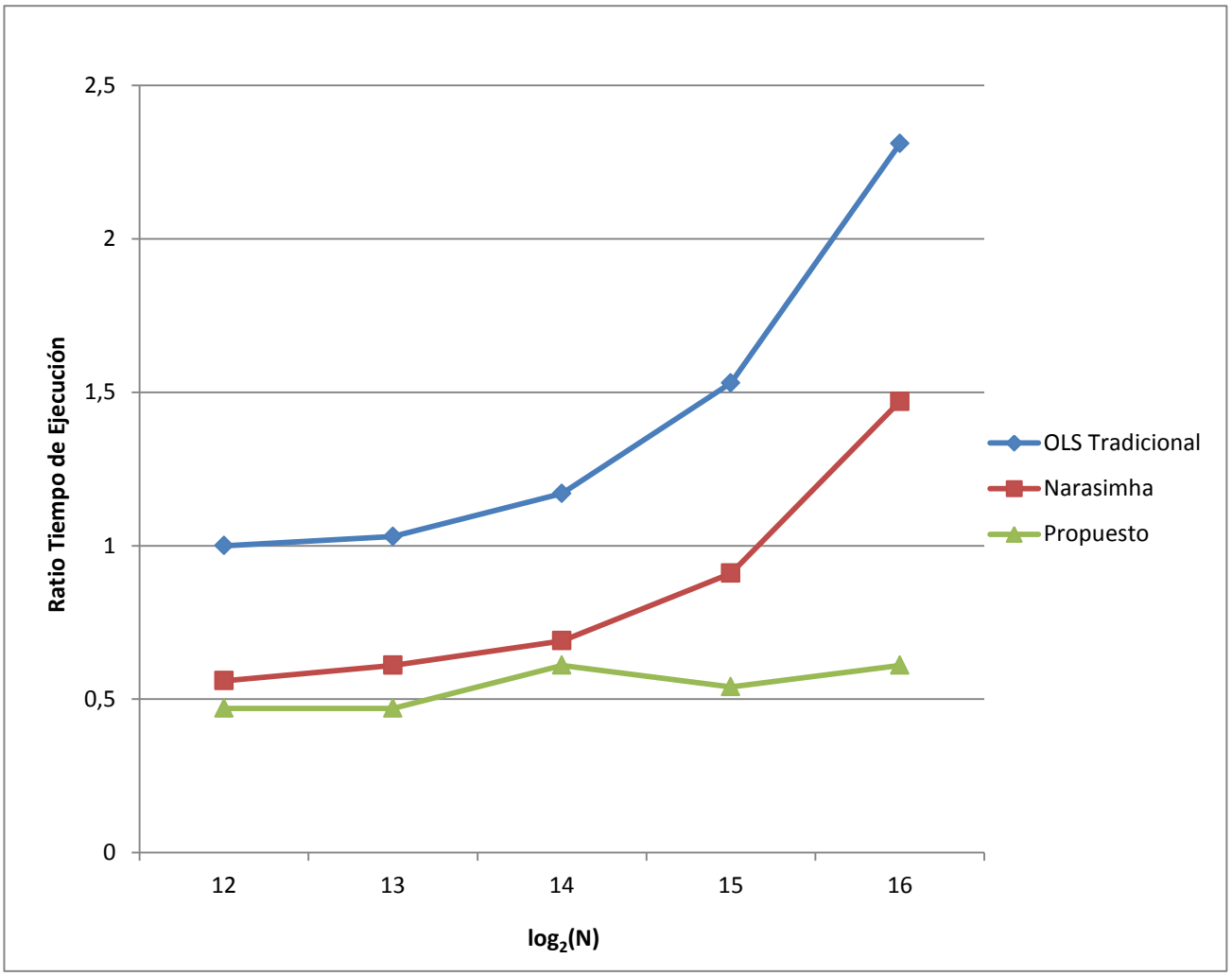
En las siguientes gráficas se representa el ratio de tiempo de ejecución entre los tres algoritmos. En la gráfica 5.1, se representa en función de distintos valores de longitud de la secuencia de entrada y valor fijo de longitud del filtro, y en la gráfica 5.2, se representa para distintos valores de longitud del filtro y valor fijo de longitud de la secuencia de entrada. No se han representado valores de tiempo absolutos, puesto que estos dependen fuertemente del software y hardware que ha dado soporte a las simulaciones.



Gráfica 5.1: Ratio Tiempo de Ejecución en función de longitud de secuencia de entrada.

Cada uno de los resultados que se muestra en la gráfica 5.1 se corresponde a la media de los resultados obtenidos en 100 iteraciones del algoritmo correspondiente. En cada iteración, los valores de la secuencia de entrada y del filtro han sido generados de forma aleatoria. La longitud del filtro se ha fijado a 2^{13} muestras.

Como se puede observar, cuando la longitud de la secuencia de entrada es muy superior a la longitud de la secuencia del filtro, el comportamiento de cada método depende fuertemente de la longitud de la secuencia de entrada. En cualquier caso, los mejores resultados en cuanto a tiempo de ejecución siempre se consiguen con el método Propuesto.



Gráfica 5.2: Ratio Tiempo de Ejecución en función de longitud del filtro.

Al igual que en el caso de la gráfica 5.1, los resultados que se muestran en la gráfica 5.2 se corresponden a la media de los resultados obtenidos en 100 iteraciones del algoritmo correspondiente. En cada iteración, los valores de la secuencia de entrada y del filtro han sido generados de forma aleatoria. La longitud de la secuencia de entrada se ha fijado a 2^{18} muestras.

Como se aprecia, para valores de la longitud de la secuencia de entrada muy superiores a la longitud del filtro, el comportamiento de cada algoritmo manifiesta una dependencia muy leve con la longitud del filtro. En todos los casos, el método Propuesto es el que ofrece los mejores resultados.

Podemos concluir, por tanto, que el tiempo de ejecución disminuye en gran medida cuando recurrimos a FFTs de entradas complejas, siendo el método propuesto el que ofrece mejores resultados en cuanto a tiempo de ejecución, requiriendo de un menor número de operaciones aritméticas.

Capítulo 6

Conclusiones

6. Conclusiones

El filtrado lineal es una operación fundamental en el procesamiento digital de señales que mayormente se aplica a señales reales, por lo que el estudio de herramientas para su cálculo de forma eficiente se convierte en una importante tarea.

Se conoce que la Transforma Discreta de Fourier permite calcular el filtrado lineal en el dominio de la frecuencia y que la existencia de algoritmos rápidos para su cálculo, como son los algoritmos FFT descritos en el capítulo 2, la convierten en una eficiente herramienta para el cálculo del filtrado.

En los métodos OLS y OLA tradicionales descritos en el capítulo 3, la secuencia de entrada al proceso FFT es siempre real. Sin embargo, los algoritmos FFT son más eficientes cuando trabajan con señales complejas. Los métodos OLS y OLA modificados (Narasimha), expuestos en el capítulo 4, a diferencia de los métodos tradicionales, trabajan con versiones complejas de la secuencia de entrada, lo que permite una reducción del número total de operaciones aritméticas, a expensas de un incremento de la latencia, siendo el doble que la de los métodos tradicionales.

El método propuesto, descrito en el capítulo 4, también aplica los procesos FFT sobre versiones complejas de las secuencias reales pero, a diferencia del método Narasimha, no está limitado al filtrado en bloques y su latencia es la misma que la de los métodos tradicionales. En cuanto al número de operaciones necesarias, el algoritmo propuesto es el que requiere el menor número de sumas con un incremento poco significativo en el número de multiplicaciones.

Como se puede comprobar a partir de los resultados de las simulaciones del capítulo 5, el ahorro en el tiempo de ejecución de los algoritmos OLS no tradicionales (Narasimha y Propuesto), con respecto al método OLS Tradicional es mucho más significativo que lo que respecta al ahorro en complejidad computacional, siendo el método propuesto el que presenta los mejores resultados en cuanto a tiempo de ejecución.

Podemos concluir, por tanto, que de los métodos que se analizan en este proyecto, el método propuesto es el más eficiente para el cálculo del filtrado lineal de señales reales.

A continuación, se comentan algunas líneas de investigación futuras que podrían ser de interés en la búsqueda de conseguir mejorar los resultados obtenidos por los métodos propuestos en este documento:

- ✓ Además de los algoritmos FFT, existen otros métodos que permiten realizar un cálculo de la Transformada Discreta de Fourier de forma más eficiente que el cálculo directo. Teniendo en cuenta la importancia del cálculo de la DFT en el desarrollo de los métodos expuestos en este proyecto, resultaría interesante analizar alternativas a la FFT, realizando una comparativa en cuanto a requerimientos computacionales (operaciones aritméticas) y tiempo de ejecución.

- ✓ Como se ha demostrado, el método OLS modificado (Narasimha) proporciona mejores resultados, en cuanto a complejidad computacional y tiempo de ejecución, que el método OLS tradicional, en base a la eficiencia de aplicar los procesos FFT a secuencias complejas, en vez de reales. Sin embargo, es un aspecto también importante a tener en cuenta el hecho de que el método OLS incurre en operaciones redundantes ya que la FFT es aplicada sobre muestras solapadas (concatenación de bloques previos con bloques actuales). Analizar técnicas que permitan reducir esta redundancia resultaría de interés en la búsqueda de algoritmos OLS más eficientes.

Referencias

[1] James D. Broesch, Dag Stranneby, William Walker, Digital Signal Processing: Instant Access, 2009, 99.

[2] Goldstine, H. H. *A History of Numerical Analysis from the 16th Through the 19th Century,* New York, Heidelberg, Berlin, 1977, Springer-Verlag, 249-253.

[3] Cooley, J.W., y J.W. Tukey. "An Algorithm for the Machine Calculation of Complex Fourier Series", *Mathematics of Computation,* Vol.19, No.2, April, 1965, 297-301.

[4] Rudnick, P. "Note on the Calculation of Fourier Series", *Mathematics of Computation,* Vol. 20, No. 3, July, 1966, 429-430.

[5] Danielson, G.C., y C. Lanczos. "Some Improvements in Practical Fourier Analysis and their Application to X-ray Scatering from Liquids", *Journal of Franklin Institute,* Vol. 233, No. 4/5, April/May, 1942, 365-380, 435-452.

[6] John G.Proakis, Dimitris G.Manolakis, Tratamiento digital de señales, 3ªEdición, Capítulo 6, 457-490.

[7] Alan V.Oppenheim, Ronald W.Schafer, Tratamiento de señales en tiempo discreto, 3ªEdición, Capítulo 9, páginas: 705-728.

[8] John G.Proakis, Dimitris G.Manolakis, Tratamiento digital de señales, 3ªEdición, Capítulo 5, 437-441.

[9] F.J Simois, J.I.Acha, Efficient real data filtering using complex-input discrete Fourier transforms, Signal Processing 92 (2012), 1132-1136.

[10] Madihally J. Narasimha, IEEE Signal processing letters, VOL.13, NO. 11 (November 2008), Modified Overlap-Add and Overlap-Save Convolution Algorithms for Real Signals, 669-671.

[11] http://www.fftw.org/, FFTW version 3.3.4.

Anexo

Códigos Fuente

A.1 Algoritmo OLS Tradicional: Programación en C++

A continuación se detalla el código fuente, programado en C++, para el cálculo del algoritmo OLS Tradicional:

```
#include <windows.h>
#include <iostream>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <math.h>
#include <unistd.h>

#include <fftw3.h>
using namespace std;

// Prototipos de funciones.
double gaussrng(void);
double etime(LARGE_INTEGER *a, LARGE_INTEGER *b);
double OLS(int length_h, int length_x, int nreal);

int main(void)
    {
        //// Establecimiento de los parámetros generales. ////

        int length_h = 65536;  // Longitud vector h.
        int length_x = 524288; // Longitud vector x.

        int nreal = 100;        // Número de realizaciones del algoritmo.

        double time_OLS[nreal]; // Declaración de vector para almacenar los resultados finales.
```

```
//// Estimación del tiempo de procesamiento////
        int realizacion = 0;                    // Contador número de realización.
        for (int realizacion = 0; realizacion < nreal; realizacion++) {
                // Calculamos el promedio.
                time_OLS[realizacion]= OLS(length_h,length_x,nreal);
                cout << "Tiempo Ejecucion Promedio OLS:  " << time_OLS[realizacion] << endl;}
         getchar();
        return 0;
    }

double etime(LARGE_INTEGER *a, LARGE_INTEGER *b)
{
  LARGE_INTEGER freq;
  QueryPerformanceFrequency(&freq);   // Frecuencia del contador interno de Windows.
  double time = (double)(a->QuadPart - b->QuadPart) / (double)freq.QuadPart;
  return time;                        // Tiempo transcurrido entre a y b.
}

double OLS(int length_h, int length_x, int nreal)
{
        //// Establecimiento de los parámetros generales. ////
        int length_c = length_h + length_x - 1;     // Longitud de la convolución.
        int latency = length_h;         // La latencia se va a tomar igual a la longitud del filtro.

        int N = length_h + latency;     // Longitud de la DFT en cada iteración del algoritmo.
        int niter = (int) ceil((double) length_c/ (double) latency);   // Número de iteraciones.

        int length_y = niter*latency + latency;    // Longitud total que ha de tener y (rellenando con ceros).

        fftw_complex *ph, *px, *py, *pH, *pX, *pY;     // Punteros a bloques complejos.
        double *pin, *pout,;     // Punteros a bloques reales.
        fftw_plan planh, planx, plany;  // Variables para crear los planes.
```

```
LARGE_INTEGER t_ini_0, t_fin_0; // Variables para almacenar el tiempo transcurrido.
double secs = 0;

//// Generación de los vectores a convolucionar. ////
srand(time(NULL));              // Se inicia aleatoriamente la semilla.

// Reserva de memoria
pin = (double*) fftw_malloc(sizeof(double)*length_y);  // Puntero a la secuencia de entrada.
ph = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*N); // Puntero a h.

px = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*N); // Puntero a cada bloque de x.
pH = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*(N)); // Puntero a la DFT de h.
pX = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*(N)); // Puntero a DFT de bloque de x.
pY = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*N); // Puntero a cada bloque de Y.
py = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*N); // Puntero a resultado convolución.
pout = (double*) fftw_malloc(sizeof(double)*length_y);  // Puntero al resultado final.

// Creación de los planes.
planh = fftw_plan_dft_1d(N, ph, pH, FFTW_FORWARD, FFTW_ESTIMATE);    // Plan h.
planx = fftw_plan_dft_1d(N, px, pX, FFTW_FORWARD, FFTW_ESTIMATE);    // Plan x.
plany = fftw_plan_dft_1d(N, pY, py, FFTW_BACKWARD, FFTW_ESTIMATE);    // Plan y.

for (int real = 0; real < nreal; real++)              // Bucle para las realizaciones.
{
   // Se crea el vector h y se calcula su DFT.
   for (int ind=0; ind < length_h; ind++) {
      ph[ind][0] = rand()%10;
      ph[ind][0] = ph[ind][0];
      ph[ind][1] = 0.0;}
   for (int ind=length_h; ind < N; ind++)  {
      ph[ind][0] = 0.0;
      ph[ind][1] = 0.0;}
```

```
// Se calcula la DFT de h.
fftw_execute(planh);

// Se crea el vector x.
for (int ind=0; ind < length_h; ind++) {pin[ind] = 0.0;}
for (int ind=length_h; ind < length_x+length_h; ind++) {pin[ind] = rand()%10;}
for (int ind=length_x+length_h; ind < length_y; ind++) {pin[ind] = 0.0;}

//// Ejecución del algoritmo. ////
QueryPerformanceCounter(&t_ini_0);      // Se inicia el contador de Windows.

for (int iter=0; iter < niter; iter++)  // Bucle para las iteraciones.
{
   int s_init = iter*latency;          // Posición de la muestra inicial del bloque.
   for (int ind=0; ind < N; ind++) {
      px[ind][0] = pin[s_init+ind];

      px[ind][1] = 0.0;}  // Selección del bloque.

   // Se calcula la DFT de px.
   fftw_execute(planx);

   // Se multiplican pH y pX.
   for (int ind = 0; ind < N; ind++)
   {
      pY[ind][0] = (pH[ind][0])*(pX[ind][0]) - (pH[ind][1])*(pX[ind][1]);
      pY[ind][1] = (pH[ind][0])*(pX[ind][1]) + (pH[ind][1])*(pX[ind][0]);}

    // Se calcula la IDFT de Y.
   fftw_execute(plany);           // Se calcula la IDFT de Y.
```

```
            // Se copia el resultado en el vector de salida.
             for (int ind=0; ind < latency; ind++) {pout[s_init+ind] = py[latency+ind][0]/N;}
        }                              // Fin del bucle de las iteraciones.

        QueryPerformanceCounter(&t_fin_0);  // Paramos el contador de Windows.
        secs += etime(&t_fin_0, &t_ini_0);  // Evaluamos el tiempo transcurrido.
        cout << "Iteracion: " << real+1 << ". Tiempo: " << etime(&t_fin_0, &t_ini_0) << endl;

    }                                  // Fin del bucle de las realizaciones.

    secs /= nreal;                     // Dividimos entre el número de realizaciones.

    //// Se libera la memoria reservada. ////
    fftw_destroy_plan(planh); fftw_destroy_plan(planx); fftw_destroy_plan(plany);
    fftw_free(pin); fftw_free(pout);
    fftw_free(ph); fftw_free(px); fftw_free(py);
    fftw_free(pH); fftw_free(pX); fftw_free(pY);

    return secs;
}
```

A.2 Algoritmo Narasimha: Programación en C++

A continuación se detalla el código fuente, programado en C++, para el cálculo del algoritmo OLS Modificado (Narasimha):

```
#include <windows.h>
#include <iostream>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <math.h>
#include <unistd.h>
```

```
#include <fftw3.h>
using namespace std;

// Prototipos de funciones.
double gaussrng(void);
double etime(LARGE_INTEGER *a, LARGE_INTEGER *b);
double OLS(int,int,int);

int main(void)
    {
       //// Establecimiento de los parámetros generales. ////
       int length_h = 4;  // Longitud vector h.
       int length_x = 16; // Longitud vector x.

       int nreal = 100;        // Número de realizaciones del algoritmo.

       double time_OLS[nreal];    // Declaración de vector para almacenar los resultados finales.

       //// Estimación del tiempo de procesamiento////
       int realizacion = 0;                  // Contador número de realización.
       for (int realizacion = 0; realizacion < nreal; realizacion++) {
            // Calculamos el promedio.
            time_OLS[realizacion]= OLS(length_h, length_x, nreal);
            cout << "Tiempo Ejecucion Promedio OLS:  " << time_OLS[realizacion] << endl;}
       getchar();
       return 0;
       }

double etime(LARGE_INTEGER *a, LARGE_INTEGER *b)
{
 LARGE_INTEGER freq;
 QueryPerformanceFrequency(&freq);   // Frecuencia del contador interno de Windows.
```

```
  double time = (double)(a->QuadPart - b->QuadPart) / (double)freq.QuadPart;
  return time;                    // Tiempo transcurrido entre a y b.
}

double OLS(int length_h, int length_x, int nreal)
{
        //// Establecimiento de los parámetros generales. ////
        int length_c = length_h + length_x - 1;     // Longitud de la convolución.
        bool fprint = 0;                // Flag para representar resultados por pantalla.

        int N = 2*length_h;     // Longitud de la DFT en cada iteración del algoritmo. La latencia se va a
                                    tomar igual a 2 veces la longitud del filtro.
        int niter = (int) ceil((double) length_c/ (double)(2*N));   // Número de iteraciones.
        int length_y = N*niter + length_h;      // Longitud total que ha de tener y (rellenando con ceros).

        double *pin, *pout;   //Punteros a los bloques reales.
        fftw_complex *ph, *px,*pH, *pX,*pP,*pp;     // Punteros a los bloques complejos.

        fftw_plan planx, planh, planp;  // Variables para crear los planes.

        LARGE_INTEGER t_ini_0, t_fin_0; // Variables para almacenar el tiempo transcurrido.
        double secs = 0;

        //// Generación de los vectores a convolucionar. ////
        srand(time(NULL));              // Se inicia aleatoriamente la semilla.

        // Reserva de memoria
        pin = (double*) fftw_malloc(sizeof(fftw_complex)*length_y);  // Puntero a la secuencia de entrada
        ph = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*N); // Puntero a h.
        px = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*N); // Puntero a cada bloque de x.
        pH = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*(N)); // Puntero a la DFT de h.
        pX = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*(N)); // Puntero a DFT de bloque de x.
        pP = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*N); // Puntero a cada bloque de P.
```

```
pp = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*N); // Puntero al resultado convolución.
pout = (double*) fftw_malloc(sizeof(double)*length_y); // Puntero a cada bloque de y.

// Creación de los planes.
planh = fftw_plan_dft_1d(N, ph, pH, FFTW_FORWARD, FFTW_ESTIMATE);  // Plan para h.
planx = fftw_plan_dft_1d(N, px, pX, FFTW_FORWARD, FFTW_ESTIMATE);   // Plan para x.
planp = fftw_plan_dft_1d(N, pP, pp, FFTW_BACKWARD, FFTW_ESTIMATE); //Plan para p.

for (int real = 0; real < nreal; real++)              // Bucle para las realizaciones.
{
 // Se crea el vector h y se calcula su DFT.
   for (int ind=0; ind < length_h; ind++) {
      ph[ind][0] = rand()%10;
      ph[ind][0] = ph[ind][0];
      ph[ind][1] = 0.0;}

   for (int ind=length_h; ind < N; ind++)  {
      ph[ind][0] = 0.0;
      ph[ind][1] = 0.0;}

   // Se calcula la DFT de h.
   fftw_execute(planh);

   // Se crea el vector x.
   for (int ind=0; ind < length_h; ind++) {pin[ind] = 0.0;}
   for (int ind=length_h; ind < length_x+length_h; ind++) {pin[ind] = rand()%10;}
   for (int ind=length_x+length_h; ind < length_y; ind++) {pin[ind] = 0.0;}

   //// Ejecución del algoritmo. ////
   QueryPerformanceCounter(&t_ini_0);      // Se inicia el contador de Windows.
```

```
for (int iter=0; iter < niter; iter++)  // Bucle para las iteraciones.
{
   int s_init = iter*2*length_h;         // Posición de la muestra inicial del bloque.

   for (int ind=0; ind < (N/2); ind++) {
   px[ind][0] = pin[(iter+1)*N+ind];
   px[ind+(N/2)][0]=pin[iter*N+(N/2)+ind];
   px[ind][1]=pin[iter*N+(N/2)+ind];
   px[ind+(N/2)][1]=pin[iter*N+ind];}  // Selección del bloque.

   // Se calcula la DFT de px.
   fftw_execute(planx);

   // Se multiplican pH y pX.
   for (int ind = 0; ind < N; ind++)
   {
      pP[ind][0] = (pH[ind][0])*(pX[ind][0]) - (pH[ind][1])*(pX[ind][1]);
      pP[ind][1] = (pH[ind][0])*(pX[ind][1]) + (pH[ind][1])*(pX[ind][0]);}

   // Se calcula la IDFT de P.
   fftw_execute(planp);           // Se calcula la IDFT de P.

   // Se copia el resultado en el vector pout.
   for (int ind=0; ind < length_h; ind++){
      pout[s_init+ind] = pp[ind][1]/N;}
   for (int ind=0; ind < length_h; ind++){
      pout[s_init+ind+length_h]= pp[ind][0]/N;}
}                                 // Fin del bucle de las iteraciones.

QueryPerformanceCounter(&t_fin_0);  // Paramos el contador de Windows.
secs += etime(&t_fin_0, &t_ini_0);  // Evaluamos el tiempo transcurrido.
cout << "Iteracion: " << real+1 << ". Tiempo: " << etime(&t_fin_0, &t_ini_0) << endl;
```

```
    }                                   // Fin del bucle de las realizaciones.

    secs /= nreal;                      // Dividimos entre el número de realizaciones.

    //// Se libera la memoria reservada. ////
    fftw_destroy_plan(planh); fftw_destroy_plan(planx); fftw_destroy_plan(planp);
    fftw_free(pin); fftw_free(pout); fftw_free(ph); fftw_free(px); fftw_free(pp);
    fftw_free(pH); fftw_free(pX); fftw_free(pP);
    return secs;
}
```

A.3 Algoritmo Propuesto: Programación en C++

A continuación se detalla el código fuente, programado en C++, para el cálculo del algoritmo Propuesto:

```
#include <windows.h>
#include <iostream>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <math.h>
#include <unistd.h>
#include <complex.h>
#include <fftw3.h>
using namespace std;

// Prototipos de funciones.
double gaussrng(void);
double etime(LARGE_INTEGER *a, LARGE_INTEGER *b);
double OLS_Propuesto(int,int,int);
```

```
int main(void)
    {
        //// Establecimiento de los parámetros generales. ////

        int length_h = 4;  // Longitud vector h.
        int length_x = 16; // Longitud vector x.

        int nreal = 100;        // Número de realizaciones del algoritmo.

        double time_OLS_Propuesto[nreal]; // Declaración de vector para almacenar  resultados finales.


        //// Estimación del tiempo de procesamiento////
        int realizacion = 0;                // Contador número de realización.

        for (int realizacion = 0; realizacion < nreal; realizacion++) {
              // Calculamos el promedio.
              time_OLS_Propuesto[realizacion]= OLS_Propuesto(length_h,length_x,nreal);
              cout << "Tiempo Ejecucion Promedio OLS:  "  << time_OLS_Propuesto[realizacion] <<
endl;
        }
         getchar();
        return 0;
    }

double etime(LARGE_INTEGER *a, LARGE_INTEGER *b)
{
  LARGE_INTEGER freq;
  QueryPerformanceFrequency(&freq);   // Frecuencia del contador interno de Windows.
  double time = (double)(a->QuadPart - b->QuadPart) / (double)freq.QuadPart;
  return time;                    // Tiempo transcurrido entre a y b.
}
```

```
double OLS_Propuesto(int length_h, int length_x, int nreal)
{
        //// Establecimiento de los parámetros generales. ////
       double pi=3.14;
        double a;
        double b;

       int length_y = length_h+length_x;     // Longitud de la convolución +1.
         int M = length_y/2;  // Longitud de las DFTs.

        double *px, *ph, *py; //Punteros a la entrada, filtro y salida.



        // Punteros a los bloques complejos.
       fftw_complex *phc, *pxc, *pyc, *phe, *pho, *pHe, *pHo, *pHI, *pHII, *pXc, *pXc_cir, *pYc;
       fftw_plan planhe, planho, planxc, planyc;  // Variables para crear los planes.

       LARGE_INTEGER t_ini_0, t_fin_0; // Variables para almacenar el tiempo transcurrido.
       double secs = 0;

       //// Generación de los vectores a convolucionar. ////
       srand(time(NULL));              // Se inicia aleatoriamente la semilla.

       // Reserva de memoria
       px = (double*) fftw_malloc(sizeof(double)*(length_y));
       ph = (double*) fftw_malloc(sizeof(double)*(length_y)); // Puntero a h.
        phe = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*M);  // Puntero a he.
        pho = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*M);  // Puntero a ho.
       pxc = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*M);   // Puntero a xc.
        pXc = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*M);  // Puntero a la DFT de xc.
        pXc_cir = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*M);  // Puntero a ….
        pHe = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*M);  // Puntero a la DFT de he.
```

```
pHo = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*M);  // Puntero a la DFT de ho.
pHI = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*M);   // Puntero a la DFT de HI.
pHII = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*M); // Puntero a la DFT de HII.
 pyc = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*M);  // Puntero a yc.
 pYc = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*M);  // Puntero a la DFT de yc.
py = (double*) fftw_malloc(sizeof(double)*(length_y)); // Puntero al resultado final.

// Creación de los planes.
planhe = fftw_plan_dft_1d(M, phe , pHe, FFTW_FORWARD, FFTW_MEASURE);  //Plan para
hc.
planho = fftw_plan_dft_1d(M, pho , pHo, FFTW_FORWARD, FFTW_MEASURE); //Plan para
ho.
planxc = fftw_plan_dft_1d(M, pxc , pXc, FFTW_FORWARD, FFTW_MEASURE); //Plan para
xc.

planyc = fftw_plan_dft_1d(M, pYc, pyc, FFTW_BACKWARD, FFTW_MEASURE); //Plan para
yc.

for (int real = 0; real < nreal; real++)              // Bucle para las realizaciones.
{
   // Se crea el vector h.
   for (int ind=0; ind < length_h; ind++) {
      ph[ind] = rand()%10;}
   for (int ind=length_h; ind < length_y; ind++)  {ph[ind] = 0.0;}

    // Se generan los vectores he y ho.
    for (int ind=0; ind<M; ind++) {
            pho[ind][0]=ph[(2*ind)+1];
            pho[ind][1]=0.0;
            phe[ind][0]=ph[(2*ind)];
            phe[ind][1]=0.0;}

// Se calcula la DFT de he.
   fftw_execute(planhe);
```

```
// Se calcula la DFT de ho.
   fftw_execute(planho);

// Se calculan H_I y H_II.
     for(int ind=0; ind<M; ind++){
          a=cos(2*pi*ind/(M));
          b=sin(2*pi*ind/(M));
          pHI[ind][0]=pHe[ind][0]-0.5*pHo[ind][0]*b+0.5*pHo[ind][1]*(a-1.0);
          pHI[ind][1]=pHe[ind][1]-0.5*pHo[ind][1]*b+0.5*pHo[ind][0]*(1.0-a);
          pHII[ind][0]=0.5*pHo[ind][0]*b-0.5*pHo[ind][1]*(a+1.0);
          pHII[ind][1]=0.5*pHo[ind][1]*b+0.5*pHo[ind][0]*(a+1.0);}

// Se crea el vector x.
for (int ind=0; ind < length_x; ind++) {px[ind] = rand()%10;}
for (int ind=length_x; ind < length_y; ind++) {px[ind] = 0.0;}

// Se genera el vector xc.
 for (int ind=0; ind<M; ind++) {
          pxc[ind][0]=px[2*ind];
          pxc[ind][1]=px[2*ind+1];}

QueryPerformanceCounter(&t_ini_0);      // Se inicia el contador de Windows.
// Se calcula la DFT de xc.
fftw_execute(planxc);

 //// Ejecución del algoritmo. ///

    // Se calcula Xc*((-k))M/2
  pXc_cir[0][0] = pXc[0][0];
  pXc_cir[0][1] = -pXc[0][1];
```

```
  for(int ind=1; ind<M; ind++){
          pXc_cir[ind][0]=pXc[M-ind][0];
          pXc_cir[ind][1]=-pXc[M-ind][1];}

// Se calcula Yc.
for (int ind = 0; ind < M; ind++){
          pYc[ind][0]=(pXc[ind][0])*(pHI[ind][0])+(pXc_cir[ind][0])*(pHII[ind][0])-
          (pXc[ind][1])*(pHI[ind][1])-(pXc_cir[ind][1])*(pHII[ind][1]);

          pYc[ind][1]=(pXc[ind][0])*(pHI[ind][1])+(pXc_cir[ind][0])*(pHII[ind][1])+(pXc[ind][1])*
          (pHI[ind][0])+(pXc_cir[ind][1])*(pHII[ind][0]);}

// Se calcula la IDFT de Yc.
fftw_execute(planyc);          // Se calcula la IDFT de Yc.

// Se genera y a partir de yc y se almacena el resultado en el vector de salida.
for (int ind=0; ind<M; ind++) {

       py[2*ind] = pyc[ind][1]/M;
       py[ind+1] = pyc[ind][0]/M;}

QueryPerformanceCounter(&t_fin_0);  // Paramos el contador de Windows.
secs += etime(&t_fin_0, &t_ini_0);  // Evaluamos el tiempo transcurrido.
cout << "Iteracion: " << real+1 << ". Tiempo: " << etime(&t_fin_0, &t_ini_0) << endl;

}                                   // Fin del bucle de las realizaciones.

secs /= nreal;                      // Dividimos entre el número de realizaciones.

//// Se libera la memoria reservada. ////
fftw_destroy_plan(planhe); fftw_destroy_plan(planho);
 fftw_destroy_plan(planxc); fftw_destroy_plan(planyc);
fftw_free(ph);
```

```
        fftw_free(px);
        fftw_free(phe); fftw_free(pho);
        fftw_free(pxc); fftw_free(pyc); fftw_free(py);
        fftw_free(pXc); fftw_free(pXc_cir);
        fftw_free(pHe); fftw_free(pHo);
        fftw_free(pHI); fftw_free(pHII);
        fftw_free(pYc);

      return secs;
}
```