UNIVERSIDAD DE MURCIA

FACULTAD DE MATEMÁTICAS

Grado en Matemáticas
Curso 2017-2018

Trabajo fin de Grado

Transformada rápida de Fourier

Implementación y algunas aplicaciones

Ana Martínez Manzano

Tutorizada por:
José María Almira (Departamento de Matemática Aplicada)
y Antonio José Pallarés Ruiz (Departamento de Matemáticas)

Declaración de originalidad

ANA MARTÍNEZ MANZANO, autora del TFG "Transformada rápida de Fourier: Implementación y algunas aplicaciones", bajo la tutela de los profesores, ANTONIO JOSÉ PALLARÉS RUIZ Y JOSÉ MARÍA ALMIRA, declara que el trabajo que presenta es original, en el sentido de que ha puesto el mayor empeño en citar debidamente todas las fuentes utilizadas.

En Murcia, a 26 de Junio de 2018.

(Nota: en la Secretaría de la Facultad de Matemáticas se ha presentado una copia firmada de esta declaración).

Resumen

El objetivo principal de este trabajo es el estudio de la **transformada de Fourier rápida** (FFT), una potente herramienta de análisis matemático que se utiliza en diversos campos de la ciencia. Concretamente, estudiamos algunos de los algoritmos con los que puede ser implementada e implementamos dos aplicaciones que son especialmente importantes: la multiplicación de números grandes y el cálculo aproximado de frecuencias de señales analógicas unidimensionales.

En 1805 Gauss[1] creó el primer algoritmo de "divide y vencerás", un algoritmo que se utilizó para el cálculo de la transformada de Fourier discreta de forma recursiva.

Fué un siglo y medio después, en 1965, cuando dos científicos Norteamericanos, J.W.Cooley y J.W. Tukey[2], redescubrieron el algoritmo más eficiente hasta entonces conocido para realizar dicho cálculo, al cual le dieron el nombre de transformada de Fourier rápida. En la actualidad se considera que este algoritmo es uno de los grandes descubrimientos más importantes de la segunda mitad del siglo XX, y su estudio ha dado lugar a miles de artículos de investigación y se utiliza permanentemente por casi todo dispositivo tecnológico que maneja cualquier tipo de información, tanto para transmitirla como para manipularla.

Como en este trabajo estamos interesados en la transformada de Fourier rápida, al ser esta un algoritmo eficaz para el cálculo de la transformada de Fourier discreta, es necesario conocer antes la definición de esta última y sus propiedades más importantes. Por ello, en el **capítulo 1**, introducimos la transformada de Fourier discreta (DFT), que no es otra cosa que un cambio de coordenadas en $\mathbb{C}^N$ entre dos bases con un significado especial. Se define formalmente la DFT como la transformación $\mathfrak{F}_{\mathfrak{N}} : \mathbb{C}^N \longrightarrow \mathbb{C}^N$ que a cada vector $f \in \mathbb{C}^N$ le hace corresponder el vector $\widehat{f} = (\widehat{f}(0), \widehat{f}(1), ..., \widehat{f}(N-1))$ donde

$$\widehat{f}(k) = \sum_{n=0}^{N-1} f(n) e^{\frac{-2i\pi kn}{N}}$$

para $0 \leqslant n \leqslant N-1$. Junto con algunas de sus propiedades, introducimos el concepto de convolución, cuya importancia se basa en que esta caracteriza a los sitemas LTI "Linear Translation Invariant", que en el contexto que tiene este trabajo, seran operadores lineales e invariantes frente a translaciones en el tiempo. Los sistemas LTI reciben también el nombre de "filtros de ondas" y son muy útiles en todo tipo de aplicaciones.

En el **capítulo 2**, definimos la transformada de Fourier rápida (FFT) como cualquier algoritmo para el cálculo de la transformada de Fourier discreta (DFT) que reduzca su complejidad del orden de N^2 operaciones al orden de $N\log(N)$ operaciones, donde N es el tamaño, la dimensión, del vector de entrada. Esto por supuesto implica una reducción significativa en el coste computacional de la DFT, el cual es del orden de N^2 operaciones si se cálcula con un algoritmo ingenuo. Además,

[1] C.F. Gauss "Theoria Interpolationis Methodo Nova Tractata" en Königlichen Gesellschaft der Nissenschaften, Band 2, pag 265-330, Göttingen 1805

[2] J.W. Cooley y J.W. Tukey "An algorithm for machine calculation of complex Fourier transform" en Math. Computation, Vol 19, pag 297-301, 1965

este es hoy un problema abierto que tiene visos de ser extremadamente difícil y se considera uno de los principales problemas de la teoría de algoritmos.

Los algoritmos básicos FFT, son los llamados algoritmos FFT de radio 2, RADIX2, que son algoritmos recursivos para el cálculo de la DFT que se obtienen dividiendo en cada paso del proceso recursivo el problema en dos subproblemas de tamaño la mitad del tamaño del problema original.

Estos algoritmos tienen la restricción de que el vector de entrada tiene que ser de tamaño una potencia de 2 ($N = 2^n$ para cierto $n \in \mathbb{N}$).

Estudiamos dos algoritmos basados en el paradigma de "divide y vencerás". En primer lugar, el algoritmo DIT FFT (decimación en tiempo), que se basa en dividir la suma que define la transformada de Fourier discreta en dos sumas, donde por un lado aparecen los términos pares y por otro los impares. En este algoritmo se aplica la denominada **operación mariposa de Cooley-Tukey**, que nos dice como movernos en los distintos subproblemas que se definen en el algoritmo.

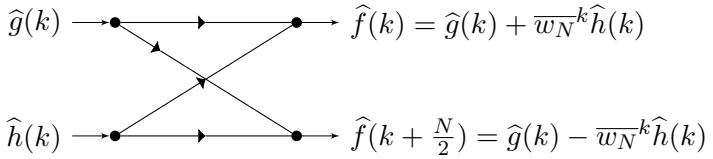


Figura 1: Véase en la sección 2.1, página 11

En segundo lugar, consideramos el algoritmo DIF FFT (decimación en frecuencias). Este algoritmo se basa de nuevo en dividir la suma que define la transformada de Fourier discreta en dos sumas, donde en una de ellas aparecen la primera mitad de los términos y en la segunda el resto. En este algoritmo se aplica la denominada **operación mariposa de Gentleman-Sande**.

El coste computacional de los algoritmos es común expresarlo en términos de operaciones en punto flotante o flops, es decir, en términos del número de operaciones de números reales que se realizan. Se demuestra que el coste computacional de los dos algoritmos que acabamos de describir es $5(\log_2(N))N$.

Después de esto pasamos a la implementación de algunos de los algoritmos. Escribimos el código en Java, utilizando el entorno de desarrollo NetBeans.

En el **capítulo 3**, vemos cómo implementar el algoritmo de la transformada de Fourier rápida con decimación en frecuencia (DIF FFT). Para ello, en un principio consideramos que los datos de entrada (las coordenadas del vector f) están dispuestos en su orden natural

$$(f(0), f(1), \ldots, f(N-1)).$$

El algoritmo consiste en, dado un vector inicial de tamaño $N = 2^m$, ir descomponiendo el problema del cálculo de su DFT al aplicar la correspondiente operación mariposa de forma recursiva, en subproblemas de cálculo de DFT de vectores de la mitad de los tamaños anteriores.

Para evitar la complejidad estática que puede producir un algoritmo recursivo en relación con el uso de memoria, en la implementación del algoritmo se utiliza un único vector en el que se van realizando todas las operaciones con el minimo movimiento posible de posiciones en el vector. Por esta razón, al realizar el algoritmo se obtiene un vector con todas las coordenadas de la DFT pero con una ordenación especial (bit-reversal).

Para entender esta ordenación, repasamos la **representación binaria** de un entero $0 \leq L < N$, $L = (i_{n-1}, i_{n-2}, \ldots, i_1, i_0)_2$, con $i_k \in \{0, 1\}$ $(k = 0, 1, \ldots, n-1)$, donde

$$L = i_{n-1} \cdot 2^{n-1} + \ldots + i_1 \cdot 2^1 + i_0 \cdot 2^0$$

La permutación bit-reversal, está definida por $\varphi(L) = (i_0, i_1, ..., i_{n-2}, i_{n-1})_2$. La salida del algoritmo es un vector con todas las coordenadas de la DFT en orden bit-reversal.

Debido a que la salida del algoritmo depende del orden de las coordenadas del vector de entrada, estudiamos además dos variaciones del algoritmo. Por un lado, el denominado $DIF_{RN}FFT$, donde las iniciales RN hacen referencia a que la entrada del algoritmo es en orden bit-reversal (R) y la salida en orden natural (N). Este algoritmo se basa en descomponer el problema inicial de tamaño N, en subproblemas de tamaño 2 e ir doblando el tamaño de los subproblemas con respecto al tamaño del subproblema anterior, donde en cada uno de los subproblemas aplicamos la correspondiente operación mariposa de Gentleman-Sande. Por otro lado, denominado $DIF_{NN}FFT$. En este algoritmo tanto la entrada como la salida son en orden natural, por lo que recibe el nombre de algoritmo ordenado. Este algoritmo se basa en ver cada operación mariposa como un paso de permutación seguido del cálculo de la operación mariposa, donde los pasos de permutación ordenan tanto la entrada como la salida en cada subproblema.

Con el objetivo de reducir el coste computacional de la FFT, en el **capítulo 4**, estudiamos la transformada de Fourier rápida de radio 4, RADIX4, para ello debemos considerar vectores de tamaño $N = 4^n = 2^{2n}$.

De forma análoga a las técnicas usadas en los algoritmos $DITFFT$ de radio 2 y $DIFFFT$ de radio 2, desarrollamos los algoritmos $DITFFT$ de radio 4 y $DIFFFT$ de radio 4, con la diferencia de que en este último caso dividimos la suma que define la transformada de Fourier discreta en cuatro sumas. Y teniendo en cuenta algunas consideraciones especiales a la hora de realizar los cálculos en las correspondientes operaciones mariposa, vemos que el coste computacional de este algoritmo es $\frac{17N\log_2 N}{4} - \frac{43N}{6} + \frac{32}{3}$.

Si comparamos el coste computacional del algoritmo FFT de radio 2 con este último tenemos:

$$\lim_{N\to\infty} \frac{S(N)}{T(N)} = \frac{\frac{17}{4}}{5} = \frac{17}{20},$$

de lo que se sigue que hemos conseguido ahorrar un 15 %, ya que el algotirmo de radio 4 tiene como coste un 85 % del coste del algoritmo de radio 2.

Una vez estudiados los algoritmos FFT de radio 2 y FFT de radio 4, resulta natural describir los algoritmos FFT de radio split, SPLIT RADIX, que consiste en dividir la suma que define la transformada de Fourier discreta en tres sumas, bien usando decimación en tiempo o decimación en frecuencia. Este método se aplica a vectores de longitud $N = 2^n$ y reduce el coste computacional a $4N\log_2 N - 6N + 8$, un 80 % del coste del algoritmo de radio 2.

En el **capítulo 5**, mostramos dos aplicaciones de la transformada de Fourier rápida. La primera de ellas es la **multiplicación de números grandes**, para la que hemos usado los algoritmos que hemos implementado en Java y las herramientas introducidas en los capítulos 2 y 3.

Multiplicar dos números naturales tiene un coste del $O(n^2)$, donde n representa el número de dígitos que tiene cada factor. En esta sección vemos cómo con el uso de los algoritmos implementados podemos conseguir que la multiplicación de dichos números tenga un coste de $O(n\log(n))$.

Dado un número natural A, si fijamos una base, b, este se puede escribir de la forma

$$A = a_0 b^0 + a_1 b^1 + ... + a_{n-1} b^{n-1},$$

para ciertos dígitos $a_i \in \{0, 1, 2, \ldots, b-1\}$. Esto tiene el efecto de que la multiplicación de números grandes está íntimamente ligada a la multiplicación de polinomios, problema cuyo coste computacional se reduce significativamente con el uso de la FFT y que también resolvemos aquí.

Multiplicar dos polinomios $p(x) = a_0 + a_1 x^1 + ... + a_{n-1} x^{n-1}$ y $q(x) = b_0 + b_1 x^1 + ... + b_{n-1} x^{n-1}$ da como resultado un tercer polinomio

$$p(x) \cdot q(x) = a_0 b_0 + (a_0 b_1 + a_1 b_0)x + ... + (a_{n-1} b_{n-1}) x^{2n-2}.$$

Esta operación tiene un coste aritmético $O(n^2)$ al igual que el de multiplicar dos números enteros. Para disminuir este coste vamos a introducir otra forma de representar un polinomio distinta de la usual.

Consideramos una familia arbitraria $\{(x_i, y_i)\}_{i=0}^{n-1}$ de n puntos con las abcisas (x_i) distintas dos a dos, el teorema de Interpolación de Lagrange nos asegura que existe un único polinomio algebráico $p(x)$ de grado al menos $n-1$ tal que $p(x_i) = y_i$ para $i = 0, 1, ..., n-1$. Este polinomio recibe el nombre de "polinomio interpolador" (o "polinomio interpolador de Lagrange") de la familia de puntos $\{(x_i, y_i)\}_{i=0}^{n-1}$, y el resultado que nos afirma que este polinomio de grado $n-1$ se define de forma única con su evaluación en el conjunto de n puntos distintos igualmente espaciados $\{x_i\}_{i=0}^{n-1}$. Por otra parte, el resultado también es cierto para cantidades complejas x_i y polinomios complejos $p(z) = a_0 + a_1 z + ... + a_{n-1} z^{n-1} \in \mathbb{C}[z]$.

Este teorema nos permite asegurar que si $p(x)$ y $q(x)$ son polinomios de grado $n-1$ y por tanto su producto de grado menor o igual que $2n-1$, este último se define de forma única a partir de su evalación en $2n$ puntos distintos.

Llegados a este punto la clave es que la evaluación de un polinomio $p(x)$ en las (n)-ésimas raíces de la unidad $e^{\frac{-2\pi ik}{n}}$ produce la transformada de Fourier del vector formado por sus propios coeficientes.

$$p\left(e^{\frac{-2\pi ik}{n}}\right) = \sum_{j=0}^{n-1} a_j e^{\frac{-2\pi ikj}{n}}.$$

Por tanto, podemos usar algoritmos FFT para calcular estas evaluaciones (las cuales reciben el nombre DFT del polinomio $p(x)$) lo cual reduce su coste computacional. Como los algoritmos FFT necesitan un vector de entreda con tamaño una potencia de dos y los coeficientes de cada uno de los polinomios son vectores de tamaño n, estos deben ser completados con ceros hasta la posición una pontencia de 2 que debe ser $\geq 2n$, es decir, si el vector de coeficientes de p es $a = (a_0, a_1, ..., a_{n-1})$ y el correspondiente vector de coeficientes de q es $b = (b_0, b_1, ..., b_{n-1})$. Completamos con ceros de forma que $a' = (a_0, a_1, ..., a_{n-1}, 0, ..., 0)$ y $b' = (b_0, b_1, ..., b_{n-1}, 0, ..., 0)$, que son vectores de tamaño 2^k, donde k es el entero más pequeño tal que $2^k \geq 2n$. El producto de los dos polinomios se puede interpretar con la convolución de estos dos vectores, y su DFT es el producto puntual de las transformadas FFT de los dos vectores. Por tanto, debemos multiplicar las salidas de la FFT, y aplicarle la FFT inversa para conseguir el polinomio producto que representa al producto de los números.

Con todo esto llegamos a la conclusión que para conseguir disminuir el coste de la multiplicación de números grandes, usando algoritmos FFT debemos seguir los siguientes pasos:

- Expresar cada uno de los números a multiplicar en base 10 y colocar los coeficientes de dicho polinomio en un vector.
- Completar, añadiendo ceros, el vector de coeficientes de cada uno de los números hasta la primera potencia de dos mayor o igual que 2n.
- Aplicar la FFT a cada vector obtenido de este modo.
- Multiplicar, coordenada a coordenada, los dos vectores obtenidos en el apartado anterior.
- Aplicar la FFT inversa al vector que resulta de la multiplicación anterior.

La segunda aplicación que consideramos muestra de forma teórica como utilizar la transformada de Fourier rápida a un caso real. En la vida real existen numerosas situaciones en las que se hace necesario estudiar en frecuencias una señal analógica $x(t)$, que se define con el nombre de

transformada de Fourier (analógica), $\mathcal{F}(x)(\xi) = \int_{-\infty}^{\infty} x(t)e^{-2\pi it\xi}dt$ (esta función se suele denotar por $\widehat{x}(\xi)$ y, en las aplicaciones de ingenieria, por $X(\xi)$), sin embargo, típicamente no podemos abordar el cálculo directo de la transformada de Fourier de la señal $x(t)$, bien porque desconocemos la expresión analítica de la señal $x(t)$ o bien porque, aún conociendo dicha expresión, no es posible calcular una primitiva de la función $x(t)e^{-2\pi i\xi t}$. En cambio, en el mundo digital siempre podemos realizar esos cálculos, y se hace precisamente con la DFT, que afortunadamente se puede calcular de forma rápida mediante los algoritmos FFT. Ahora bien, afortunadamente, el paso de señales analógicas a señales digitales sin perdida de información puede ser alcanzada en todas las aplicaciones reales. Esto es gracias al siguiente resultado importante del análisis de Fourier:

Teorema del muestreo: Supongamos que $x(t) \in L^2(\mathbb{R})$ y $\widehat{x}(\xi) = 0$ para todo $|\xi| > b$. Entonces $x(t)$ queda completamente determinada a partir de sus valores en los puntos $\left\{\frac{k}{2b}\right\}_{k\in\mathbb{Z}}$. Además se satisface la fórmula:

$$x(t) = \sum_{k=-\infty}^{\infty} x(\frac{k}{2b})\frac{\sin(\pi(-k+2bt))}{\pi(-k+2bt)}$$

donde la serie converge en $L^2(\mathbb{R})$ y también absoluta y uniformemente en $\mathbb{R}$.

Una vez que estamos en el mundo digital, podemos aplicar la FFT a las muestras tomadas como nos indica el teorema del muestreo, obteniendo la descripción de la señal en términos de frecuencias en esas muestras. Por último, mediante técnicas de reescalado y reordenación podemos conseguir una buena aproximación de la descripción de la señal original en términos de frecuencias.

El teorema anterior nos permite hacer los cálculos con rigor para funciones en $L^2(\mathbb{R})$, con banda limitada. Lo cual excluye funciones tan importantes y habituales como pueden ser las funciones trigonométricas, por lo que incluimos una idea, en la que usamos distribuciones temperadas, para poder aplicar este teorema a dichas funciones.

Además incluimos un programa en Matlab, que es una herramienta de software matemático que ofrece un entorno de desarrollo integrado con un lenguaje de programación propio que incluye entre sus atributos gráficas integradas que facilitan la visualización de los datos. Al pasar al programa una fución analógica, junto con su transformada de Fourier, el periodo de muestreo deseado y el intervalo donde queremos que se tomen las muestras, nos devuelve tres gráficas, en las que podemos ver la representación en trazo continuo de la señal analógica, las muestras tomadas y las transformadas de Fourier en trazo continuo y la obtenida mediante las muestras.

En el **capítulo 6**: comprobamos que hemos cumplido los objetivos que nos propusimos al inicio de este trabajo, no sólo en cuanto a que los cálculos de la DFT sean correctos, si no también al tiempo de ejecución de nuestros programas.

Por último, en este trabajo incluimos los siguentes anexos:

En el anexo A, estudiamos la ecuación de recurrencia que satisface el tiempo de ejecución de un algoritmo. En el anexo B, explicamos el cálculo y el almacenamiento, no trivial, de las raíces de la unidad que necesitamos tener almacenadas para su uso en los diferentes algoritmos de implementación de la FFT. En el anexo C mostramos los códigos realizados en Java para implementar los diferentes algoritmos de cálculo de la FFT. En el anexo D incluimos los códigos que necesitamos para la multiplicación de números grandes. En el anexo E, explicamos una forma curiosa de definir la transformada de Fourier en $L^2(\mathbb{R})$ (debida a Norbert Wiener), usando los polinomios de Hermite y demostramos el Teorema integral de Fourier, para funciones en $L^2(\mathbb{R})$, que necesitamos usar en la demostración del Teorema del muestreo. Por último, en el anexo F, mostramos el código en Matlab, del programa comentado anteriormente, que usamos para la recuperación de señales en términos de frecuencias.

Abstract

The main goal of this work is to study the fast Fourier transform (FFT), an extremely powerful tool of mathematical analysis which is used in a great variety of science disciplines, including physics, engineering, medicine, psychology, geology and many others. In particular, this work also includes the theoretical study and the practical implementation of several useful algorithms which compute the FFT and a detailed discussion of two well known applications of the FFT. Concretely, we study fast multiplication of large numbers and the numerical approximation, in the frequency domain, of one-dimensional analog signals.

In 1805, Gauss[3] developed the first known "divide-and-conquer" algorithm, which was focused on the computation of the discrete Fourier transform (DFT) of a vector, using an iterative procedure. In 1965, just 150 years later, two North American scientists, J.W. Cooley and J.W. Tukey [4], rediscovered the most efficient algorithm known to their date for the computation of the DFT, and named it as Fast Fourier transform (FFT). Nowadays, this algorithm is considered one of the biggest discoveries in the second half of the XXth century and its study has produced thousands of papers and is permanently used by almost every technological device which manages any kind of information to communicate it or to manipulate it.

As in this work we are interested in the study of the FFT, a method to efficiently calculate the DFT, it is necessary to include, at the very beginning of the document, the definition and most important properties of the DFT. In this context, we introduce it in **chapter 1**. The discrete Fourier transform, which can be understood as a change of coordinates in $\mathbb{C}^n$ between two bases with a special meaning, is defined as the linear transformation $\mathcal{F} : \mathbb{C}^N \longrightarrow \mathbb{C}^N$ that sends any vector $f = (f(0), f(1), \cdots, f(N-1)) \in \mathbb{C}^N$ to another vector $\widehat{f} = (\widehat{f}(0), \widehat{f}(1), ..., \widehat{f}(N-1))$ whose coordinates are given by

$$\widehat{f}(k) = \sum_{n=0}^{N-1} f(n) e^{\frac{-2\pi i k n}{N}}$$

for $0 \leq n \leq N-1$. Together with some of its properties, the concept of convolution is also introduced in this chapter. The importance of the convolution is based on the fact that this operation completely characterizes linear translation invariant systems (LTI) defined over N-periodic signals $f : \mathbb{Z} \to \mathbb{C}$. LTI systems are also named "wave filters" and they are very useful in a broad variety of applications.

In **chapter 2**, the FFT is defined as any algorithm to compute the DFT of a vector of size N with a computational cost of order $N \log(N)$ (this means that the algorithm requieres $O(N \log N)$ operations). This of course implies a significant reduction in the cost of the computation of the DFT, which would be of order $O(N^2)$ operations if a naive algorithm is used. Furthermore, it is a main open problem in Theory of Complexity to decide whether a stronger reduction of the

[3] C.F. Gauss "Theoria Interpolationis Methodo Nova Tractata" en Königlichen Gesellschaft der Nissenschaften, Band 2, pag 265-330, Göttingen 1805

[4] J.W. Cooley y J.W. Tukey "An algorithm for machine calculation of complex Fourier transform", Math. Computation, Vol 19, pages 297-301, 1965

computational cost of the calculation of the DFT of an arbitrary vector of size N is (or is not) possible.

The most basic FFT algorithms are the so called RADIX2 FFT. They are iterative algorithms to compute the DFT by dividing each step of the problem in two other problems with half of the size of the original one. These algorithms have the restriction that the input vector must be of size a power of 2 ($N = 2^n$ for a certain $n \in \mathbb{N}$).

In this chapter, two algorithms based on the 'divide-and-conquer" paradigm are proposed. The first one is the so called decimation in time FFT or DIT-FFT, which is based on dividing the sum that defines the DFT in two summands, where in the first one, only the even terms appear and, in the second one, only the odd terms do it. To do this, an scheme of computations named "Cooley-Tukey's butterfly" by its appearance, is applied. This scheme defines in a very precise form how to move in the different subproblems which are defined by the algorithm.

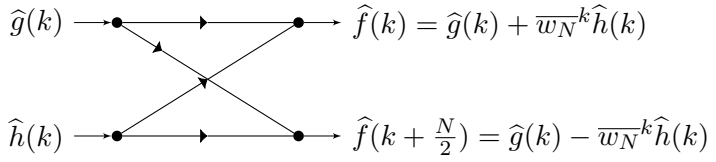


Figure 2: See section 2.1, page 11

The second algorithm we explain is the so called decimation in frequency algorithm (DIF FFT), which, once more, is based on dividing the sum that defines the DFT into two sums. This time, in the first term we sum the first half of the terms appearing in the original sum, and, in the second term, the second half of the terms are summed. The scheme describing this algorithm is the so called Gentleman-Sande's butterfly. Both algorithms have been implemented in Java by using the Java development environment.

The computational cost of an algorithm is usually expressed as floating point operations or flops. We demonstrate that the computational cost of the two described algorithms is given by $5(\log_2(N))N$.

In chapter 3, the details of DIF FFT algorithm implementation are shown. First, the input data (coordinates of vector f) are disposed in its natural order:

$$(f(0), f(1), ..., f(N-1)).$$

DIF FFT algorithm consist in: giving an initial vector of size $N = 2^m$, decompose the problem of the calculation of the DFT of this vector by applying the corresponding butterfly operation in an iterative way and dividing it into sub-problems with half size of the size of the previous step.

In order to avoid the static complexity derived from an iterative algorithm in the context of memory consumption, a single vector is used to realize all necessary operations, minimizing the vector position movements. Due to this imposition of overwriting the results while the algorithm is running, when the algorithm ends, a vector with all the coordinates of the DFT, but ordered with an especial order (bit-reversal), is obtained.

To understand this order, we need to use the binary representation of any integer $0 \leq L \leq N$. This is given by $L = (i_{n-1}, i_{n-2}, .., i_1, i_0)_2$, where $i_k \in \{0, 1\}$ $(k = 0, 1, ..., n-1)$ and

$$L = i_{n-1} \cdot 2^{n-1} + ... + i_1 \cdot 2^1 + i_0 \cdot 2^0$$

The bit-reversal permutation is defined by $\varphi(L) = (i_0, i_1, ..., i_{n-2}, i_{n-1})_2$.

Motivated by the fact that the output of the algorithm depends on the order of the coordinates of the input vector, we introduce two variants of the $DIF-FFT$ algorithm. On the one hand, we have the $DIF_{RN}FFT$ algorithm, where the initials (RN) mean that the algorithm input is in bit-reversal order (R) and the output algorithm is in natural order (N). This algorithm is based on decomposing the inicial problem of size N in subproblems of size 2 and doubling the size of the subproblems in relation to the size of the previous subproblems, where the corresponding butterflies are used. On the other hand, we consider the $DIF_{NN}FFT$ algorithm. In this algorithm, the input and the output are in natural order. We say that this algorithm is an ordered algorithm. This algorithm is based on reading each butterfly as a step of permutation followed by the computation of the butterfly, where the permutation steps order the outputs and inputs of the algorithm.

In order to reduce the computational cost of the FFT algorithm, the RADIX4 FFT algorithm is considered in **chapter 4**. This algorithm applies to vectors of size $N = 4^n = 2^{2n}$, with $n \in \mathbb{N}$. Following a similar approach to the one followed by RADIX2 algorithms, the RADIX4 DIT and DIF FFT algorithms are developed, with the main difference that in these cases we divide each problem in 4 subproblems. The computational cost of these algorithms has been estimated to be $\frac{17N\log_2(N)}{4} - \frac{43N}{6} + \frac{32}{3}$, where $N = 4^n$ is the size of the input vector.

A comparison of this computational cost with the cost of the RADIX2 algorithm renders:

$$\lim_{x\to\infty} \frac{S(N)}{T(N)} = \frac{\frac{17}{4}}{5} = \frac{17}{20},$$

which demonstrates a 15 % reduction of computational cost when using RADIX4 instead of RADIX2 algorithms.

Once RADIX4 and RADIX2 algorithms have been introduced, it is natural to consider another algorithm named SPLIT which improves the computational cost of both algorithms by a combination of the ideas behind them to create a new butterfly, and renders a reduction of 20 % when compared to RADIX2FFT and 5 % when compared to RADIX4FFT.

In **chapter 5**, two important applications of the FFT are shown. The first one is the multiplication of large numbers. The computational cost of multiplying two natural numbers, each one of size n (where n denotes the number of digits used in the decimal expansion of the number) is known to be $O(n^2)$. Now, this cost can be reduced to $O(n\log n)$ by using the implemented algorithms, described in chapter 2 and 3.

Giving a natural number A and a fixed base b, the number A can be expressed as:

$$A = a_0b^0 + a_1b^1 + ... + a_{n-1}b^{n-1},$$

for certain digits $a_i \in 0, 1, 2.., b-1$, This conduces to a close relationship between multiplication of large numbers (meaning by this, numbers whose expansion in a fixed base b requires a large amount n of digits, $A = (a_{n-1}, \cdots, a_0)_b$) and multiplication of two polynomials. In fact, we simultaneously face both problems.

Multiplying two polynomials $p(x) = a_0+a_1x^1+...+a_{n-1}x^{n-1}$ and $q(x) = b_0+b_1x^1+...+b_{n-1}x^{n-1}$ results a third polynomial

$$p(x)\cdot q(x) = a_0b_0 + (a_0b_1 + a_1b_0)x + ... + (a_{n-1}b_{n-1})x^{2n-2},$$

and the direct naive algorithm to get such a product has computational cost of order $O(n^2)$, which is equivalent to the cost of multiplying two integers by the standard algorithm. To reduce this cost, another representation of polynomials is introduced.

Considering any family $\{(x_i, y_i)\}_{i=0}^{n-1}$ of n points with pairwise distinct abscissae (x_i), Lagrange's Interpolation theorem guarantees that there exist a unique algebraic polynomial $p(x)$ of degree at most $n-1$ such that $p(x_i) = y_i$ for $i = 0, \ldots, n-1$. This polynomial is named "interpolation

polynomial" (or "Lagrange interpolation polynomial") of the family of points $\{(x_i, y_i)\}_{i=0}^{n-1}$ and the result can be interpreted as the claim that a polynomial of degree $n-1$ is uniquely characterized through its evaluation at any set of n pairwise distinct nodes $\{x_i\}_{i=0}^{n-1}$. Moreover, the result also holds true for complex quantities x_i and complex polynomials $p(z) = a_0 + a_1 z + \cdots + a_{n-1} z^{n-1} \in \mathbb{C}[z]$. This theorem allows us to guarantee that if $p(x)$ and $q(x)$ are polynomials of degree at most $n-1$ and since their product has degree smaller than $2n-1$, this polynomial can be completely recovered from its evaluation at $2n$ different points.

The key point here is that the evaluation of a polynomial $p(z)$ at the n-th roots of unity $e^{\frac{-2\pi ik}{n}}$ produces the Fourier transform of the vector formed by its coefficients,

$$p(e^{\frac{-2\pi ik}{n}}) = \sum_{j=0}^{n-1} a_j e^{\frac{-2\pi ikj}{n}}.$$

Hence, we can use FFT algorithms to compute these evaluations (which are named DFT of the polynomial $p(x)$) with a reduced computational cost.

Since FFT algorithms need vector inputs with size a power of 2 and the coefficients of each of the polynomials are vectors of size n, they are completed with zeros to the position a power of 2 which should be $\geq 2n$, i.e. the vector of coefficients of p will be of the form $a = (a_0, a_1, ..., a_{n-1})$ and the corresponding vector of coefficients of q will be of the form $b = (b_0, b_1, ..., b_{n-1})$. Thus, these vectors will be completed to $a' = (a_0, a_1, ..., a_{n-1}, 0..., 0)$ and $b' = (b_0, b_1, ..., b_{n-1}, 0..., 0)$, which are vectors of size 2^k, where k is the smallest integer such that $2^k \geq 2n$. The product of the two polynomials can be understood as the convolution of these two vectors and its DFT is the point product of the FFT of the two vectors. Therefore, the outputs of the FFT of both vectors have to be multiplied component-wise and, subsequently, we apply the inverse FFT of the resulting vector to get the polynomial that represents the product of the numbers.

Hence, to reduce the computational cost of multiplication of large numbers using the FFT algorithms the next steps have to be followed:

- Express each of the factors in base 10 and place the coefficients of the corresponding polynomial in a vector.
- Complete, adding zeros, the vector of coefficients of each of the factors to the first power of 2 which satisfies the inequality $2^k \geq 2n$, where n is the maximum of the degrees of the polynomials given by the previous step.
- Apply the FFT to each vector computed in this way.
- Multiply, coordinate by coordinate, the two vectors obtained in the previous step.
- Apply the inverse FFT to the vector we got in the previous step.

The second application we consider in this work theoretically demonstrates how the FFT can be used in a real case. Real life is plenty of situations where it is necessary to analyze the frequency components of an analogical signal $x(t)$, which are defined by the so called (analogical) Fourier Transform, $\mathcal{F}(x)(\xi) = \int_{-\infty}^{\infty} x(t) e^{-2\pi it\xi} dt$ (this function is also denoted by $\widehat{x}(\xi)$ and, in Engineering applications, by $X(\xi)$). Nevertheless, for real data it is usually impossible to tackle the calculation of the Fourier Transform of the signal directly. This is due to either to the impossibility of knowing an analytical expression of the signal $x(t)$ or to the impossibility of computing the involved integrals. On the other hand, in the digital world, the computation of frequencies of signals can always be achieved and this is done precisely by the DFT which, fortunately, can be fast computed by FFT algorithms. Now, we are lucky because a conversion of analogical signals to digital ones with no loss

of information can be achieved for all real applications. This is so thanks to the following important result of Fourier Analysis:

Sampling theorem: Let $x(t) \in L^2(\mathbb{R})$ be such that $\widehat{x}(\xi) = 0$ for all $|\xi| > b$. Then $x(t)$ can be completely recovered from its values at the discrete set of points $\{\frac{k}{2b}\}_{k\in\mathbb{Z}}$. Moreover, the following formula is satisfied:

$$x(t) = \sum_{k=-\infty}^{\infty} x(\frac{k}{2b}) \frac{sin(\pi(-k+2bt))}{\pi(-k+2bt)},$$

where the series converges in $L^2(\mathbb{R})$ and also converges absolutely and uniformly in $\mathbb{R}$.

Once in the digital world, the FFT can be applied to samples acquired as is indicated by the sampling theorem, which produces a description of the sampled signal in terms of digital frequencies. Last, through reordering and re-scaling techniques, it is possible in many cases to reach a good description of the analogical Fourier transform of the original signal. We explain these techniques and their main limitations in this work.

The previous theorem only allows to perform exact calculations for band limited functions belonging to $L^2(\mathbb{R})$. This fact excludes many common and important functions as the trigonometric ones. Due to this, we state a generalization of the Sampling Theorem, whose proof uses the theory of tempered distributions, that overcomes this issue. Moreover, we show the efficacy of these techniques by implementing a Matlab program which, using as input data an analogical function for which its corresponding Fourier transform is known, the desired sampling frequency and the sampling range, computes three plots which contain, in the first plot, the representation of a superposition of the analogical signal with the samples acquired, and in the other two plots, the real and imaginary parts of the known Fourier transform, superposed to the corresponding real and imaginary parts of the approximated Fourier transform obtained from the information given by the samples.

In **chapter 6**, it is shown how the objectives of this work are achieved considering the feasibility of the DFT calculations and the execution time of the developed programs.

Finally, some annexes are included:

In annex A, it is included a study of the iterative equation that fulfills the execution time of the FFT algorithms. In annex B, it is explained the calculation and storage, not trivial, of the squared roots of the unity needed for the use of the FFT using the different developed algorithms. In annex C, it is shown the source code in Java for the different algorithms for FFT computation. In annex D, it is presented the needed source codes for the multiplication of large numbers. In annex E, it is described a not common way to define the Fourier transform in $L^2(\mathbb{R})$ (due to Norbert Wiener), using Hermite polynomials. It is also demonstrated the Fourier integral theorem for functions in $L^2(\mathbb{R})$, which is needed for proving the sampling theorem. Finally in annex F, it is shown the Matlab code used to approximate analogical signals in their frequency domain.

Índice general

1 | Transformada de Fourier discreta

Aunque el objetivo principal de este trabajo es la transformada de Fourier rápida (FFT) y su implementación mediante diversos algoritmos, debemos tener unos conocimientos previos de la transformada de Fourier discreta (DFT), ya que la transformada de Fourier rápida es un algoritmo para el cálculo eficaz de la transformada de Fourier discreta.

En este capítulo haremos una introducción a la transformada de Fourier de funciones finitas y discretas. La DFT no es otra cosa que un cambio de coordenadas en $\mathbb{C}^N$ entre dos bases que tienen un significado especial. Los contenidos de este capítulo hacen referencia al libro [14], a la página web [15] y al artículo [10] (también mirar [12]).

Definición 1.1:

Una **señal discreta** de tamaño N es, por definición, un vector $f = (f_0, f_1, ..., f_{N-1})$. Estas señales se interpretan de forma natural como funciones $f : \{0, ..., N-1\} \to \mathbb{C}$ simplemente forzando la igualdad $f_k = f(k)$ para $k = 0, ..., N-1$.

Como numerosas expresiones que aparecen ligadas a la DFT (que definiremos en breve) se corresponden con funciones periódicas, resulta natural identificar el vector $f = (f_0, f_1, ..., f_{N-1})$ (o, lo que es lo mismo la función $f : \{0, 1, ..., N-1\} \to \mathbb{C}$) con la única función N-periódica $f : \mathbb{Z} \to \mathbb{C}$ que verifica $f(k) = f_k$ para $k = 0, 1, ..., N-1$.

A lo largo de todo el trabajo vamos a denotar por $\mathbb{P}_N$ al conjunto de las funciones N-periódicas definidas sobre los enteros y con valores complejos.

Definición 1.2:

Llamaremos **producto escalar habitual** al definido de la siguiente forma:

$$\langle f, g \rangle = \sum_{k=0}^{N-1} f_k \overline{g_k}$$

donde $\overline{\mathbb{Z}}$ denota el complejo conjugado del número complejo $\mathbb{Z}$.

Las dos bases con un significado especial comentadas anteriormente son las siguientes:

- Por un lado tenemos la base canónica, formada por los vectores de la forma $e_k[i] = 0$ si $i \neq k$ y $e_k[k] = 1$. Cuando escribimos $f = (f(0), f(1), ..., f(N-1))$ estamos escribiendo, $f = f(0)e_0 + ... + f(N-1)e_{N-1} = \sum_{k=0}^{N-1} f_k e_k$, es decir, estamos considerando la representación de la señal f en el dominio del tiempo.

- Por otro lado tenemos la base que nos describe estas señales en términos de frecuencias, $\{E_k\}_{k=0}^{N-1}$. Se trata de la base formada por los vectores de la forma $E_k = \sum_{n=0}^{N-1} e^{\frac{2i\pi kn}{N}} e_n$.

Veamos la ortogonatidad de la familia $\{E_k\}_{k=0}^{N-1}$.

Teorema 1.1:

Los vectores $\{E_k\}_{k=0}^{N-1}$ forman una base ortogonal de $\mathbb{C}^N$ con el producto escalar habitual.

Demostración. En primer lugar como $\mathbb{C}^N$ es un espacio vectorial de dimensión finita N, y el conjunto $\{E_k\}_{k=0}^{N-1}$ tiene N elementos, basta con demostrar que estos vectores son linealmente independientes para concluir que son una base de $\mathbb{C}^N$. Ahora bien, todo sistema ortogonal formado por vectores no nulos es linealmente independiente (esto es obvio, pues si $\alpha_0 E_0 + \ldots + \alpha_{N-1} E_{N-1} = 0$ entonces, para cada $k \in \{0, 1, ..., N-1\}$, se tiene que $0 = \langle 0, E_k\rangle = \langle \alpha_0 E_0 + \ldots + \alpha_k E_K + \ldots + \alpha_{N-1} E_{N-1}, E_k\rangle = \alpha_k \langle E_k, E_k\rangle \Rightarrow \alpha_k = 0$).

Se sigue que sólo queda demostrar la ortogonalidad de nuestro sistema.

Para comprobar que es ortogonal distinguimos dos casos:

1. Si $k \neq p$

$$\begin{array}{rclcll} \langle E_k, E_p\rangle & = & \sum_{n=0}^{N-1} E_k(n)\overline{E_p(n)} & = & \sum_{n=0}^{N-1} e^{\frac{2i\pi kn}{N}} e^{\frac{-2i\pi pn}{N}} & \\ & = & \sum_{n=0}^{N-1} \left(e^{\frac{2i\pi(k-p)}{N}}\right)^n & = & \frac{e^{\frac{2i\pi(k-p)N}{N}}-1}{e^{\frac{2i\pi(k-p)}{N}}-1} & \\ & = & \frac{e^{2i\pi(k-p)}-1}{e^{\frac{2i\pi(k-p)}{N}}-1} & = & \frac{1-1}{e^{\frac{2i\pi(k-p)}{N}}-1} & = \quad 0 \end{array}$$

donde en las igualdades anteriores hemos tenido en cuenta : $\overline{e^{\theta i}} = e^{-\theta i}$, la fórmula de la suma de una progresión geométrica, es decir, $1 + z + ... + z^{N-1} = \frac{z^N - 1}{z-1}$ y que si $0 \leq k$, $p \leq N$ y $k \neq p$, entonces $e^{\frac{2i\pi(k-p)}{N}} \neq 1$.

2. Si $k = p$

$$\begin{array}{rclcll} \langle E_k, E_k\rangle & = & \sum_{n=0}^{N-1} E_k(n)\overline{E_k(n)} & = & \sum_{n=0}^{N-1} e^{\frac{2i\pi kn}{N}} e^{\frac{-2i\pi pn}{N}} & \\ & = & \sum_{n=0}^{N-1} \left(e^{\frac{2i\pi(k-p)}{N}}\right)^n & = & \sum_{n=0}^{N-1} e^{\frac{2i\pi kn}{N}} e^{\frac{-2i\pi kn}{N}} & \\ & = & \sum_{n=0}^{N-1} 1 & = & N. & \end{array}$$

□

Ya podemos definir y estudiar las propiedades elementales de la transformada de Fourier:

Definición 1.3:

Se define la **transformada de Fourier discreta** como la transformación $\mathfrak{F}_{\mathfrak{N}} : \mathbb{P}_N \longrightarrow \mathbb{P}_N$ que a cada vector $f \in \mathbb{C}^N$ le hace corresponder el vector $\widehat{f} = (\widehat{f}(0), \widehat{f}(1), ..., \widehat{f}(N-1))$ donde

$$\widehat{f}(k) = \sum_{n=0}^{N-1} f(n) e^{\frac{-2i\pi kn}{N}} = \langle f, E_k\rangle, \tag{1.1}$$

para $0 \leqslant n \leqslant N-1$.

Teorema 1.2:

Para $n = 0, 1, .., N-1$ se verifica que:

$$f(n) = \frac{1}{N} \sum_{k=0}^{N-1} \widehat{f}(k) e^{\frac{2i\pi kn}{N}}$$

Demostración. Como la familia $\{E_k\}_{k=0}^{N-1}$es ortogonal podemos escribir

$$\begin{aligned} f &= \sum_{k=0}^{N-1} \frac{\langle f, E_k\rangle}{\langle E_k, E_k\rangle} E_k \\ &= \frac{1}{N} \sum_{k=0}^{N-1} \widehat{f}(k) E_k \end{aligned}$$

Por tanto:

$$\begin{aligned} f(n) &= \frac{1}{N} \sum_{k=0}^{N-1} \widehat{f}(k) E_k(n) \\ &= \frac{1}{N} \sum_{k=0}^{N-1} \widehat{f}(k) e^{\frac{2i\pi kn}{N}}. \end{aligned}$$

□

Definición 1.4:

Se define la **transformada de Fourier inversa** como la transformación $\mathfrak{F}_{\mathfrak{N}}^{-1} : \mathbb{P}_N \longrightarrow \mathbb{P}_N$ dada por $\mathfrak{F}_{\mathfrak{N}}^{-1}\{f\} = \check{f}$, donde para $0 \leqslant n \leqslant N-1$,

$$\check{f}(n) = \frac{1}{N} \sum_{k=0}^{N-1} f(k) e^{\frac{2i\pi kn}{N}} = \frac{1}{N} \langle f, \overline{E_n}\rangle = \frac{1}{N} \langle f, E_{-n}\rangle, \tag{1.2}$$

Como f está identificada por los N valores de la función en $n = 0, 1, ..., N-1$, a lo largo del trabajo usaremos $f = (f(0), f(1), ..., f(N-1))$, como ya hemos usado en la definición de señal discreta, en Definición 1.1.

De la definición de transformada de Fourier discreta se siguen las siguientes propiedades.

Proposición 1.1:

Sean $f_1, f_2, ..., f_m$, g elementos de $\mathbb{P}_N$ y sean $\widehat{f_1}, \widehat{f_2}, ..., \widehat{f_m}$, $\widehat{g}$ sus respectivas transformadas de Fourier. Entonces se verifican las siguientes propiedades:

1. **Linealidad:** Si $g(n) = c_1 f_1(n) + c_2 f_2(n) + ... + c_m f_m(n)$ entonces su transformada de Fourier es $\widehat{g}(n) = c_1 \widehat{f_1}(n) + c_2 \widehat{f_2}(n) + ... + c_m \widehat{f_m}(n)$.
2. **Reflexión:** Si $g(n) = f(-n)$ entonces tiene como transformada de Fourier $\widehat{g}(k) = \widehat{f}(-k)$.
3. **Conjugación:** Si $g(n) = \overline{f(n)}$ entonces su tranformada de Fourier es $\widehat{g}(k) = \overline{\widehat{f}(-k)}$
4. **Translación:** Si $g(n) = f(n - n_0)$ entonces su transformada de Fourier es $\widehat{g}(k) = e^{\frac{-2i\pi kn_0}{N}} \widehat{f}(k)$, para $n_0 = 0, \pm 1, \pm 2, ...$.
5. **Modulación:** Si $g(n) = e^{\frac{2i\pi nk_0}{N}} f(n)$ entonces su transformada de Fourier es $\widehat{g}(k) = \widehat{f}(k - k_0)$, para $k = 0, \pm 1, \pm 2, ...$
6. **Inversión:** Si $g(n) = \widehat{f}(n)$ entonces su transformada de Fourier es $\widehat{g}(k) = Nf(-k)$.

Demostración. 1. Para demostrar la linealidad basta realizar los siguientes cálculos:

- Sea $g(n) = cf(n)$. Entonces

$$\widehat{g}(k) = \sum_{n=0}^{N-1} cf(n)e^{\frac{-2i\pi kn}{N}} = c\sum_{n=0}^{N-1} f(n)e^{\frac{-2i\pi kn}{N}} = c\widehat{f}(k).$$

- Sea $g(n) = f_1(n) + f_2(n)$. Entonces

$$\begin{aligned}\widehat{g}(k) &= \sum_{n=0}^{N-1} g(n)e^{\frac{-2i\pi kn}{N}} \\ &= \sum_{n=0}^{N-1}(f_1(n) + f_2(n))e^{\frac{-2i\pi kn}{N}} \\ &= \sum_{n=0}^{N-1} f_1(n)e^{\frac{-2i\pi kn}{N}} + \sum_{n=0}^{N-1} f_2(n)e^{\frac{-2i\pi kn}{N}} \\ &= \widehat{f_1}(k) + \widehat{f_2}(k).\end{aligned}$$

2. Sea $g(n) = f(-n)$. Entonces

$$\widehat{g}(k) = \sum_{n=0}^{N-1} g(n)e^{\frac{-2i\pi kn}{N}} = \sum_{n=0}^{N-1} f(-n)e^{\frac{-2i\pi kn}{N}}.$$

Haciendo el cambio de variable $u = -n$, obtenemos

$$\widehat{g}(k) = \sum_{n=0}^{N-1} f(u)e^{\frac{-2i\pi(-k)u}{N}} = \widehat{f}(-k).$$

3. Sea $g(n) = \overline{f(n)}$. Entonces

$$\begin{aligned}\widehat{g}(k) &= \sum_{n=0}^{N-1} g(n)e^{\frac{-2i\pi kn}{N}} \\ &= \underline{\sum_{n=0}^{N-1} \overline{f(n)}e^{\frac{-2i\pi kn}{N}}} \\ &= \overline{\sum_{n=0}^{N-1} f(n)e^{\frac{-2i\pi(-k)n}{N}}} \\ &= \widehat{f}(-k).\end{aligned}$$

4.Sea $g(n) = f(n - n_0)$. Entonces

$$\begin{aligned}\widehat{g}(k) &= \sum_{n=0}^{N-1} g(n)e^{\frac{-2i\pi kn}{N}} \\ &= \sum_{n=0}^{N-1} f(n - n_0)e^{\frac{-2i\pi kn}{N}}.\end{aligned}$$

Haciendo el cambio de variable $u = n - n_0$ entonces $n = u + n_0$ (por tanto, $n = 0 \Leftrightarrow u = -n_0$ y $n = N - 1 \Leftrightarrow u = N - 1 - u_0$)

$$\widehat{g}(k) = e^{\frac{-2\pi ikn_0}{N}} \sum_{u=-n_0}^{N-n_0-1} f(u)e^{\frac{-2i\pi ku}{N}}.$$

Ahora como f y E_k son N-periódicas tenemos:

$$\begin{aligned}\widehat{g}(k) &= e^{\frac{-2\pi ikn_0}{N}} \sum_{u=0}^{N-1} f(u)e^{\frac{-2i\pi ku}{N}} \\ &= e^{\frac{-2\pi ikn_0}{N}} \widehat{f}(k).\end{aligned}$$

.

5. Sea $g(n) = e^{\frac{2i\pi k_0 n}{N}} f(n)$. Entonces

$$\begin{aligned}\widehat{g}(k) &= \sum_{n=0}^{N-1} g(n)e^{\frac{-2i\pi kn}{N}} \\ &= \sum_{n=0}^{N-1} f(n)e^{\frac{2\pi ik_0 n}{N}} e^{\frac{-2i\pi kn}{N}} \\ &= \sum_{n=0}^{N-1} f(n)e^{\frac{-2i\pi(k-k_0)n}{N}} \\ &= \widehat{f}(k - k_0).\end{aligned}$$

6. Sea $g(n) = \widehat{f}(n)$. Entonces

$$\begin{aligned}\widehat{g}(k) &= \sum_{n=0}^{N-1} g(n)e^{\frac{-2i\pi kn}{N}} \\ &= \sum_{n=0}^{N-1} \widehat{f}(n)e^{\frac{-2i\pi kn}{N}} \\ &= \frac{N}{N}\sum_{n=0}^{N-1} \widehat{f}(n)e^{\frac{2i\pi(-k)n}{N}} \\ &= Nf(-k).\end{aligned}$$

□

1.1. Convolución de señales discretas y sistemas LTI

Definición 1.5:

Sean f, g dos elementos de $\mathbb{P}_N$ se define su **convolución** como:

$$(f * g)(n) := \sum_{m=0}^{N-1} f(m)g(n-m) \qquad n \in \mathbb{Z} \tag{1.3}$$

es fácil comprobar que $(f * g) \in \mathbb{P}_N$.

De la definición anterior se obtienen las siguientes propiedades.

Proposición 1.2:

La definición de convolución satisface las siguientes propiedades:

1. $*$ es **homogénea**, es decir, para todo $\alpha \in \mathbb{C}$ se verifica que
$$(\alpha f) * f_1 = \alpha(f * f_1).$$
2. $*$ es **distributiva** respecto a la suma, es decir,
$$f * (f_1 + f_2) = (f * f_1) + (f * f_2).$$
3. $*$ es **asociativa**, esto es,
$$f * (f_1 * f_2) = (f * f_1) * f_2.$$
4. $*$ es **conmutativa**, esto es,
$$f * f_1 = f_1 * f.$$

Demostración. 1. De la definición sabemos que

$$\begin{aligned}((\alpha f) * f_1)(n) &= \sum_{m=0}^{N-1}(\alpha f)(m)f_1(n-m) \\ &= \alpha\sum_{m=0}^{N-1} f(m)f_1(n-m) \\ &= \alpha(f * f_1)(n)\end{aligned}$$

2. De la definición sabemos que

$$\begin{aligned} f*(f_1+f_2)(n) &= \sum_{m=0}^{N-1} f(m)\cdot(f_1+f_2)(n-m) \\ &= \sum_{m=0}^{N-1} f(m)f_1(n-m) + \sum_{m=0}^{N-1} f(m)f_2(n-m) \\ &= (f*f_1)(n) + (f*f_2)(n). \end{aligned}$$

3. Por la definición sabemos que

$$f*(f_1*f_2)(n) = \sum_{m=0}^{N-1} f(m)(f_1*f_2)(n-m)$$

y de nuevo por la definición $(f_1*f_2)(n-m) = \sum_{k=0}^{N-1} f_1(k)f_2(n-m-k)$, de modo que tenemos

$$f*(f_1*f_2)(n) = \sum_{m=0}^{N-1} f(m)\left[\sum_{k=0}^{N-1} f_1(k)f_2(n-m-k)\right].$$

Haciendo el cambio de variable $k = u - m \Rightarrow u = k + m$:

$f*(f_1*f_2)(n) = \sum_{m=0}^{N-1} f(m)\sum_{u=m}^{N-1+m} f_1(u-m)$.

Haciendo uso de que f_1 y f_2 son funciones N-periódicas:

$$\begin{aligned} f*(f_1*f_2)(n) &= \sum_{m=0}^{N-1} f(m)\sum_{u=m}^{N-1+m} f_1(u-m)f_2(n-u) \\ &= \sum_{u=0}^{N-1}\left(\sum_{m=0}^{N-1} f(m)f_1(u-m)\right)f_2(n-u) \\ &= \sum_{u=0}^{N-1}(f*f_1)(u)f_2(n-u) \\ &= (f*f_1)*f_2(n). \end{aligned}$$

4. Por la definición tenemos que

$$f*f_1(n) = \sum_{m=0}^{N-1} f(m)f_1(n-m).$$

Haciendo el cambio de variable $t = n - m \Rightarrow m = n - t$:

$$(f*f_1)(n) = \sum_{t=n}^{N-1+n} f(t-n)f_1(t).$$

Haciendo uso de que f y f_1 son funciones N-periódicas:

$$\begin{aligned} (f*f_1)(n) &= \sum_{t=0}^{N-1} f(t-n)f_1(t) \\ &= (f_1*f)(n). \end{aligned}$$

□

Proposición 1.3:

Sean f_1, f_2 y g elementos de $\mathbb{P}_N$. Sean $\widehat{f_1}, \widehat{f_2}$ y $\widehat{g}$ sus transformadas de Fourier. Entonces se verifican las siguientes propiedades:

1. La DFT de $g = f_1 * f_2$ es $\widehat{g} = \widehat{f_1}\cdot\widehat{f_2}$.
2. Análogamente la DFT de $g = f_1\cdot f_2$ es $\widehat{g} = \frac{1}{N}(\widehat{f_1}*\widehat{f_2})$.

Demostración. 1. Suponemos que $g(n) = (f_1 * f_2)(n)$. Por definición

$$\begin{aligned}\widehat{g}(k) &= \sum_{n=0}^{N-1} g(n)e^{\frac{-2\pi ikn}{n}} \\ &= \sum_{n=0}^{N-1}(f_1 * f_2)(n)e^{\frac{-2\pi ikn}{N}} \\ &= \sum_{m=0}^{N-1} f_1(m)\sum_{n=0}^{N-1} f_2(n-m)e^{\frac{-2\pi ikn}{N}} \\ &= \sum_{m=0}^{N-1} f_1(m)e^{\frac{-2\pi ikm}{N}}\sum_{n=0}^{N-1} f_2(n-m)e^{\frac{-2\pi ik(n-m)}{N}} \\ &= \widehat{f_1}(k)\sum_{n=0}^{N-1} f_2(n-m)e^{\frac{-2\pi ik(n-m)}{N}}.\end{aligned}$$

Ahora, como f_2 es N-periódica, tenemos que

$$\sum_{n=0}^{N-1} f_2(n-m)e^{\frac{-2\pi ik(n-m)}{N}} = \sum_{n=0}^{N-1} f_2(n)e^{\frac{-2\pi ikn}{N}} = \widehat{f_2}(k).$$

Por tanto $\widehat{g}(k) = \widehat{f_1}(k)\cdot\widehat{f_2}(k)$.
2. Suponemos que $g(n) = f_1(n)f_2(n)$. Entonces por definición tenemos que:

$$\begin{aligned}\widehat{g}(k) &= \sum_{n=0}^{N-1}(f_1(n)f_2(n))e^{\frac{-2\pi ikn}{n}} \\ &= \sum_{n=0}^{N-1}\left(\frac{1}{N}\sum_{j=0}^{N-1}\widehat{f_1}(j)e^{\frac{2\pi ijn}{N}}\right)f_2(n)e^{\frac{-2\pi ikn}{N}} \\ &= \frac{1}{N}\sum_{j=0}^{N-1}\widehat{f_1}(j)\sum_{n=0}^{N-1} f_2(n)e^{\frac{-2\pi i(k-j)n}{N}} \\ &= \frac{1}{N}\sum_{j=0}^{N-1}\widehat{f_1}(j)\cdot f_2(k-j) \\ &= \frac{1}{N}(\widehat{f_1} * \widehat{f_2})(j).\end{aligned}$$

□

La importancia de la operación de convolución que acabamos de introducir se basa en que ésta caracteriza los llamados sistemas LTI, que, en el contexto en el que se desarrolla este trabajo, son los operadores lineales $L : \mathbb{P}_N \to \mathbb{P}_N$ que tienen la propiedad adicional de ser invariantes frente a traslaciones en el tiempo. Es decir, verifican la relación

$$L(f[\cdot - m]) = L(f)[\cdot - m], \text{ para todo } m \in \mathbb{Z} \text{ y todo } f \in \mathbb{P}_N. \tag{1.4}$$

Esto significa que si introducimos como entrada del sistema definido por el operador L la señal que resulta de trasladar m unidades de tiempo a la entrada $f \in \mathbb{P}_N$, el efecto sobre la salida del sistema es simplemente trasladar la salida, $L(f)$ la misma cantidad de unidades temporales. Este tipo de operadores son muy importantes en Física y en Ingeniería. En particular, cuando un sistema lineal describe una ley física parece razonable que esta no dependa del momento en el que se aplica y, por tanto, debería ser invariante frente a traslaciones temporales (y espaciales, si se aplica en diferentes puntos del espacio). Estos operadores se pueden caracterizar de varios modos. Por ejemplo, podríamos intentar una caracterización de la matriz asociada a uno de estos operadores cuando tomamos la base del tiempo en $\mathbb{P}_N$. Para lograrlo, conviene observar que, si $T : \mathbb{P}_N \to \mathbb{P}_N$ denota el operador traslación $T(f)(n) = f(n-1)$, entonces $T^m(f)(n) = f(n-m)$, y la relación (1.4) se puede escribir como:

$$LT^m = T^m L, \text{ para todo } m \in \mathbb{Z} \tag{1.5}$$

Esto permite demostrar el siguiente resultado técnico:

Lema 1.1:

El operador lineal $L : \mathbb{P}_N \to \mathbb{P}_N$ define un sistema LTI si y solo si $LT = TL$.

Demostración. Ya hemos visto que si L es invariante por traslaciones, entonces $LT^m = T^m L$ para todo $m \in \mathbb{Z}$. En particular, $LT = TL$.

Para realizar nuestra prueba podemos asumir que $m \geq 0$ porque, al ser los elementos de $\mathbb{P}_N$ funciones N-periódicas, se tiene que, si $f \in \mathbb{P}_N$, entonces $T^N(f)(n) = f(n+N) = f(n)$ y, por tanto, $T^N = 1_d$ es la identidad. Se sigue que $T^{-1} = T^{N-1}$ y, por tanto, $T^{-m} = (T^{-1})^m = (T^{N-1})^m = T^{Nm-m}$, siendo $m, Nm - m \geq 0$.

Supongamos ahora que $LT = TL$ y sea $m \in \mathbb{N}$. Veamos por inducción sobre m que $T^m L = LT^m$. Para $m = 1$ es nuestra hipótesis. Lo asumimos cierto para $m - 1$. Entonces

$$LT^m = (LT)T^{m-1} = (TL)T^{m-1} = T(LT^{m-1}) = T(T^{m-1}L) = (TT^{m-1})L = T^m L,$$

que es lo que buscábamos. □

Evidentemente, el resultado anterior se puede escribir en términos de matrices. Para ello, denotemos por P la matriz asociada a T y por A la matriz asociada a L (tomando en ambos casos la base del tiempo tanto en el espacio de salida como en el espacio de llegada). Obviamente, el lema anterior nos dice que L es LTI si y solo si $AP = PA$. Ahora bien, un sencillo cálculo nos conduce a la conclusión de que

$$P = \begin{bmatrix} 0 & 0 & \cdots & 0 & 1 \\ 1 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix}$$

(para verificarlo, basta multiplicar P por f, donde $f^t = (f[0], f[1], \cdots, f[N-1])$, lo que arroja el valor $(f[N-1], f[0], \cdots, f[N-2])^t = (f[-1], f[0], \cdots, f[N-2])^t = T(f)^t$).

La matriz P tiene inversa

$$P^{-1} = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ 1 & 0 & 0 & \cdots & 0 \end{bmatrix},$$

por lo que $AP = PA$ si y solo si $A = PAP^{-1}$, pero un sencillo cálculo conduce a la igualdad

$$PAP^{-1} = (a_{[i-1]_N,[j-1]_N})_{i,j=0}^{N-1},$$

donde $A = (a_{i,j})_{i=0}^{N-1}$ y $(k)_N$ representa la clase de $k \in \mathbb{Z}$ módulo N, tomada en $\{0, 1, \cdots, N-1\}$, es decir, $[i-1]_N$ hace referencia a $i-1$ módulo N. Esto significa que $A = (a_{i,j})_{i=0}^{N-1}$ es la matriz asociada (en la base del tiempo) a un sistema LTI si y solo si

$$a_{i,j} = a_{[i-1]_N,[j-1]_N} \text{ para todo } i, j = 0, \cdots, N-1.$$

Es decir, A es una matriz circulante,

$$A = \begin{bmatrix} c_0 & c_1 & c_2 & \cdots & c_{N-1} \\ c_{N-1} & c_0 & c_1 & \cdots & c_{N-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_2 & c_3 & c_4 & \cdots & c_1 \\ c_1 & c_2 & c_3 & \cdots & c_0 \end{bmatrix} = \begin{bmatrix} c_{[0]_N} & c_{[1]_N} & c_{[2]_N} & \cdots & c_{[N-1]_N} \\ c_{[-1]_N} & c_{[0]_N} & c_{[1]_N} & \cdots & c_{[N-2]_N} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_{[-(N-2)]_N} & c_{[-(N-3)]_N} & c_{[-(N-4)]_N} & \cdots & c_{[-(N-1)]_N} \\ c_{[-(N-1)]_N} & c_{[-(N-2)]_N} & c_{[-(N-3)]_N} & \cdots & c_{[0]} \end{bmatrix}$$

De modo que, si tomamos $h(k) = c_{[-k]}$, el operador L actúa del siguiente modo:

$$L(f) = \begin{bmatrix} c_{[0]_N} & c_{[1]_N} & c_{[2]_N} & \cdots & c_{[N-1]_N} \\ c_{[-1]_N} & c_{[0]_N} & c_{[1]_N} & \cdots & c_{[N-2]_N} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_{[-(N-2)]_N} & c_{[-(N-3)]_N} & c_{[-(N-4)]_N} & \cdots & c_{[-(N-1)]_N} \\ c_{[-(N-1)]_N} & c_{[-(N-2)]_N} & c_{[-(N-3)]_N} & \cdots & c_{[0]} \end{bmatrix} \begin{bmatrix} f(0) \\ f(1) \\ \cdots \\ f(N-1) \end{bmatrix} = \begin{bmatrix} y(0) \\ y(1) \\ \vdots \\ y(N-1) \end{bmatrix}$$

donde

$$y(k) = \sum_{s=0}^{N-1} f(s) c_{[s-k]_N} = \sum_{s=0}^{N-1} f(s) c_{[-(k-s)]_N} = \sum_{s=0}^{N-1} f(s) h(k-s) = (f*h)(k), \text{ para } k = 0, \cdots, N-1.$$

Esto demuestra que la operación de convolución caracteriza completamente los sistemas LTI con entrada y salida en $\mathbb{P}_N$.

Si consideramos uno de tales sistemas, $L(f) = f * h$, tendremos que $L(e_1) = h$, por lo que h se denomina "respuesta al impulso unidad" del sistema, y lo caracteriza en el dominio del tiempo. Si tomamos la transformada de Fourier discreta a ambos lados de la ecuación $L(f) = f * h$, esta queda en la forma

$$\widehat{L(f)} = \widehat{f} \cdot \widehat{h},$$

donde $v \cdot w$ denota el producto componente a componente de los vectores v, w. En particular, el vector $H = \widehat{h}$ caracteriza la acción del operador L en el dominio de las frecuencias. Obsérvese que, si tomamos H de modo que, para ciertos valores de k, $H(k) = 1$ mientras que $H(k) = 0$ en el resto de valores, el efecto del operador $L(f) = f * h$ (donde $\widehat{h} = H$) es filtrar la señal f, eliminando ciertas frecuencias y dejando pasar con libertad otras. Es por esta razón que los sistemas LTI reciben también el nombre de "filtros de ondas" y son muy útiles en todo tipo de aplicaciones.

2 | Introducción a la transformada de Fourier rápida

En este capítulo vamos a definir la transformada de Fourier rápida y los algoritmos básicos que la describen, con sus costes aritméticos correspondientes. Los contenidos de este capítulo están basados en los libros [4], [7], [8] y [9].

Calcular la transformada discreta de Fourier mediante la fórmula $\widehat{f}(k) = \frac{1}{N}\sum_{n=0}^{N-1} f(n)e^{\frac{-2i\pi kn}{N}}$ puede resultar muy costoso ya que el número de operaciones (multiplicaciones y sumas) a realizar para calcular el vector $(\widehat{f}(0), \widehat{f}(1), ..., \widehat{f}(N-1))$ es el siguiente:

- $\widehat{f}(0) = \sum_{n=0}^{N-1} f(n)e^{-0} = \sum_{n=0}^{N-1} f(n) \Rightarrow$ 0 multiplicaciones complejas y $(N-1)$ sumas complejas.
- $\widehat{f}(1) = \sum_{n=0}^{N-1} f(n)e^{\frac{-2i\pi n}{N}} \Rightarrow (N-1)$ multiplicaciones complejas y $(N-1)$ sumas complejas.

$$\vdots$$

- $\widehat{f}(N-1) = \sum_{n=0}^{N-1} f(n)e^{\frac{-2i\pi n(N-1)}{N}} \Rightarrow (N-1)$ multiplicaciones complejas y $(N-1)$ sumas complejas.

Por tanto para calcular el vector $(\widehat{f}(0), \widehat{f}(1), ..., \widehat{f}(N-1))$ necesitamos $N(N-1)$ sumas complejas y $(N-1)(N-1)$ multiplicaciones complejas, es decir, un total de $N(N-1)+(N-1)^2$ operaciones, lo cual, para un valor típico de N como puede ser $N = 1000$, implicaría un millón de operaciones de cada tipo.

Teniendo en cuenta la gran cantidad de operaciones que requiere el cálculo de la transformada de Fourier discreta es natural buscar un algoritmo que la calcule con un coste computacional sensiblemente menor.

Fué Gauss en 1805 quién creó el algoritmo de "divide y vencerás", es decir, un algoritmo basado en calcular el vector $(\widehat{f}(0), \widehat{f}(1), ..., \widehat{f}(N-1))$ de forma recursiva dividiendo cada componente en dos sumas, aunque no se ocupó de precisar computacionalmente el número de operaciones requeridas por su algoritmo.

Un siglo y medio después, en 1965, dos científicos norteamericanos, J.W.Cooley y J.W.Tukey, redescubrieron el algoritmo más eficiente hasta entonces conocido, al cual le dieron el nombre de **transformada de Fourier rápida**, que fue uno de los grandes descubrimientos de la última mitad del sigo XX. Este algoritmo fue la inspiración en muchas investigaciones de algebra, en el procesamiento de señales e imágenes y en el importante paso en la investigación de minimizar el número de operaciones numéricas.

Definición 2.1:

Definimos la **Transformada de Fourier rápida** como cualquier algoritmo para el cálculo de la Transformada de Fourier discreta que reduzca su complejidad de $O(N^2)$ a $O(N \log N)$ donde N es el tamaño del vector de entrada del algoritmo.

Antes de comenzar con el estudio de los algoritmos básicos debemos tener en cuenta lo siguiente:

- Para realizar el cálculo de la DFT, es decir , dado el vector $f = (f_0, f_1, ..., f_{N-1})$ de tamaño N , calcular el vector $f = (\widehat{f_0}, \widehat{f_1}, ..., \widehat{f_{N-1}})$, donde

$$\widehat{f_k} = \sum_{n=0}^{N-1} f(n) e^{\frac{-2i\pi kn}{N}} \tag{2.1}$$

  con $k = 0, ..., N$, basta calcular el producto de la matriz $\overline{M}$ por el vector columna f, donde

$$\overline{M} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \overline{w_N} & \overline{w_N}^2 & \cdots & \overline{w_N}^{N-1} \\ 1 & \overline{w_N}^2 & \overline{w_N}^4 & \cdots & \overline{w_N}^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \overline{w_N}^{N-1} & \overline{w_N}^{2(N-1)} & \cdots & \overline{w_N}^{(N-1)(N-1)} \end{bmatrix}$$

  y $w_N = e^{\frac{2i\pi}{N}}$.

- Los algoritmos FFT explotan esta estructura empleando el paradigma de "divide y vencerás", cuyos tres pasos principales son:

  1. Dividir el problema en varios subproblemas análogos de tamaños menores.
  2. Resolver de forma recursiva cada uno de los subproblemas con el mismo algoritmo.
  3. Obtener la solución del problema original combinando la solución de cada uno de los subproblemas.

NOTA: A lo largo de este capítulo y del siguiente vamos a asumir que $N = 2^m$ para cierto $m \in \mathbb{N}$.

Definición 2.2:

Llamamos **FFT de radio 2** a un algoritmo recursivo para el cálculo de la DFT que se obtiene dividiendo en cada paso del proceso recursivo el problema original en dos subproblemas de tamaño la mitad del tamaño del problema original.

2.1. Algoritmos básicos FFT

Para calcular (2.1) intentando reducir de forma significativa el número de operaciones necesarias vamos a proponer dos técnicas, ambas basadas en el paradigma de “divide y vencerás” y las cuáles reciben el nombre de: DIT FFT (decimación en tiempo) y DIF FFT (decimación en frecuencia).

2.1.1. DIT FFT radio 2

Este algoritmo está basado en dividir la expresión (2.1) en dos sumas donde por un lado aparecen los términos pares y por otro lado los términos impares, esto es:

$$\begin{array}{rcl} \widehat{f}(k) & = & \sum_{n=0}^{\frac{N}{2}-1} f(2n)\overline{w_N}^{k(2n)} + \sum_{n=0}^{\frac{N}{2}-1} f(2n+1)\overline{w_N}^{k(2n+1)} \\ & = & \sum_{n=0}^{\frac{N}{2}-1} f(2n)(\overline{w_{\frac{N}{2}}})^{kn} + \left(\sum_{n=0}^{\frac{N}{2}-1} f(2n+1)(\overline{w_{\frac{N}{2}}})^{nk}\overline{w_N}^{k}\right) \end{array} \tag{2.2}$$

En las igualdades anteriores hemos tenido en cuenta que

$$\overline{w_{\frac{N}{2}}} = \left(e^{\frac{-2i\pi}{N}}\right)^2 = \overline{w_N}^2.$$

Obsérvese que cada una de las sumas de la expresión (2.2) se puede interpretar como una transformada de Fourier de un vector de tamaño $\frac{N}{2}$.

Llamando $g(k) = f(2n)$ y $h(n) = f(2n+1)$, la primera suma de la expresión (2.2) es la transformada de Fourier de $(g(0), g(1), ..., g(\frac{N}{2}-1))$ que vamos a denotar por $\widehat{g}(k)$ para $k = 0, 1, \ldots, \frac{N}{2}-1$. La segunda suma sin el factor $\overline{w_N}^k$ es la transformada de Fourier discreta de $(h(0), h(1), ..., (h(\frac{N}{2})-1))$ que denotamos por $\widehat{h}(k)$ para $k = 0, ..., \frac{N}{2}-1$.

Ahora bien, con la notación que acabamos de introducir tenemos la igualdad

$$\widehat{f}(k) = \widehat{g}(k) + \overline{w_N}^k\widehat{h}(k)$$

para $k = 0, .., \frac{N}{2}-1$. Esto nos permite obtener los $\frac{N}{2}$ primeros términos.

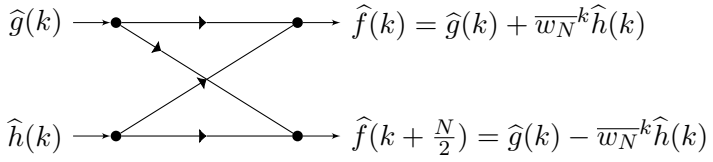
Para obtener los términos restantes hacemos lo que sigue:

$$\begin{array}{rcl} \widehat{f}(k+\frac{N}{2}) & = & \sum_{n=0}^{\frac{N}{2}-1} g(n)\overline{w_{\frac{N}{2}}}^{(k+\frac{N}{2})n} + \overline{w_N}^{k+\frac{N}{2}}\sum_{n=0}^{\frac{N}{2}-1} h(n)\overline{w_{\frac{N}{2}}}^{(k+\frac{N}{2})n} \\ & = & \sum_{n=0}^{\frac{N}{2}-1} g(n)\overline{w_{\frac{N}{2}}}^{(k+\frac{N}{2})n} - \overline{w_N}^{k}\sum_{n=0}^{\frac{N}{2}-1} h(n)\overline{w_{\frac{N}{2}}}^{kn} \\ & = & \widehat{g}(k) - \overline{w_N}^k\widehat{h}(k) \end{array} \tag{2.3}$$

para $k = 0, 1, ..., \frac{N}{2}-1$.

Definición 2.3:

Es usual representar las relaciones anteriores con el siguiente esquema llamado **operación mariposa de Cooley-Tukey**:



La operación de mariposa indica la forma en la que se van combinando los diferentes subproblemas que aparecen en el cálculo recursivo de la DFT mediante el algoritmo "divide y vencerás" que acabamos de describir. Concretamente, si el vector inicial es de tamaño $n = 2^m$, será necesario realizar los cálculos en m etapas, que aparecen al ir descomponiendo el cálculo inicial primero en el cálculo de 2 transformadas de tamaño 2^{m-1}, luego cada uno de los subproblemas dará lugar a otros 2, de tamaño 2^{m-2} (con lo que tendremos $4 = 2^2$ subproblemas de tamaño 2^{m-2}), etc., hasta que en la fase m-ésima hemos llegado a 2^m subproblemas de tamaño $2^0 = 1$. En realidad, el algoritmo arranca resolviendo estos últimos subproblemas -que son triviales, pues la DFT de un número

es él mismo- y, utilizando las soluciones así obtenidas, siguiendo las indicaciones de la mariposa de Cooley-Tukey, se van resolviendo los subproblemas de las etapas anteriores, hasta finalizar. La mariposa nos dice cómo movernos de una etapa a la anterior.

Coste computacinal del algoritmo:

Para calcular el coste computacional del algoritmo, vamos a denotar por $T(N)$ dicho coste y asumimos que los $\overline{w_N}^k$ ya están calculados y almacenados para su uso, como se explica en el Anexo B. Entonces, para obtener el vector $(\widehat{f}(0), \widehat{f}(1), ..., \widehat{f}(N-1))$ a partir de los vectores $(\widehat{g}(0), \widehat{g}(1), ..., \widehat{g}(\frac{N}{2}-1))$ y $(\widehat{h}(0), \widehat{h}(1), ..., \widehat{h}(\frac{N}{2}-1))$ necesitamos:
$\Rightarrow N$ sumas de números complejos.
$\Rightarrow \frac{N}{2}$ multiplicaciones de números complejos, ya que como $\overline{w_N}^k \widehat{h}(k)$ aparece en $\widehat{f}_k$ y en $\widehat{f}_{\frac{N}{2}+k}$ para cada $k = 0, 1, ..., \frac{N}{2}-1$, sólo lo tenemos que calcular una vez.

Ahora bien, el coste de un algoritmo es común expresarlo en términos de operaciones en punto flotante por segundo o flops, en términos del número de operaciones con números reales que se realizan.

Para expresarlo de esta forma tenemos que tener en cuenta las siguientes observaciones:

- La suma de dos números complejos cualquiera, como pueden ser $z_1 = a + ib$ y $z_2 = c + id$ es $z_1 + z_2 = (a + ib) + (c + id) = (a + c) + i(b + d)$, que requiere dos sumas reales.

- Para realizar multiplicaciones complejas en algoritmos FFT se tiene una especial consideración. Como los valores $\overline{w_N}^k$ son siempre uno de los factores implicados en cada multiplicación compleja en el cálculo de la FFT, las múltiplicaciones complejas las vamos a realizar de la siguiente forma: sean $\overline{w_N}^k = c + id$, $\delta = d + c$ y $\varphi = d - c$ (estos dos últimos los suponemos calculados). Como cada múltiplicación que necesitamos es de la forma $\lambda = h_k \overline{w_N}^k$, para calcular λ hacemos uso de que, si definimos los números complejos $h_k = a + ib$,

$$\begin{array}{rcccccl} m_1 & = & (a+b)\cdot c & = & ac+bc & & \\ m_2 & = & \delta \cdot b & = & (c+d)\cdot b & = & c\cdot b + d\cdot b \\ m_3 & = & \varphi \cdot a & = & (d-c)\cdot a & = & d\cdot a - c\cdot a, \end{array}$$

entonces

$$\begin{array}{rcccl} Re(\lambda) & = & a\cdot c - b\cdot d & = & m_1 - m_2 \\ Im(\lambda) & = & a\cdot d + b\cdot c & = & m_1 + m_3 \end{array}$$

de modo que, para calcular λ hemos necesitado 3 multiplicaciones reales y 3 sumas reales, reduciendo el número de multiplicaciones reales (que son las operaciones más costosas para la máquina) de 4 a 3.

Volviendo al coste computacional del algoritmo, las N sumas de números complejos que necesitamos, equivalen a $2N$ sumas de números reales, es decir $2N$ flops. Las $\frac{N}{2}$ multiplicaciones de números complejos equivalen a $\frac{3N}{2} + \frac{3N}{2} = 3N$ multiplicaciones de números reales, es decir, 3N flops. Por tanto $5N$ flops son necesarios para calcular la transformación después de resolver cada subproblema cuyo coste estamos asumiendo que vale $T(\frac{N}{2})$.

Por tanto, tenemos que el coste aritmético $T(N)$ del algoritmo se representa mediante la siguiente fórmula de recurrencia, que es un caso particular de la fórmula que se explica en el Anexo A. De todas formas, como los cálculos para esta fórmula son sencillos, la resolvemos también aquí. En efecto, a partir de la fórmula

$$T(N) = \begin{cases} 2T(\frac{N}{2}) + 5N & \text{si } N = 2^n \geqslant 2, \\ 0 & \text{si } N = 1 \end{cases}$$

se sigue que:

$$\begin{aligned} T(N) = T(2^m) &= 2T(2^{m-1}) + 5 \cdot 2^m \\ &= 2[2T(2^{m-2}) + 5 \cdot 2^{m-1}] + 5 \cdot 2^m \\ &= 2^2 \cdot T(2^{m-2}) + 5 \cdot 2^m + 5 \cdot 2^m \\ &= 2^2 \cdot T(2^{m-2}) + 2 \cdot 5 \cdot 2^m \\ &= 2^2[2 \cdot T(2^{m-3}) + 5 \cdot 2^{m-2}] + 2 \cdot 5 \cdot 2^m \\ &= 2^3 T(2^{m-3}) + 3 \cdot 5 \cdot 2^m \\ &\vdots \\ &\vdots \\ &= 2^m T(1) + m \cdot 5 \cdot 2^m \\ &= 5 \cdot m \cdot 2^m \\ &= 5(\log_2 N) \cdot N, \quad \text{pues } \ T(1) = 0. \end{aligned}$$

2.1.2. DIF FFT radio 2

Empezamos de nuevo con :

$$\begin{aligned} \widehat{f}(r) = \sum_{n=0}^{N-1} f(n)\overline{w_N}^{rn} &= \sum_{n=0}^{\frac{N}{2}-1} f(n)\overline{w_N}^{rn} + \sum_{n=\frac{N}{2}}^{N-1} f(n)\overline{w_N}^{rn} \\ &= \sum_{n=0}^{\frac{N}{2}-1} f(n)\overline{w_N}^{rn} + \sum_{n=0}^{\frac{N}{2}-1} f(n+\tfrac{N}{2})\overline{w_N}^{r(n+\frac{N}{2})} \\ &= \sum_{n=0}^{\frac{N}{2}-1} \left(f(n) + f(n+\tfrac{N}{2}) \cdot \overline{w_N}^{r\frac{N}{2}}\right) \cdot \overline{w_N}^{rn} \end{aligned} \tag{2.4}$$

para $r = 0, 1, ..., N-1$,

Tomando $r = 2k$ en la ecuación (2.4):

$$\begin{aligned} \widehat{f}(2k) &= \sum_{n=0}^{\frac{N}{2}-1} (f(n) + f(n+\tfrac{N}{2})\overline{w_N}^{kN}) \cdot \overline{w_N}^{2kn} \\ &= \sum_{n=0}^{\frac{N}{2}-1} (f(n) + f(n+\tfrac{N}{2})) \cdot \overline{w_N}^{2kn} \\ &= \sum_{n=0}^{\frac{N}{2}} (f(n) + f(n+\tfrac{N}{2})) \cdot \overline{w_{\frac{N}{2}}}^{kn} \\ &= \sum_{n=0}^{\frac{N}{2}} g(n) \cdot \overline{w_{\frac{N}{2}}}^{kn} = \widehat{g}(k) \end{aligned} \tag{2.5}$$

para $k = 0, ..., \frac{N}{2} - 1$. Donde hemos definido $g(n) = f(n) + f(n + \frac{N}{2})$ y hemos usado las igualdades $\overline{w_N}^{kN} = 1$ y $\overline{w_N}^{2kn} = \overline{w_{\frac{N}{2}}}^{2}$.

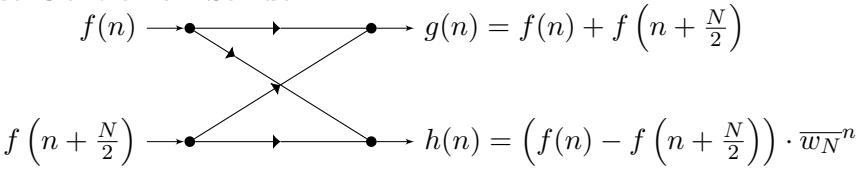
Tomamos ahora en la ecuación (2.4) $r = 2k + 1$:

$$\begin{aligned} \widehat{f}(2k+1) &= \sum_{n=0}^{\frac{N}{2}-1} (f(n) + f(n+\tfrac{N}{2})\overline{w_N}^{(2k+1)\frac{N}{2}}) \cdot \overline{w_N}^{(2k+1)n} \\ &= \sum_{n=0}^{\frac{N}{2}-1} (f(n) - f(n+\tfrac{N}{2}))\overline{w_N}^{n}(\overline{w_{\frac{N}{2}}})^{kn} \\ &= \sum_{n=0}^{\frac{N}{2}-1} h(n)\overline{w_{\frac{N}{2}}}^{kn} = \widehat{h}(k) \end{aligned} \tag{2.6}$$

para $k = 0, .., \frac{N}{2} - 1$. Donde hemos definido $h(n) = (f(n) - f(n + \frac{N}{2}))\overline{w_N}^{n}$ y hemos tenido en cuenta la igualdad $\overline{w_N}^{(2k+1)\frac{N}{2}} = (\overline{w_N}^{N})^k \cdot \overline{w_N}^{\frac{N}{2}} = 1 \cdot (-1) = -1$.

Definición 2.4:

Es usual representar las relaciones anteriores con el siguiente esquema, llamado **operación mariposa de Gentleman-Sande:**



Esta operación mariposa lo que hace es el cálculo de $g(n)$ y $h(n)$ en cada etapa de subdivisión. Como podemos observar en este caso las multiplicaciones por las raíces de la unidad se hacen a posteriori.

Coste computacional del algoritmo:

El cálculo de $g(n)$ y $h(n)$ en el paso de subdivisión requiere N sumas complejas y $\frac{N}{2}$ multiplicaciones complejas. Por tanto, análogamente al coste del algoritmo anterior, $5N$ flops son necesarios para calcular la transformación, además del coste de resolver cada subproblema, que viene dado por $T(\frac{N}{2})$.

Por tanto, el coste aritmético verifica también la relación de recurrencia :

$$T(N) = \begin{cases} 2T(\frac{N}{2}) + 5N & \text{si } N = 2^n \geqslant 2, \\ 0 & \text{si } N = 1, \end{cases}$$

lo que nos lleva a afirmar que $T(N) = 5N \cdot \log_2 N$ nuevamente.

2.1.3. Diferencias entre los métodos DIT FFT (radio 2) y DIF FFT (radio2)

Como hemos visto la FFT se puede implementar con dos algoritmos básicos que provienen o bien de separar en la expresión (2.1) los puntos pares de los impares, es decir, “diezmando en tiempo”, con lo cual llegamos al algoritmo DIT FFT, o bien separamos en dos trozos los puntos que aparecen en la suma que define cada componente de la DFT, dividiendo dicha suma en su primera y su segunda mitad, proceso que recibe el nombre de “diezmando en frecuencia” y que da lugar al algoritmo DIF FFT.

Otra de las diferencias significativas entre ambos métodos se aprecia en las correspondientes operaciones mariposa, ya que en el DIT FFT la multiplicación se hace antes de las sumas y en el DIF FFT esta se hace después.

Finalmente, es importante destacar que el algoritmo DIF FFT proporciona directamente la DFT buscada, cosa que no sucede en el algoritmo DIT FFT.

2.2. Complejidad algoritmica del cálculo de la DFT de un vector

Como hemos visto la aplicación de la técnica “divide y vencerás” al cálculo efectivo de la DFT de un vector da lugar a algoritmos cuyo coste computacional es del orden $O(N \log(N))$, donde N representa la longitud del vector de entrada -parámetro al que también llamamos "tamaño"del vector. En particular, es habitual medir el coste computacional en términos del tiempo que tarda el algoritmo en ser ejecutado. Cabe preguntarse si este orden de magnitud puede (o no) mejorarse. En otras palabras: ¿existe un algoritmo para el cálculo de la DFT que, tomando como entrada vectores de tamaño N se pueda ejecutar en un tiempo de orden inferior a $O(N \log(N))$?. Lo cierto

es que aún no se conocen algoritmos de este tipo y, de hecho, su existencia (o inexistencia) es hoy por hoy un problema abierto que tiene visos de ser extremadamente difícil y se considera uno de los principales problemas de la teoría de algoritmos. Lo que sí se sabe con certeza es que dicho coste computacional no puede ser, para vectores arbitrarios, de orden inferior a $O(N)$ (ver [8]).

Ahora bien, en casos especiales, en los que se imponen ciertas restricciones sobre los vectores a los que se aplican los algoritmos, es posible reducir el coste computacional de forma sensible. Por ejemplo, en el artículo [9] (ver tambié́m [8]) se demuestra que, si las entradas son k-dispersas, es decir, son vectores cuya DFT posee sólo k entradas no nulas aunque no las tengamos identificadas, entonces su DFT se puede calcular con un coste computacional del orden $O(k\log(N))$. Aquí $k = k(N)$ denota un parámetro que puede cambiar con N y que, además, satisface la identidad $k = o(N)$ $(i.e, \lim_{N\to\infty} \frac{k}{N} = 0)$. En particular, se ve que para señales k-dispersas el cálculo de la DFT es sublineal (es decir, con orden menor que $O(N)$). Ahora bien, en numerosas aplicaciones, la mayoría de los términos de la DFT de las señales que se están considerando son nulos o están proximos a cero, por lo que dichas señales se pueden asumir aproximadamente k-dispersas. Por ejemplo, esto sucede con las señales de video, las imágenes y las señales de audio. Y es precisamente esta propiedad de dispersidad la que justifica los algoritmos de compresión de imágenes y sonido, tan frecuentemente utilizados hoy en día.

Existen, por tanto, importantes motivos para estudiar vectores que son aproximadamente k-dispersos. Concretamente, tiene interés encontrar algoritmos que, dada una entrada general x calculen el vector $\widehat{x}^*$ con a lo sumo k entradas no nulas tal que

$$\|dft(x) - \widehat{x}^*\|_{l^2} \leqslant C \qquad \min_{y \ \ es \ \ k-dispersa} \|dft(x) - dft(y)\|_{l^2}$$

para cierta constante de control C que no dependa del tamaño N del vector x, problema que recibe el nombre genérico de "aprocimación k-dispersa de la DFT" y para el cual sí se conocen algoritmos cuyo coste computacional es estrictamente inferior a $O(N)$. Una monografía reciente dedicada exclusivamente a esta cuestión es [8]. Concretamente, en [8] y [9] se muestra un algoritmo de este tipo cuyo coste computacional es $O\left(k\log N\log\left(\frac{N}{k}\right)\right)$, y se demuestra que cualquier algoritmo para el cálculo de una aproximación k-dispersa de la DFT requiere utilizar al menos $\frac{k\log(N/k)}{\log\log N}$ muestras del vector x.

2.3. La transformada de Fourier rápida de datos reales

Como ya sabemos la transformada de Fourier discreta de vectores complejos de tamaño N, se define con la fórmula:

$$\begin{aligned} \widehat{f}(k) &= \sum_{n=0}^{N-1} f(n)\overline{w_N}^{kn} \qquad r = 0, 1, ..., N-1 \\ &= \sum_{n=0}^{N-1} \left(Re(f(n)) + iIm(f(n))\right)\overline{w_N}^{kn} \end{aligned}$$

Cuando los vectores son reales, lo que hacemos es tratarlos como números complejos con parte imaginaria cero, es decir, $Im(f[n]) = 0$ para $0 \leq n \leq N-1$. En otras palabras, los datos reales representan un caso en el que se evitan aproximadamente la mitad de las operaciones aritméticas que se tienen que realizar.

Por lo tanto, cabe destacar que existen algoritmos más eficientes que los que hemos desarrollado hasta ahora, cuando las entradas son números reales. En concreto, en el libro [4] se desarrollan dos algoritmos más eficientes: el primero consiste en calcular dos FFTs reales de tamaño N calculando una FFT compleja de tamaño N, y el segundo nos permite calcular una FFT real de tamaño N calculando una FFT compleja de tamaño $\frac{N}{2}$.

Como también se explica en [4], con lo anterior se pueden implementar las transformadas rápidas del coseno y del seno.

La primera de estas transformadas se desarrolla a partir de la transformada discreta del coseno, que se define mediante la siguiente fórmula:

$$\widehat{C}(k) = \frac{f(0) + (-1)^k f(N)}{2} + \sum_{n=1}^{N-1} f(n) cos\left(\frac{\pi k n}{N}\right)$$

para $k = 0, 1, ..., N$.

Para aplicar este algoritmo necesitamos que la secuencia de datos, f, de la que queremos calcular la transformada, sea real y par.

La transformada rápida del seno, se desarrolla a partir de la transformada discreta del seno, que viene dada por la fórmula:

$$\widehat{S}(k) = \sum_{n=0}^{N-1} f(n) sin\left(\frac{\pi k n}{N}\right)$$

para $k = 1, 2, ..., N-1$.

Para poder aplicar dicho algoritmo se necesita que la entrada sea impar y con datos reales.

3 | Implementación del algoritmo DIF FFT de radio 2

En este capítulo vamos a ver como implementar el algoritmo trasformada de Fourier rápida con decimación en frecuencia (DIF FFT) y sus variantes. Para la trasformada de Fourier rápida con decimación en tiempo (DIT FFT) los cálculos se harían de forma análoga. Los contenidos de este capítulo hacen referencia al libro [4].

A lo largo del capítulo vamos a tomar $a :=$ array de dimensión uno y tamaño N en el cual se introducen los datos de entrada y, después del proceso, los de salida.

Antes de comenzar con la implementación de los algoritmos debemos de tener ciertos conocimientos previos que vamos a introducir a continuación.

Definición 3.1:

Decimos que los datos de entrada $f(0), f(1), \ldots, f(N-1)$ están en **orden natural** si $f(i)$ y $f(i+1)$ están situados de forma consecutiva en nuestro array (o lista) para $0 \leqslant i \leqslant N-2$.

Suponemos que los datos de entrada están colocados en a en orden natural. Sin embargo, el orden en el que aparecen los elementos de salida después de realizar el algoritmo DIF FFT no es obvío (como veremos más adelante).

En este algoritmo cada paso de subdivisión está definido recursivamente por la operación mariposa de Gentleman- Sande dada en la Definición 2.4.

El algoritmo iterativo por el cual se divide cada subproblema es el que se muestra a continuación, donde hemos supuesto calculados y almacenados los $\overline{w_N}^n$ que necesitamos y almacenados como se explica en el Anexo B.

Algoritmo 1 El algoritmo DIF FFT de radio 2 en pseudocódigo.

Require: $w[n] = \overline{w_N}^n$ precalculado y almacenado. $N = 2^n$
  NumOfProblems :=1
  ProblemSize := N
  while $ProblemSize > 1$ **do**
    HalfSize := ProblemSize/2
    for $K = 0$ **to** $K = NumOfProblems - 1$ **do**
      JFirst := K * ProblemSize
      JLast := JFirst + HalfSize -1
      Jtwiddle := 2 * (N - (N / NumOfProblems))
      for $J = JFisrt$ **to** $J = JLast$ **do**
        W:= w[Jtwiddle]
        Temp:=a[J]
        a[J]:= Temp + a[J + HalfSize]
        a[J + HalfSize]:= W * (Temp - a[J + HalfSize])
        Jtwiddle := Jtwiddle + 1
      end for
    end for
    NumOfProblems := 2 * NumOfProblems
    ProblemSize := HalfSize
  end while
  return a

La implementación de este algoritmo en Java se encuetra en el Anexo C.

Veamos un ejemplo concreto de aplicar el algoritmo anterior para una secuencia de entrada de tamaño $N = 8 = 2^3$. ,

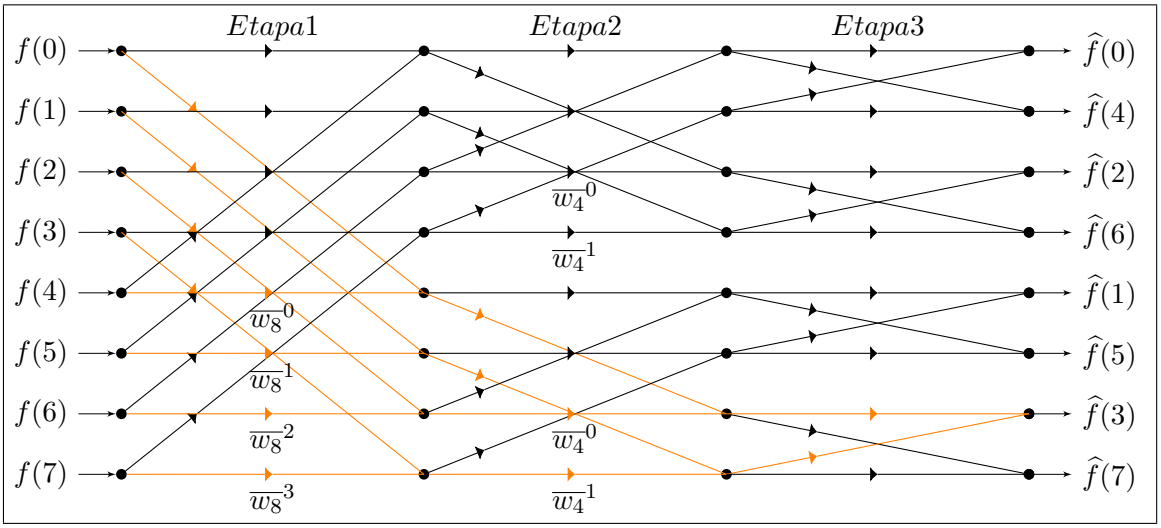


Figura 3.1: DIF FFT $N = 8$.

La Figura 3.1 muestra los valores iniciales contenidos en a, las tres etapas de la computación de la operación mariposa, el resultado de aplicar las transformaciones del algoritmo que veremos más adelante y resaltado en color naranja aparece la recursión realizada para calcular la salida $\widehat{f}(3)$.

Podemos observar que, en general, para una secuencia de entrada de $N = 2^n$ elementos, hay

exactamente $\frac{N}{2}$ mariposas en cada una de las $\log_2 N$ etapas en las que se divide el algoritmo.

Para enterder bien la salida que se muestra en la Figura 3.1 y, en general, en el Algoritmo 1, necesitamos conocer lo siguiente:

Definición 3.2:

Dado un número L entero positivo, se define su **representación binaria** como $L = (i_{n-1}, i_{n-2}, \ldots, i_1, i_0)$, donde

$$L = i_{n-1} \cdot 2^{n-1} + ... + i_1 \cdot 2^1 + i_0 \cdot 2^0$$

y $i_k \in \{0, 1\}$ $(k = 0, 1, \ldots, n-1)$.

Si consideramos $N = 2^n$ (como hemos supuesto para todo este capítulo y en el siguiente) y $0 \leq L \leq N-1$ entonces tenemos las siguientes propiedades:

P1. L es un número par $\Leftrightarrow i_0 = 0$

P2. L es un número impar $\Leftrightarrow i_0 = 1$

P3. $L < \frac{N}{2} \Leftrightarrow i_{n-1} = 0$

P4. $L > \frac{N}{2} \Leftrightarrow i_{n-1} = 1$

P5. Para $0 \leq L < M \leq N-1$, L y M se diferencian en el binario $i_k \Leftrightarrow M - L = 2^k$

Utilizando estas propiedades se puede descifrar fácilmente la salida del algoritmo.

Como sabemos en el Algoritmo 1, el valor inicial del tamaño del problema, la variable "ProblemSize" es $N = 2^n$ y la variable "HalfSize" toma los valores $2^{n-1}, 2^{n-2}, .., 2, 1$. Por tanto la distancia entre $a[J]$ y $a[J + HalfSize]$, donde tenemos que aplicar la operación mariposa, es una pontencia de dos. Así pues, aplicando la propiedad $P5$, el Algoritmo 1 quedaría de la siguiente forma:

Algoritmo 2 Algoritmo DIF FFT de radio 2 en términos binarios

```
k:= n-1
while k ≤ 0 do
  Gentleman-Sande a los pares de elementos
  cuya representación binaria se diferencia en el bit i_k
  k = k-1
end while
```

Como se muestra en el Algoritmo 2, el Algoritmo 1 se puede expresar en térmimos de direcciones binarias, por tanto vamos a introducir una notación binaria para los índices de a y para los subíndices de f.

Cuando escribamos $f_{i_2 i_1 i_0}$ nos referímos a f_k donde $i_2 i_1 i_0$ es la representación binaria de k. Y de la misma forma, escribir $a[i_2 i_1 i_0]$ es lo mismo que escribir $a(m)$ donde $i_2 i_1 i_0$ es la representación binaria de m.

Ahora bien, teniendo en cuenta esto, las tres etapas del algoritmo DIF FFT para $N = 8$ que se muestran en la Figura 3.1 con esta notación se describen por la secuencia

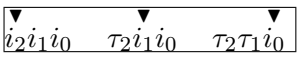


Figura 3.2: DIF FFT $N = 8$ notación binaria

donde en la figura Figura 3.2 $\overset{\blacktriangledown}{i_k}$ indica que la computación de la mariposa que implica a los pares de a que se diferencian en el bit i_k se está realizando en la etapa actual, y τ_k indica que la correspondiente operación mariposa ya se ha realizado en alguna etapa anterior.

El algoritmo finaliza cuando todas las operaciones mariposa de todas las etapas están realizadas.

Ahora sí, con esta notación veamos cual es el orden de la salida del algoritmo:

Definición 3.3:

Definimos la **permutación bit-reversal** como una permutación especial de una secuencia de N elementos, donde N es una potencia de dos. Concretamente, esta permutación se realiza indexando la sucesión con los números del 0 al $N-1$ y luego dándo la vuelta a la representación binaria de estos números. A cada elemento se le asigna entonces la nueva posición dada por este valor invertido.
Así, por ejemplo, si $n = (i_{n-1}i_{n-2}\dots i_1 i_0)_2$ es la posición inicial de a_m, entonces pasamos a colocar el valor a_m en la posición $\varphi(m) = (i_0 i_1, \dots, i_{n-1})_2$ de la lista.

Para realizar esta permutación, hemos implementado el código que se encuetra en el Anexo B, cuyo pseudocódigo es el siguiente:

Algoritmo 3 Un algoritmo bit reversal en pseudocódigo.

Require: Un vector de entrada a de tamaño $n = 2^k$.
a vector de los datos de entrada en orden natural.
R vector con los datos de salida en orden bit reversal.
powrev array tamaño k.
powrev[0]= $n/2$
for $i = 1$ **to** $i = k - 1$ **do**
  $powrev[i] = powrev[i-1]/2$
end for
for $m = 0$ **to** $m = n - 1$ **do**
  $j = 0$
  $mm = m$
  for $p = 0$ **to** $p = k - 1$ **do**
    $j = j + [mm]_2 * powrev[p]$
    $mm = mm/2$
  end for
  $R[m] = a[j]$
end for
return R

Resulta obvio comprobar que esta permutación repetida dos veces sobre sí misma, vuelve a la ordenación original de los elementos de la lista a la que se aplica.

Definición 3.4:

Decimos que los datos de entrada $f(0), f(1), ..., f(N-1)$ están en **orden bit-reversal** si se les ha aplicado una permutación bit-reversal estando en orden natural. De la misma forma decimos que los datos de salida $\widehat{f}(0), \widehat{f}(1), ..., \widehat{f}(N-1)$ están en **orden bit-reversal** si se les ha aplicado una permutación bit-reversal a su orden natural.

Ahora bien, nuestro objetivo es saber el elemento de salida $\widehat{f}$ que se encuentra en cada $a[i_2i_1i_0]$. Si interpretamos las ecuaciones (2,5) y (2,6) como implica la operación mariposa de la Definicion 2.2 la primera mitad del array a va a contener las salidas $\widehat{f}$ con índices pares y la segunda mitad del array a va a contener las salidas $\widehat{f}$ con índices impares.

Esto, después de la primera etapa, que se identifica con la operación $\overset{\blacktriangledown}{i_2}i_1i_0$, nos lleva a que a$[0i_1i_0]$ contiene la salida $\widehat{f}$ donde k tiene que ser par. Entonces, por las propiedades de los números binarios, vistas anteriormente el bit i_k más a la derecha debe ser 0. De igual forma para a$[1i_1i_0]$ tenemos que contiene la salida $\widehat{f}$, donde k tiene que ser impar, entonces el bit más a la derecha debe ser 1 . Esto implica que el bit más a la derecha de k para cada $\widehat{f}$ en a$[i_2i_1i_0]$ es i_2.

Ahora con los datos actualizados en a$[0i_1i_0]$ definimos un subproblema que se puede subdividir y con los datos actualizados en a$[1i_1i_0]$ también. Si aplicamos el mismo argumento que anteriormente a i_1 , i_0 tenemos que el bit más a la derecha de cada $\widehat{f}$ es i_1.

Repitiendo el mismo argumento una vez más en porciones sucesivamente a la mitad del array a, por lo que llegamos a la conclusión de que a$[i_2i_1i_0]$ finalmente contiene a la salida $\widehat{f}_{i_2i_1i_0}$, es decir, la salida $\widehat{f}$ está en orden bit-reversal respecto al orden de entrada.

Por ejemplo, en el Cuadro 3.1 para el caso $N = 8$ vemos las entradas y las salidas del algoritmo.

Entrada a :	f_{000}	f_{001}	f_{010}	f_{011}	f_{100}	f_{101}	f_{110}	f_{111}

Salida a :	$\widehat{f}_{000}$	$\widehat{f}_{100}$	$\widehat{f}_{010}$	$\widehat{f}_{110}$	$\widehat{f}_{001}$	$\widehat{f}_{101}$	$\widehat{f}_{011}$	$\widehat{f}_{111}$

Cuadro 3.1: La entrada en el array a con orden natural, y la salida en el array a con orden bit-reversal.

3.1. Variantes del algoritmo DIF-FFT radio 2

Técnicamente hablando el Algoritmo 1 depende de como se introduzcan los datos iniciales en el vector a.

En esta sección vamos a desarrollar dos variantes del algoritmo DIF FFT de radio 2.

Para ello antes debemos introducir la siguiente notación.

- $\boldsymbol{DIF_{NR}FFT}$: lo cúal hace referencia a que la entrada del algoritmo es en orden natural y la salida en orden bit-reversal (este algoritmo se ha desarrollado en el apartado anterior).

- $\boldsymbol{DIF_{RN}FFT}$: la entrada del algoritmo se realiza en orden bit-reversal y la salida se obtiene en orden natural.

- $\boldsymbol{DIF_{NN}FFT}$: la entrada y la salida de este algoritmo son en orden natural.

Para el algoritmo DIT FFT las tres variantes son las mismas y se desarrollan de forma análoga, es decir, las variantes son: $\boldsymbol{DIT_{NR}FFT}$, $\boldsymbol{DIT_{RN}FFT}$ y $\boldsymbol{DIT_{NN}FFT}$.

Poniendo como objetivo que la entrada y la salida del algoritmo sea en orden natural tenemos las posibilidades que aparecen en el siguiente esquema:

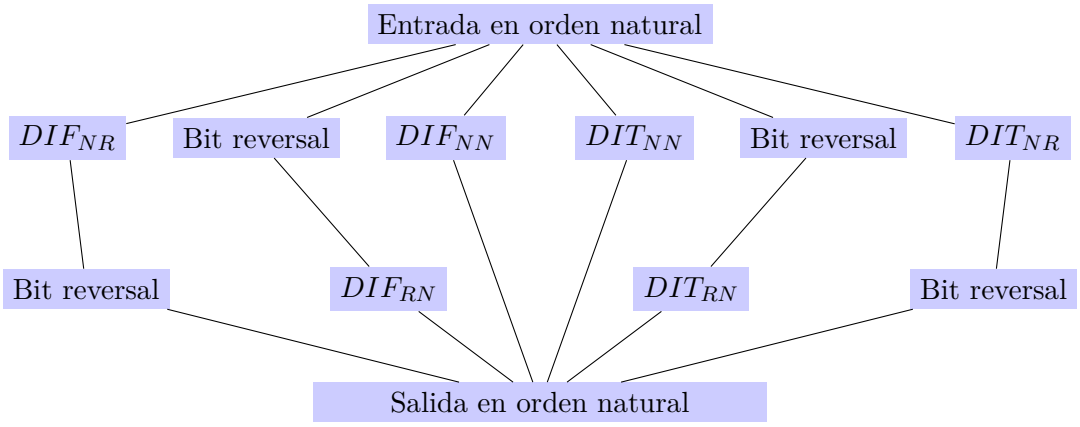


Figura 3.3: Combinaciones de un algoritmo bit-reversal y variantes DIF FFT y DIT FFT para que la entrada y la salida del algoritmo sean en orden natural.

3.1.1. Algoritmo FFT con entrada en orden bit-reversal y salida con orden natural

Para este algoritmo si denotamos $f_m^{(l)}$ como el resultado de aplicarle a f_m una operación mariposa en la etapa l, entonces aplicando el algoritmo Algoritmo 2 el espacio de a que contenía inicialmente a $f_{i_2 i_1 i_0}$, una vez terminada su ejecución va a contener al elemento $f^3_{i_2 i_1 i_0} = f_{i_0 i_1 i_2}$, es decir, si en $a[i_0 i_1 i_2]$ está contenido $f_{i_2 i_1 i_0}$ al final de la computación $a[i_0 i_1 i_2]$ va a contener a $f^3_{i_2 i_1 i_0}$.

Si los datos de entrada están en orden natural el algoritmo se lleva acabo aplicándole a los datos de entrada una permutación bit-reversal, con un algoritmo como el que podemos ver en el Anexo C antes de aplicar el método.

f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7

↓ **permutación bit-reversal**

f_0	f_4	f_2	f_6	f_1	f_5	f_3	f_7

↓ **Aplicando el algoritmo DIT_{RN}**

$\widehat{f_0}$	$\widehat{f_1}$	$\widehat{f_2}$	$\widehat{f_3}$	$\widehat{f_4}$	$\widehat{f_5}$	$\widehat{f_6}$	$\widehat{f_7}$

Ahora bien, suponiendo de nuevo que los $\overline{w_N}^n$ están ya calculados y almacenados para su uso, el algoritmo $\boldsymbol{DIF_{RN}\ FFT}$ es el siguiente:

Algoritmo 4 El algoritmo iterativo DIF FFT de radio 2 con entrada en orden bit-reversal y salida en orden natural en pseudocódigo.

Require: $N = 2^n$. $\overline{w_N}^k$ calculados y almacenados en w.

```
NumOfProblems :=1
ProblemSize := N
Distance := 1
while ProblemSize > 1 do
  for JFirst = 0 to JFirst = NumOfProblems − 1 do
    J := JFirst
    Jtwddle = 0
    while J > N − 1 do
      W:= w[Jtwiddle]
      Temp := a[J]
      a[J] := Temp + a[J + Distance]
      a[J + Distance] := (Temp - a[J + Distance]) * W
      Jtwiddle := Jtwiddle + 1
      J := J + 2 * NumOfProblems
    end while
  end for
  NumOfProblems := 2 * NumOfProblems
  ProblemSize := ProblemSize/2
  Distance := Distance * 2
end while
return a
```

El código implementado para este algoritmo en Java se encuentra en Anexo C. Veamos un ejemplo concreto de aplicación de este algoritmo para una secuencia de entrada de tamaño $N = 8 = 2^3$.

,

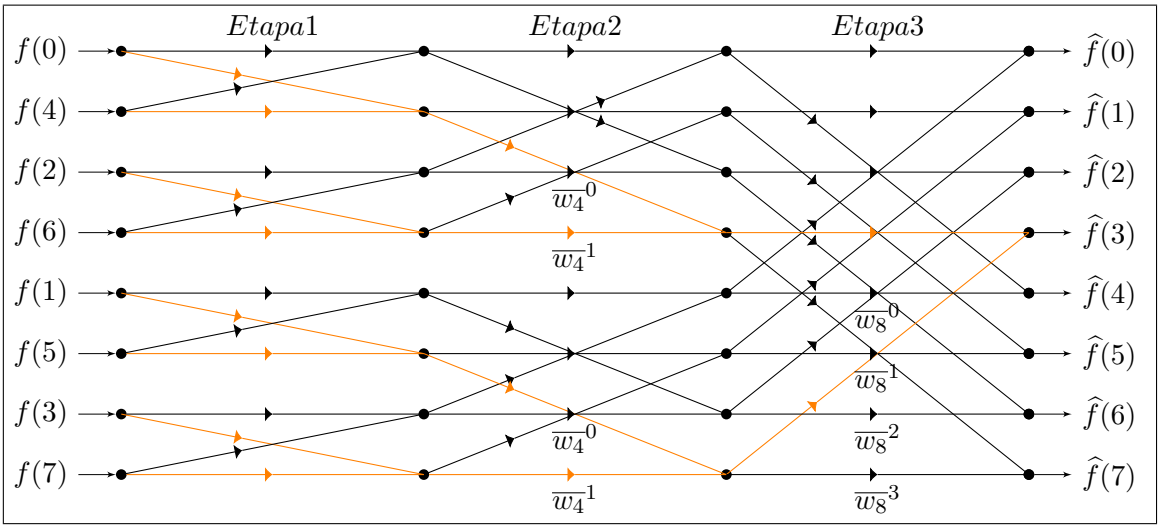


Figura 3.4: DIF_{RN} FFT $N = 8$

En el diagrama de la figura 3.4 representamos gráficamente los distintos cálculos realizados para hacer la FFT en el caso de $N = 8$, resaltando en color naranja la recursión hecha para calcular $\widehat{f}(3)$. Comparando la figura 3.4 con la figura $figura3{,}1$ podemos observar que los subproblemas que se definen con sus respectivos algoritmos son los mismos, aunque estén ordenados de forma distinta, debido al orden de los datos iniciales y las operaciones intermedias.

Como consecuencia podemos decir que los algoritmos $DIF_{NR}\ FFT$ y $DIF_{RN}\ FFT$ son esencialmente iguales, aunque los datos de entrada y de salida estén situados en distintos lugares.

3.1.2. Utilización de salidas de algoritmos FFT como entradas de FFT inversa

Como la transformada de Fourier discreta y la transformada de Fourier inversa tienen esencialmente la misma computación, salvo por el escalar $\frac{1}{N}$ y los w_N^n (en la inversa debemos de multplicar por el conjugado de $\overline{w_N}^n$ que es w_N^n). Entonces, con los algoritmos introducidos hasta ahora de la FFT podemos calcular también la IDFT. Por tanto, aunque los algoritmos DIF_{NR} y DIF_{RN} son esencialmente iguales es útil tener implementados los dos y la inversa de cada uno de ellos (lo cuál sólo supone dos cambios triviales en cada uno de los algoritmos). Ya que al combinar uno de los métodos (NR o RN) con la inversa del contrario (RN o NR) podemos recuperar la señal inicial. Además, con esto podemos comprobar que los algoritmos funcionan correctamente. Es decir, como se muestra en el siguiente esquema, si introducimos una entrada en orden natural, aplicamos un algoritmo NR y a la salida le aplicamos un algoritmo RN inverso obtenemos de nuevo la entrada inicial.

f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7

↓ **FFT** *usando* DIF_{NR}

$\widehat{f_0}$	$\widehat{f_4}$	$\widehat{f_2}$	$\widehat{f_6}$	$\widehat{f_1}$	$\widehat{f_5}$	$\widehat{f_3}$	$\widehat{f_7}$

↓ **FFT inversa** *usando* DIF_{RN}

f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7

De la misma forma, si la entrada de los datos está en orden bit-reversal, podemos aplicar un algoritmo RN y seguidamente uno NR inverso para obtener los datos iniciales en su corresponciente orden.

En cualquier caso, con una combinación apropiada de algoritmos RN y NR podemos recuperar la señal inicial y así comprobar que funcionan correctamente los algoritmos, por tanto, es útil tener los dos métodos (NR y RN) implementados.

3.1.3. Algoritmo FFT con entrada en orden natural y salida con orden natural

Como hemos visto en secciones anteriores, si los datos de entrada están en orden natural la salida será en orden bit-reversal y viceversa. Si tenemos como objetivo que la entrada y la salida del algoritmo sean en orden natural con lo visto hasta ahora tenemos que aplicar una permutación bit-reversal, en esta sección vamos a ver como eliminar el coste de memoria extra de la permutación bit-reversal combinando la permutación de datos con las operaciones mariposa en cada paso.

Definición 3.5:

Cuando la entrada y la salida de un algoritmo están en orden natural, tal algoritmo recibe el nombre de **algoritmo ordenado**.

La clave para entender el Algoritmo 5 es ver cada operación mariposa como un paso de permutación seguido del cálculo de la mariposa. Estos pasos de permutación ordenan tanto la entrada como la salida de cada subproblema.

Con la notación introducida anteriormente en términos binarios. Consideramos $N = 8$. Veamos como será el algoritmo:

Como la entrada del algoritmo es en orden natural, $a[i_2 i_1 i_0] = f_{i_2 i_1 i_0}$. La primera operación mariposa será $i_2 \overset{\blacktriangledown}{i_1} i_0$. Esta operación mariposa está precedida de una permutación de datos de $a[i_2 i_1 i_0]$ a $b[i_1 i_0 i_2]$. Esto lo podemos escribir

$$\underset{a}{i_2 i_1 i_0} \qquad \underset{b}{i_1 i_0 \overset{\blacktriangledown}{i_2}}$$

En el segundo paso se realiza la permutación de $f^{(1)}_{i_2 i_1 i_0}$ de $b[i_1 i_0 i_2]$ a $a[i_0 i_1 i_2]$ donde se calcula la operación mariposa correspondiente que con la notación que estamos siguiendo

$$\underset{b}{i_1 i_0 \tau_2} \qquad \underset{a}{i_0 \tau_2 \overset{\blacktriangledown}{i_1}}.$$

Por último, se realiza la permutación de $f^{(2)}_{i_2 i_1 i_0}$ de $a[i_0 i_2 i_1]$ a $b[i_2 i_1 i_0]$ donde se calcula la operación mariposa correspondiente, con la notación seguida.

$$\underset{b}{i_0 \tau_2 \tau_1} \qquad \underset{a}{\tau_2 \tau_1 \overset{\blacktriangledown}{i_0}}$$

Por tanto $f^{(3)} = \widehat{f}_{i_0 i_1 i_2}$, lo que indica que la salida y la entrada del algoritmo son en orden natural.

El algoritmo ordenado con el que llevamos acabo la FFT es el siguiente:

Algoritmo 5 El algoritmo iterativo DIF FFT de radio 2 con entrada en orden natural y salida en orden natural en pseudocódigo.

Require: w$[n] = \overline{w_N}^n$ precalculado y almacenado. La talla inicial del problema debe ser $N = 2^n$.

```
NumOfProblems :=1
ProblemSize := N
HalfSize = ProblemSize/2
Distance := 1
NotSwitchInput := true
while ProblemSize > 1 do
  if NotSwitchInput then
    for JFirst = 0 hasta K = NumOfProblems − 1 do
      J := JFirst
      K:= JFirst
      Jtwiddle := 0
      while J < N − 1 do
        W:= w[Jtwiddle]
        b[J]:= a[k] + a[K + N/2]
        b[J + Distance] := (a[K] - a[K + N/2])*W
        Jtwiddle := Jtwiddle + NumOfProblems
        J := J + 2 * NumOfProblems
        K := K + NumOfProblems
      end while
    end for
    NotSwitchInput := false
  else
    for JFirst = 0 hasta K = NumOfProblems − 1 do
      J := JFirst
      K:= JFirst
      Jtwiddle := 0
      while J < N − 1 do
        W:= w[Jtwiddle]
        a[J]:= b[K] + b[K + N/2]
        a[J + Distance] := (b[K] - b[K + N/2])*W
        Jtwiddle := Jtwiddle + NumOfProblems
        J := J + 2 * NumOfProblems
        K := K + NumOfProblems
      end while
    end for
  end if
  NumOfProblems := 2 * NumOfProblems
  ProblemSize := ProblemSize/2
  Distance := Distance * 2
end while
return  a ó b (según sean N)
```

Veamos un ejemplo concreto de aplicación de este algoritmo para una secuencia de entrada de tamaño $N = 8 = 2^3$.

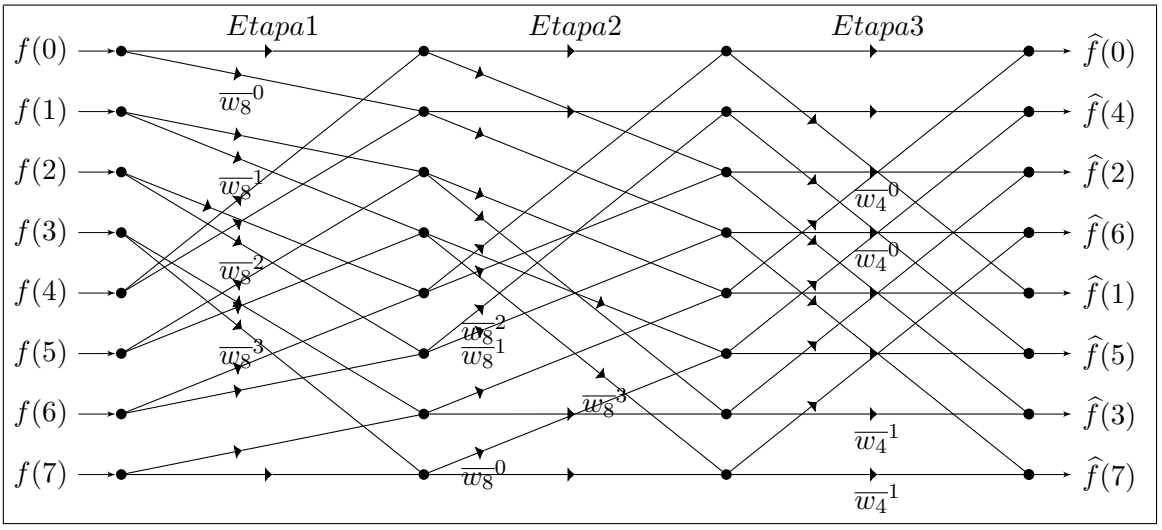


Figura 3.5: $DIF_{NN}FFT$, $N = 8$

Comparando *figura*3,5, *figura*3,4 y *figura*3,1 observamos que las tres variantes del algoritmo DIF FFT tienen exactamente el mismo número de subproblemas, durante las diferentes etapas de la computación, ordenados de forma diferente. Por tanto, las tres variantes del algoritmo $DIFFFT$ de radio 2, son esencialmente iguales.

4 | FFT radio 4 y FFT radio split

4.1. Transformada de Fourier rápida de radio 4

En este capítulo desarrollamos la transformada de Fourier rápida de radio 4, para ello consideramos muestras discretas de tamaño $N = 4^n = 2^{2n}$.

La razón por la que merece la pena desarrollar la FFT de radio 4, en lugar de usar simplemente la FFT de radio 2, es que el coste arimético puede reducirse considerablemente. Los contenidos de este capítulo hacen referencia al libro [4].

4.1.1. DIT FFT radio 4

Este algoritmo está basado en dividir la expresión (2.1), que define la transformada de Fourier discreta de una serie compleja, en cuatro sumas parciales.

$$\begin{aligned}
\widehat{f}(k) &= \textstyle\sum_{n=0}^{N-1} f(n) e^{\frac{-2i\pi kn}{N}} \\
&= \textstyle\sum_{n=0}^{\frac{N}{4}-1} f(4n)\overline{w_N}^{k(4n)} + \sum_{n=0}^{\frac{N}{4}-1} f(4n+1)\overline{w_N}^{k(4n+1)} \\
&\quad + \textstyle\sum_{n=0}^{\frac{N}{4}-1} f(4n+2)\overline{w_N}^{k(4n+2)} + \sum_{n=0}^{\frac{N}{4}-1} f(4n+3)\overline{w_N}^{k(4n+3)} \\
&= \textstyle\sum_{n=0}^{\frac{N}{4}-1} f(4n)\overline{w_N}^{k(4n)} + \overline{w_N}^{k} \sum_{n=0}^{\frac{N}{4}-1} f(4n+1)\overline{w_N}^{(4n)} \\
&\quad + \overline{w_N}^{2k} \textstyle\sum_{n=0}^{\frac{N}{4}-1} f(4n+2)\overline{w_N}^{(4n)} + \overline{w_N}^{3k} \sum_{n=0}^{\frac{N}{4}-1} f(4n+3)\overline{w_N}^{(4n)}.
\end{aligned}$$

Ahora bien, consideramos estos cuatro conjuntos:

$$\begin{aligned}
&\left\{g(k) : g(k) = f(4k), 0 \leq k \leq \tfrac{N}{4} - 1\right\} \\
&\left\{h(k) : h(k) = f(4k+1), 0 \leq k \leq \tfrac{N}{4} - 1\right\} \\
&\left\{p(k) : p(k) = f(4k+2), 0 \leq k \leq \tfrac{N}{4} - 1\right\} \\
&\left\{q(k) : q(k) = f(4k+3), 0 \leq k \leq \tfrac{N}{4} - 1\right\}
\end{aligned}$$

y teniendo en cuenta que:

$$\overline{w_N}^4 = e^{\frac{-2i\pi 4}{N}} = e^{\frac{-2i\pi}{\frac{N}{4}}} = \overline{w_{\frac{N}{4}}}, \tag{4.1}$$

pasamos de definir los subproblemas de tamaño cuatro.

$$\widehat{g}(k) = \sum_{n=0}^{\frac{N}{4}-1} f(4n)\overline{w_N}^{k(4n)} = \sum_{n=0}^{\frac{N}{4}-1} f(4n)\overline{w_N}^{4(kn)} = \sum_{n=0}^{\frac{N}{4}-1} f(4n)\overline{w_{\frac{N}{4}}}^{kn}. \tag{4.2}$$

$$\widehat{h}(k) = \sum_{n=0}^{\frac{N}{4}-1} f(4n+1)\overline{w_N}^{k(4n)} = \sum_{n=0}^{\frac{N}{4}-1} f(4n+1)\overline{w_N}^{4(kn)} = \sum_{n=0}^{\frac{N}{4}-1} f(4n+1)\overline{w_{\frac{N}{4}}}^{kn}. \tag{4.3}$$

$$\widehat{p}(k) = \sum_{n=0}^{\frac{N}{4}-1} f(4n+2)\overline{w_N}^{k(4n)} = \sum_{n=0}^{\frac{N}{4}-1} f(4n+2)\overline{w_N}^{4(kn)} = \sum_{n=0}^{\frac{N}{4}-1} f(4n+2)\overline{w_{\frac{N}{4}}}^{kn}. \tag{4.4}$$

$$\widehat{q}(k) = \sum_{n=0}^{\frac{N}{4}-1} f(4n+3)\overline{w_N}^{k(4n)} = \sum_{n=0}^{\frac{N}{4}-1} f(4n+3)\overline{w_N}^{4(kn)} = \sum_{n=0}^{\frac{N}{4}-1} f(4n+3)\overline{w_{\frac{N}{4}}}^{kn}. \tag{4.5}$$

Cada uno de ellos para $k = 0, 1, ..., \frac{N}{4} - 1$.

Por tanto para obtener la solución del problema inicial debemos resolver estos cuatro subproblemas. Como $\widehat{g}(k)$ tiene periodo $\frac{N}{4}$ tenemos que $\widehat{g}(k) = \widehat{g}(k + \frac{N}{4}) = \widehat{g}(k + \frac{N}{2}) = \widehat{g}(k + \frac{3N}{4})$ y esto mismo ocurre para $\widehat{h}(k)$, $\widehat{p}(k)$ y $\widehat{q}(k)$. Vamos a expresar la salida $\widehat{f}(k)$ en términos de $\widehat{g}(k)$, $\widehat{p}(k)$ y $\widehat{q}(k)$ para $k = 0, 1, ..., \frac{N}{4} - 1$.

Para ello debemos de tener en cuenta las igualdades: $\overline{w_N}^{\frac{N}{4}} = -i$, $\overline{w_N}^{\frac{N}{2}} = -1$, $\overline{w_N}^{\frac{3N}{4}} = i$, $\overline{w_N}^{N} = 1$, $\overline{w_N}^{\frac{3N}{2}} = 1$ y $\overline{w_N}^{\frac{9N}{4}} = 1$.

$$\widehat{f}(k) = \widehat{g}(k) + \overline{w_N}^{k}\widehat{h}(k) + \overline{w_N}^{2k}\widehat{p}(k) + \overline{w_N}^{3k}\widehat{q}(k). \tag{4.6}$$

$$\begin{aligned}\widehat{f}\left(k + \frac{N}{4}\right) &= \widehat{g}(k) + \overline{w_N}^{k+\frac{N}{4}}\widehat{h}(k) + \overline{w_N}^{2\left(k+\frac{N}{4}\right)}\widehat{p}(k) + \overline{w_N}^{3\left(k+\frac{N}{4}\right)}\widehat{q}(k) = \\ &= \widehat{g}(k) - i\overline{w_N}^{k}\widehat{h}(k) - \overline{w_N}^{2k}\widehat{p}(k) + i\overline{w_N}^{3k}\widehat{q}(k).\end{aligned} \tag{4.7}$$

$$\begin{aligned}\widehat{f}\left(k + \frac{N}{2}\right) &= \widehat{g}(k) + \overline{w_N}^{k+\frac{N}{2}}\widehat{h}(k) + \overline{w_N}^{2\left(k+\frac{N}{2}\right)}\widehat{p}(k) + \overline{w_N}^{3\left(k+\frac{N}{2}\right)}\widehat{q}(k) = \\ &= \widehat{g}(k) - \overline{w_N}^{k}\widehat{h}(k) + \overline{w_N}^{2k}\widehat{p}(k) - \overline{w_N}^{3k}\widehat{q}(k).\end{aligned} \tag{4.8}$$

$$\begin{aligned}\widehat{f}\left(k + \frac{3N}{4}\right) &= \widehat{g}(k) + \overline{w_N}^{k+\frac{3N}{4}}\widehat{h}(k) + \overline{w_N}^{2\left(k+\frac{3N}{4}\right)}\widehat{p}(k) + \overline{w_N}^{3\left(k+\frac{3N}{4}\right)}\widehat{q}(k) = \\ &= \widehat{g}(k) + i\overline{w_N}^{k}\widehat{h}(k) - \overline{w_N}^{2k}\widehat{p}(k) - i\overline{w_N}^{3k}\widehat{q}(k).\end{aligned} \tag{4.9}$$

Todas estas ecuaciones para $k = 0, 1, ..., \frac{N}{4} - 1$.

Si implementamos el algoritmo de radio 4 basándonos en las euaciones (4,6), (4,7), (4,8) y (4,9) un paso de tal algoritmo va a requerir más operaciones que dos pasos del algoritmo de radio 2, ya que algunos resultados parciales eran calculados más de una vez. Sin embargo, tales resultados parciales pueden ser identificados y calculados una vez y así un paso del algoritmo de radio 4 podrá requerir menos operaciones aritméticas que dos pasos del algoritmo de radio 2, lo que supondrá que el coste total del algoritmo de radio 4 sea menor que el coste total del algormito de radio 2, que es nuestro objetivo.

Reescribiendo las ecuaciones anteriores, tenemos:

$$\widehat{f}\,(k) = \left(\widehat{g}(k) + \overline{w_N}^{2k}\widehat{p}(k)\right) + \left(\overline{w_N}^{k}\widehat{h}(k) + \overline{w_N}^{3k}\widehat{q}(k)\right). \tag{4.10}$$

$$\widehat{f}\left(k + \frac{N}{4}\right) = \left(\widehat{g}(k) - \overline{w_N}^{2k}\widehat{p}(k)\right) - i\left(\overline{w_N}^{k}\widehat{h}(k) - \overline{w_N}^{3k}\widehat{q}(k)\right). \tag{4.11}$$

$$\widehat{f}\left(k + \frac{N}{2}\right) = -\left(\widehat{g}(k) + \overline{w_N}^{2k}\widehat{p}(k)\right) + \left(\overline{w_N}^{k}\widehat{h}(k) + \overline{w_N}^{3k}\widehat{q}(k)\right). \tag{4.12}$$

$$\widehat{f}\left(k + \frac{3N}{4}\right) = i\left(\widehat{g}(k) - \overline{w_N}^{2k}\widehat{p}(k)\right) + \left(\overline{w_N}^{k}\widehat{h}(k) - \overline{w_N}^{3k}\widehat{q}(k)\right). \tag{4.13}$$

El cálculo de las ecuaciones (4,10), (4,11), (4,12) y (4,13) puede ser representado en dos etapas por el cáculo de la siguiente operación mariposa:

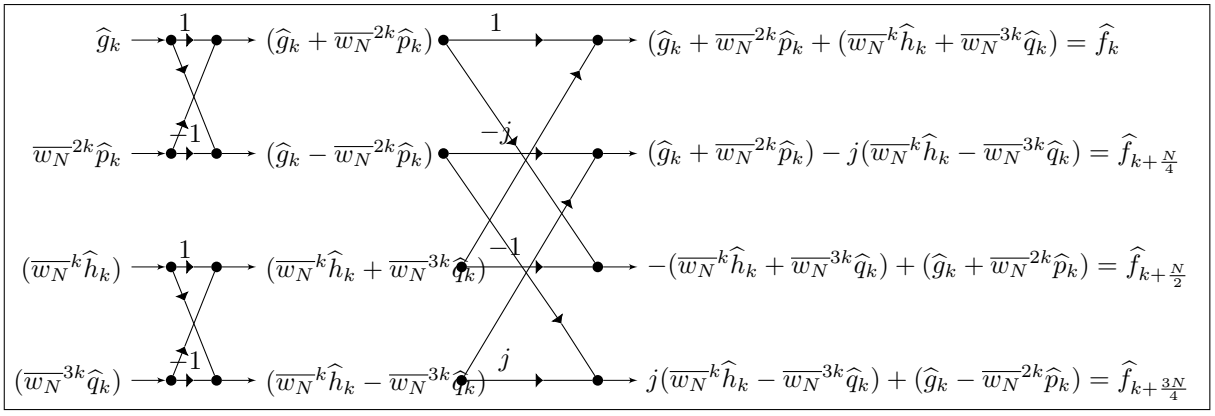


Figura 4.1: Operación mariposa, decimación en tiempo, FFT radio 4

Dado un vector de tamaño $n = 4^m$, será necesario realizar m etapas, que aparecen al ir descomponiendo el cálculo inicial primero en el cálculo de 4 transformadas de tamaño 4^{m-1}, luego cada uno de los subproblemas dará lugar a otros 4, de tamaño 4^{m-2} (con lo cual tendremos $4 = 2^2$ subproblemas de tamaño 4^{m-2}) etc..., hasta la fase m-ésima.

La operación mariposa que acabamos de definir, tiene dos pasos, que podemos interpretar como dos operaciones mariposa de tamaño 2. Si el tamaño del vector inicial es $n = 4^n = 2^{2n}$ y aplicamos estas operaciones para resolver el problema inicial, necesitamos m estapas, mientras que si aplicamos la correspondiente mariposa de un algoritmo de radio dos necesitamos 2m etapas.

Para calcular cada subproblema de tamaño $\frac{N}{4}$ necesitamos $\frac{3N}{4}$ multiplicaciones complejas y N sumas complejas en la primera etapa de la mariposa, en la segunda etapa de la mariposa necesitamos N sumas complejas. Esto implica que necesitamos $\frac{3N}{4}$ multimplicaciones complejas y $2N$ sumas complejas en cada operación mariposa.

Ahora bien, como el coste computacional de los algoritmos se mide en el número de operaciones reales, si recordamos que cada suma compleja requiere dos sumas reales y que cada multiplicación compleja requiere tres multiplicaciones reales y tres sumas reales, entonces necesitamos:

- $\frac{9N}{4}$ multiplicaciones reales.
- $4N + \frac{9N}{4} = \frac{16N+9N}{4} = \frac{25N}{4}$ sumas reales.

Por lo que un paso de este algoritmo, en términos de flops requiere $\frac{9N}{4} + \frac{25N}{4} = \frac{34N}{4} = \frac{17N}{2}$ flops.

Como el objetivo del desarrollo del algoritmo de radio 4 es minimizar las operaciones reales, vamos a excluir las multiplicaciones triviales, estas son , $\overline{w_N}^0 = 1$ y $\overline{w_4}^l = (-i)^n = \pm 1$ ó $\pm i$.

Además, podemos observar que el coste de una multiplicación de una raíz de la unidad que es potencia impar de $\overline{w_8} = \frac{(1-i)}{\sqrt{2}}$, es de menor coste que una multiplicación compleja ya que

$$\overline{w_8}^{2n+1} = \overline{w_4}^n \cdot \overline{w_8} = (-i)^n \left(\frac{1-i}{\sqrt{2}}\right) = \pm\left(\frac{1-i}{\sqrt{2}}\right) \text{ ó } \pm\left(\frac{1+i}{\sqrt{2}}\right).$$

Estos factores especiales son identificados para el cálculo de $\overline{w_N}^{2k}\widehat{p}(k)$, $\overline{w_N}^k\widehat{h}(k)$ y $\overline{w_N}^{3k}\widehat{q}(k)$ para $k = 0, 1, ..., \frac{N}{4} - 1$.

Ahora bien, estos casos especiales para las multiplicaciones de las raíces de la unidad en el algoritmo de radio 4 los podemos identificar de la siguiente forma:

$0 \le k \le \frac{N}{4} - 1$	$k = 0$	$k = \left(\frac{1}{2}\right)\frac{N}{4}$	$k = \left(\frac{1}{4}\right)\frac{N}{4}$	$k = \left(\frac{3}{4}\right)\frac{N}{4}$
$\overline{w_N}^{2k}\widehat{p}(k)$	$1 \cdot \widehat{p}(k) = \widehat{p}(k)$	$\overline{w_4}\widehat{p}(k) = -i\widehat{p}(k)$	$\overline{w_8}\widehat{p}(k)$	$\overline{w_8}^3\widehat{p}(w)$
$\overline{w_N}^k$	$1 \cdot \widehat{h}(k) = \widehat{h}(k)$	$\overline{w_8}\widehat{h}(k)$	x	x
$\overline{w_N}^{3k}\widehat{q}(k)$	$1 \cdot \widehat{q}(k) = \widehat{q}(k)$	$\overline{w_8}^3\widehat{q}(k)$	x	x

observando la tabla anterior vemos que hay 8 casos especiales:

- Cuatro de ellos tiene una multiplicación por 1 y (-i).
- Cuatro de ellos tienen una mutiplicación por una pontencia impar de $\overline{w_8}$.

El total de multiplicaciones complejas no triviales se reduce entonces a $\frac{3N}{4} - 8$. Por lo que como sólo necesitamos 4 flops para el cálculo de cada $\overline{w_8}\widehat{p}(k)$, $\overline{w_8}^3\widehat{p}(k)$, $\overline{w_8}\widehat{h}(k)$ y $\overline{w_8}^3\widehat{h}(k)$.

Por tanto, el total de operaciones reales se reduce a :

$$(3+3) \cdot \left(\frac{3N}{4} - 8\right) + 4 \cdot 4 + 2 \cdot (2N) = \frac{17N}{2} - 32.$$

Estos factores especiales aparecen en cada uno de los pasos del algoritmo, entonces los ahorros pueden ser incorporados a las ecuaciones de recurrencia. Para ello, asumiendo que el tamaño del problema es $N = 4^n$, estos factores especiales son identificados por $k = 0, 1, ..., \frac{N}{4^{j+1}}$ en el j-ésimo paso para $j = 0, 1, ..., n-2$ como vemos acontinuación.

$0 \leq k \leq \frac{N}{4^{j+1}} - 1$	$k = 0$	$k = \left(\frac{1}{2}\right)\frac{N}{4^{j+1}}$	$k = \left(\frac{1}{4}\right)\frac{N}{4^{j+1}}$	$k = \left(\frac{3}{4}\right)\frac{N}{4^{j+1}}$
$\overline{w_{\frac{N}{4^j}}}^{2k}\widehat{p}(k)$	$1 \cdot \widehat{p}(k) = \widehat{p}(k)$	$\overline{w_4}\widehat{p}(k) = -i\widehat{p}(k)$	$\overline{w_8}\widehat{p}(k)$	$\overline{w_8}^3\widehat{p}(w)$
$\overline{w_{\frac{N}{4^j}}}^{k}\widehat{p}(k)$	$1 \cdot \widehat{h}(k) = \widehat{h}(k)$	$\overline{w_8}\widehat{h}(k)$	x	x
$\overline{w_{\frac{N}{4^j}}}^{3k}\widehat{q}(k)$	$1 \cdot \widehat{q}(k) = \widehat{q}(k)$	$\overline{w_8}^3\widehat{q}(k)$	x	x

Con todo esto podemos calcular el coste computacional del algoritmo.

Coste computacinal del algoritmo:
El coste del algoritmo DIT FFT de radio 4 puede ser representado por la siguiente ecuación de recurrencia como se explica en Anexo A .

$$S(N) = \begin{cases} 4S(\frac{N}{4}) + \frac{17N}{2} - 32 & \text{si } N = 4^n \geqslant 4, \\ 16 & \text{si } N = 4 \end{cases}$$

De esto se sigue que:

$$\begin{aligned} S(N) &= S(4^m) \\ &= 4S(4^{m-1}) + \tfrac{17}{2} \cdot 4^m - 32 \\ &= 4^2 \cdot S(4^{m-2}) - 4 \cdot 32 - 32 + 2 \cdot \tfrac{17}{2} 4^m \\ &= 4^2[4 \cdot S(4^{m-3}) + \tfrac{17}{2} \cdot 4^{m-2} - 32] - 4 \cdot 32 - 32 + 2 \cdot \tfrac{17}{2} \\ &= 4^3 S(2^{m-3}) - 4^2 \cdot 32 - 4 \cdot 32 - 32 + 3 \cdot \tfrac{17}{2} \cdot 4^m \\ &\vdots \\ &\vdots \\ &= 4^{m-1} S(4) - \textstyle\sum_{i=0}^{m-2} 4^i \cdot 32 + (m-1) \cdot \frac{17}{2} \cdot 4^m \\ &= 4^{m-1} \cdot 16 - 32 \cdot \textstyle\sum_{i=0}^{m-2} 4^i + (m-1) \cdot \frac{17}{2} \cdot 4^m \\ &= 4^{m+1} - 32 \cdot \textstyle\sum_{i=0}^{m-2} 4^i + (m-1) \cdot \frac{17}{2} \cdot 4^m \\ &= N \cdot 4 + \frac{32\left(1 - \frac{N}{4}\right)}{3} + \frac{17 \cdot N(\log_4 N)}{2} \\ &= \tfrac{17N \log_2 N}{4} - \tfrac{43}{6} N + \tfrac{32}{3}. \end{aligned}$$

Ahora bien, si comparamos el coste del algoritmo de radio 2, que era $T(N) = 5N \log_2 N$, con el coste del agoritmo de radio 4 que acabamos de calcular tenemos:

$$\lim_{N \to \infty} \frac{S(N)}{T(N)} = \frac{\frac{17}{4}}{5} = \frac{17}{20},$$

y de esto último se sigue que hemos conseguido ahorrar un 15 %, ya que el algotirmo de radio 4 tiene como coste un 85 % del coste del algoritmo de radio 2.

4.1.2. DIF FFT radio 4

Este algoritmo está basado de nuevo en dividir la expresión (2.1) en cuatro sumas parciales de la siguiente forma:

$$\begin{aligned}
\widehat{f}(k) &= \sum_{n=0}^{N-1} f(n)\overline{w_N}^{kn} \\
&= \sum_{n=0}^{\frac{N}{4}-1} f(n)\overline{w_N}^{kn} + \sum_{n=\frac{N}{4}}^{\frac{N}{2}-1} f(n)\overline{w_N}^{kn} + \sum_{n=\frac{N}{2}}^{\frac{3N}{4}-1} f(n)\overline{w_N}^{kn} + \sum_{n=\frac{3N}{4}}^{N-1} f(n)\overline{w_N}^{kn} \\
&= \sum_{n=0}^{\frac{N}{4}-1} f(n)\overline{w_N}^{kn} + \overline{w_N}^{k\left(\frac{N}{4}\right)} \sum_{n=0}^{\frac{N}{4}-1} f\left(n+\frac{N}{4}\right)\overline{w_N}^{kn} \\
&\quad + \overline{w_N}^{k\left(\frac{N}{2}\right)} \sum_{n=0}^{\frac{N}{4}-1} f\left(n+\frac{N}{2}\right)\overline{w_N}^{kn} + \overline{w_N}^{k\left(\frac{3N}{4}\right)} \sum_{n=0}^{\frac{N}{4}-1} f\left(n+\frac{3N}{4}\right)\overline{w_N}^{kn} \\
&= \sum_{n=0}^{\frac{N}{4}-1} f(n)\overline{w_N}^{kn} + \overline{w_4}^{k} \sum_{n=0}^{\frac{N}{4}-1} f\left(n+\frac{N}{4}\right)\overline{w_N}^{kn} + \overline{w_4}^{2k} \sum_{n=0}^{\frac{N}{4}-1} f\left(n+\frac{N}{2}\right)\overline{w_N}^{kn} \\
&\quad + \overline{w_4}^{3k} \sum_{n=0}^{\frac{N}{4}-1} f\left(n+\frac{3N}{4}\right)\overline{w_N}^{kn} \\
&= \sum_{n=0}^{\frac{N}{4}-1} \left(f(n) + \overline{w_4}^{k} f\left(n+\frac{N}{4}\right) + f\left(n+\frac{N}{2}\right)\overline{w_4}^{2k} + f\left(n+\frac{3N}{4}\right)\overline{w_4}^{3k}\right)\overline{w_N}^{kn},
\end{aligned}$$

donde en las igualdades anteriores hemos tenido en cuenta que $\overline{w_N}^{k\frac{N}{4}} = \overline{w_4}^{k}$, $\overline{w_N}^{k\frac{N}{2}} = \overline{w_4}^{k}$ y $\overline{w_N}^{k\frac{3N}{4}} = \overline{w_4}^{3k}$.

Ahora definimos los cuatro subproblemas:

$$\begin{aligned}
\widehat{g}(k) &= \widehat{f}(4k) \\
&= \sum_{n=0}^{\frac{N}{4}-1} \left(f(n) + \overline{w_4}^{4k} f\left(n+\frac{N}{4}\right) + f\left(n+\frac{N}{2}\right)\overline{w_4}^{2(4k)} + f\left(n+\frac{3N}{4}\right)\overline{w_4}^{3(4k)}\right)\overline{w_N}^{4kn}.
\end{aligned}$$

Usando que $\overline{w_4}^{k\cdot 4} = 1$, $\overline{w_4}^{2\cdot(4k)} = 1$, $\overline{w_4}^{3\cdot(4k)} = 1$ y que $\overline{w_{\frac{N}{4}}}^{4kn} = 1$.

$$\begin{aligned}
\widehat{g}(k) &= \sum_{n=0}^{\frac{N}{4}-1} \left(f(n) + f\left(n+\frac{N}{4}\right) + f\left(n+\frac{N}{2}\right) + f\left(n+\frac{3N}{4}\right)\right)\overline{w_N}^{4kn} \\
&= \sum_{n=0}^{\frac{N}{4}-1} \left(f(n) + f\left(n+\frac{N}{2}\right) + f\left(n+\frac{N}{4}\right) + f\left(n+\frac{3N}{4}\right)\right)\overline{w_{\frac{N}{4}}}^{nk} \\
&= \sum_{n=0}^{\frac{N}{4}-1} g(n)\overline{w_{\frac{N}{4}}}^{nk},
\end{aligned}$$

para $k = 0, 1, ..., \frac{N}{4} - 1$ y $g(n) = \left(f(n) + f\left(n+\frac{N}{2}\right) + f\left(n+\frac{N}{4}\right) + f\left(n+\frac{3N}{4}\right)\right)$.

$$\begin{aligned}
\widehat{h}(k) &= \hat{f}(4k+1) \\
&= \sum_{n=0}^{\frac{N}{4}-1} \left(f(n) + f\left(n+\frac{N}{4}\right)\overline{w_4}^{4k+1} + f\left(n+\frac{N}{2}\right)\overline{w_4}^{2(4k+1)} + f\left(n+\frac{3N}{4}\right)\overline{w_4}^{3(4k+1)}\right)\overline{w_N}^{n(4k+1)} \\
&= \sum_{n=0}^{\frac{N}{4}-1} \left(f(n) + f\left(n+\frac{N}{4}\right)\overline{w_4}^{4k+1} + f\left(n+\frac{N}{2}\right)\overline{w_4}^{2(4k+1)} + f\left(n+\frac{3N}{4}\right)\overline{w_4}^{3(4k+1)}\right)\overline{w_N}^{n}\overline{w_N}^{4kn}.
\end{aligned}$$

Usando que $\overline{w_4}^{(4k+1)} = -i$, $\overline{w_4}^{2(4k+1)} = -1$ y $\overline{w_4}^{3(4k+1)} = i$,

$$\begin{aligned}
\widehat{h}(k) &= \sum_{n=0}^{\frac{N}{4}-1} \left(f(n) - f\left(n+\frac{N}{2}\right)\right) - i\left(f\left(n+\frac{N}{4}\right) - f\left(n+\frac{3N}{4}\right)\right)\overline{w_N}^{n}\overline{w_{\frac{N}{4}}}^{kn} \\
&= \sum_{n=0}^{\frac{N}{4}-1} h(n)\overline{w_{\frac{N}{4}}}^{kn},
\end{aligned}$$

para $k = 0, 1, .., \frac{N}{4} - 1$ y $h(n) = \left(\left(f(n) - f\left(n+\frac{N}{2}\right)\right) - i\left(f\left(n+\frac{N}{4}\right) - f\left(n+\frac{3N}{4}\right)\right)\right)\overline{w_N}^{n}$.

$$\begin{aligned}
\widehat{p}(k) &= \widehat{f}(4k+2) \\
&= \sum_{n=0}^{\frac{N}{4}-1} \left(f(n) + \overline{w_4}^{4k+2} f\left(n+\frac{N}{4}\right) + \overline{w_4}^{2(4k+2)} f\left(n+\frac{N}{2}\right) + \overline{w_4}^{3(4k+2)} f\left(n+\frac{3N}{4}\right)\right)\overline{w_N}^{n(4k+2)}.
\end{aligned}$$

Usando que $\overline{w_4}^{4k+2} = -1$, $\overline{w_4}^{2(4k+2)} = 1$ y $\overline{w_4}^{3(4k+2)} = -1$,

$$\begin{aligned}\widehat{p}(k) &= \sum_{n=0}^{\frac{N}{4}-1}\left(f(n)+f\left(n+\tfrac{N}{2}\right)-\left(f\left(n+\tfrac{N}{4}\right)+f\left(n+\tfrac{3N}{4}\right)\right)\right)\cdot\overline{w_N}^{2n}\overline{w_{\frac{N}{4}}}^{kn}\\ &= \sum_{n=0}^{\frac{N}{4}-1} p(n)\overline{w_{\frac{N}{4}}}^{kn},\end{aligned}$$

para $k = 0, 1, ..., \frac{N}{4} - 1$, con $p(n) = \left(f(n)+f\left(n+\frac{N}{2}\right)-\left(f\left(n+\frac{N}{4}\right)+f\left(n+\frac{3N}{4}\right)\right)\right)\cdot\overline{w_N}^{2n}$.

$$\begin{aligned}\widehat{q}(k) &= \widehat{f}(4k+3)\\ &= \sum_{n=0}^{\frac{N}{4}-1}\left(f(n)+\overline{w_4}^{4k+3}f\left(n+\tfrac{N}{4}\right)+\overline{w_4}^{2(4k+3)}f\left(n+\tfrac{N}{2}\right)+\overline{w_4}^{3(4k+3)}f\left(n+\tfrac{3N}{4}\right)\right)\overline{w_N}^{n}\overline{w}_{\frac{N}{4}}^{kn}.\end{aligned}$$

Usando que $\overline{w_4}^{4k+3} = i$, $\overline{w_4}^{2(4k+3)} = -1$ y $\overline{w_4}^{3(4k+3)} = -i$.

$$\begin{aligned}\widehat{q}(k) &= \sum_{n=0}^{\frac{N}{4}-1}\left(\left(f(n)-f\left(n+\tfrac{N}{2}\right)\right)+i\left(f\left(n+\tfrac{N}{4}\right)-f\left(n+\tfrac{3N}{4}\right)\right)\right)\overline{w_N}^{3n}\overline{w_{\frac{N}{4}}}^{kn}\\ &= \sum_{n=0}^{\frac{N}{4}-1} q(n)\overline{w_{\frac{N}{4}}}^{nk},\end{aligned}$$

para $k = 0, 1, .., \frac{N}{4} - 1$, con $q(n) = \left(\left(f(n)-f\left(n+\frac{N}{2}\right)\right)+i\left(f\left(n+\frac{N}{4}\right)-f\left(n+\frac{3N}{4}\right)\right)\right)$.

El cálculo de $g(n)$, $h(n)$, $p(n)$ y $q(n)$ para $n = 0, 1, ..., \frac{N}{4} - 1$ se puede representar con la siguiente operación mariposa de dos etapas.

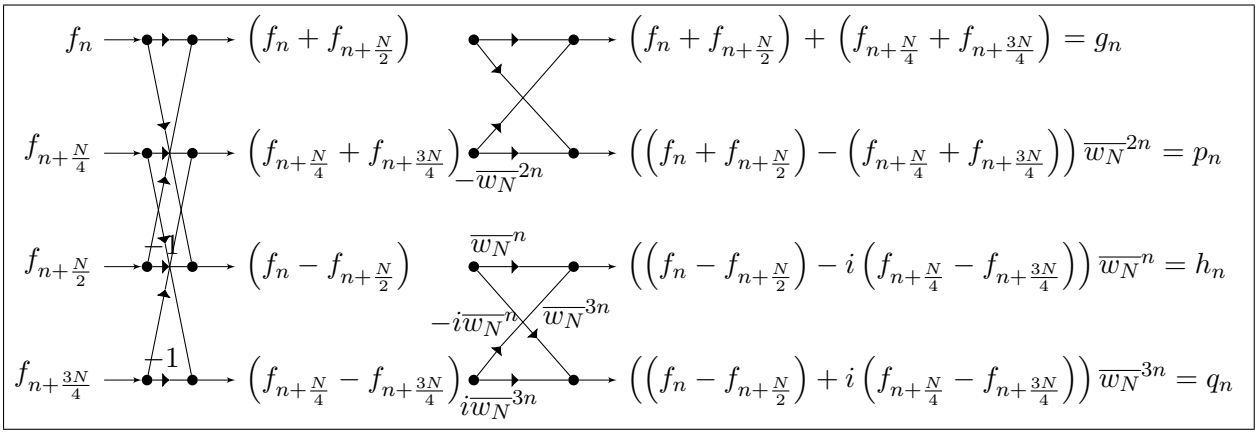


Figura 4.2: Operación mariposa, decimación en frecuencia, FFT radio 4.

Esta operación mariposa lo que hace es el cálculo de $g(n)$, $h(n)$, $p(n)$ y $q(n)$ en cada etapa de subdivisión, como podemos observar en este caso las multiplicaciones por las raíces de la unidad se hacen a posteriori.

El Algoritmo 6, es el algoritmo iterativo por el cual se divide cada subproblema es el que se muestra a continuación, donde hemos supuesto cálculados y almacenados los $\overline{w_N}^{n}$ que necesitamos como se explica en Anexco B. Los códigos que hemos usado para la implementación del algoritmo de radio 4 están en el anexo Anexco C.

Algoritmo 6 El algoritmo DIF FFT de radio 4 en pseudocódigo.

Require: $N = 4^n$, $w[n] = \overline{w_N}^n$, $w2[n] = \overline{w_N}^{2n}$, $w3[n] = \overline{w_N}^{3n}$

```
NumOfProblems :=1
ProblemSize := N
Jtwiddle = 0
while ProblemSize > 1 do
  HalfSize := ProblemSize/2
  quarterSize := ProblemSize/4
  threequarterSize := (3 · ProblemSize)/4
  for K = 0 to K = NumOfProblems − 1 do
    JFirst := K ∗ ProblemSize
    JLast := JFirst + HalfSize − 1
    Jtwiddle := (N − ProblemSize)/3
    for J = JFisrt to J = JLast do
      W:= w[Jtwiddle]
      W2:= w2[Jtwiddle]
      W3:= w3[Jtwiddle]
      Temp1:=a[J]
      Temp2:=a[J + quarterSize]
      A = temp1 + a[J + HalfSize]
      B = temp2 + a[J + threequarterSize]
      C = temp1 - a[J + HalfSize]
      D = temp2 - a[J + threequarterSize]
      a[J]:= A + B
      a[J + J + threequarterSize]:= (A - B)*W2
      a[J + HalfSize]:= (C - iD)*W
      a[J + threequarterSize]:= (C + iD)*W3
      Jtwiddle := Jtwiddle + 1
    end for
  end for
  NumOfProblems := 4 * NumOfProblems
  ProblemSize := ProblemSize/4
end while
return a
```

En Figura 4.3 se muestra un ejemplo para aplicar el algoritmo anterior para una secuencia de entrada de tamaño $N = 16 = 4^2$.

,

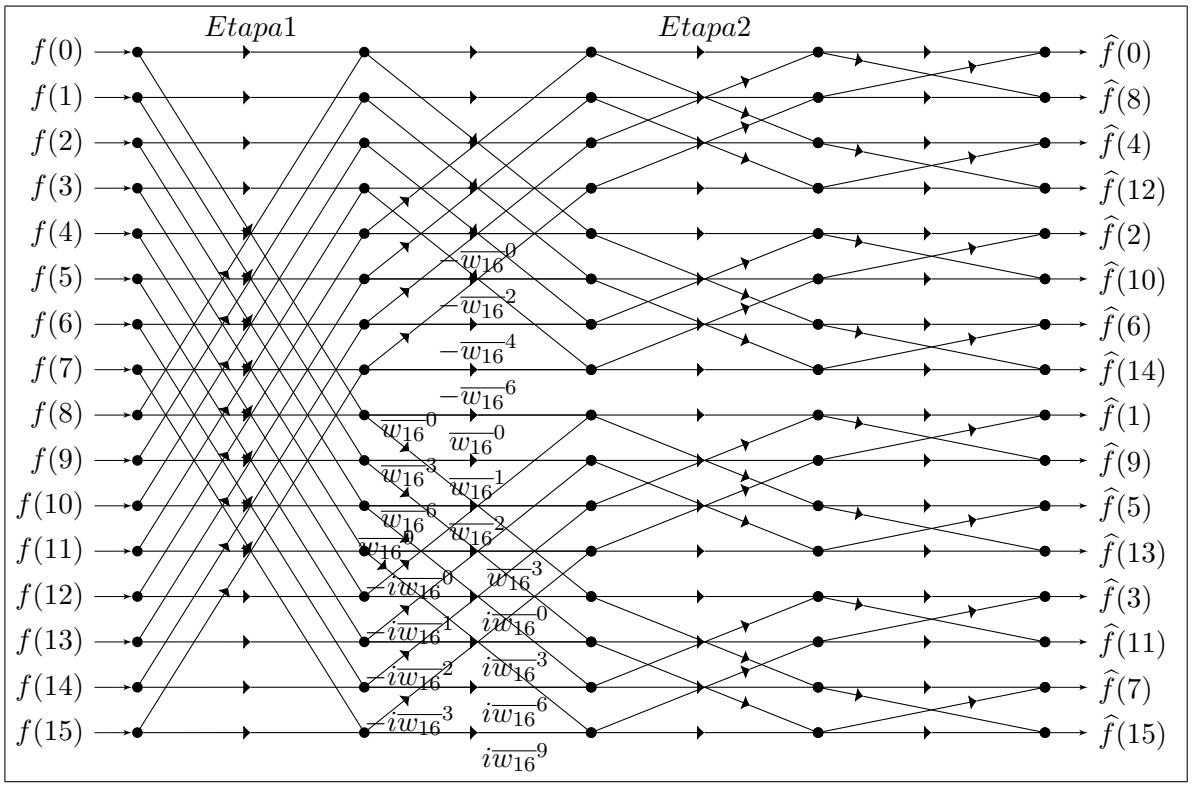


Figura 4.3: DIF FFT radio 4 $N = 16$

4.2. FFT radio split

Una vez estudiados los algoritmos FFT de radio 2 y FFT de radio 4, es interesante ver como el coste computacional de los algoritmos FFT puede reducirse combiando ambos, en los denominados algoritmos de radio split, algoritmo que fue propuesto por primera vez por Duhamel and Hollmann en 1984.

Es interesante señalar que, por ejemplo en el caso de $N = 2^{2k+1}$ no se trata de hacer un paso del algoritmo de radio 2 para continuar con los pasos del algoritmo de radio 4, que tal y como comprobó Singelton en 1983 no es mucho más eficiente que un buen algoritmo de radio 2. Es otra cosa, tal y como veremos en los dos apartados siguientes.

De nuevo como en el resto de algoritmos vuelven a aparecer dos versiones: DIT FFT de radio split y DIF FFT de radio split. Este método se aplica a vectores de longitud $N = 2^n$.

4.2.1. DIT FFT radio split

El algoritmo DIT FFT de radio split se deriva de la fórmula (2,1), que define la transformada de Fourier discreta. Para ello, dividimos la suma en dos, donde por un lado aparecen los términos pares y por otro lado los impares, a su vez la suma de los términos impares queda dividida en dos: donde aparecen los términos: $f(4k+1)$ y $f(4k+3)$.

La operación mariposa que nos permite el cálculo de la transformada de Fourier discreta es la siguiente:

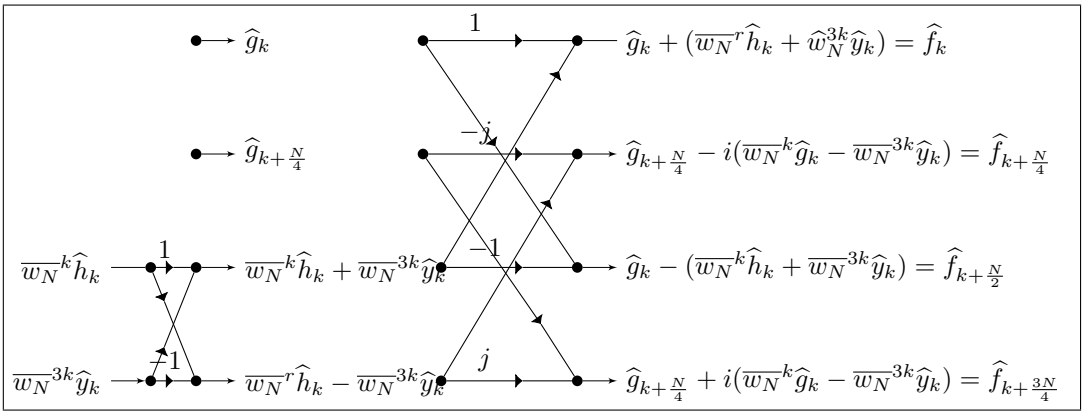


Figura 4.4: Operación mariposa, decimación en tiempo, FFT radio split.

4.2.2. DIF FFT radio split

El algoritmo DIF FFT consiste en dividir la suma que define la transformada de Fourier discreta en tres sumas una con la mitad de los sumandos y otras dos sumas con un cuarto de los sumandos originales cada una.

La operación mariposa que nos permite el cálculo de la transformada de Fourier discreta es la siguiente:

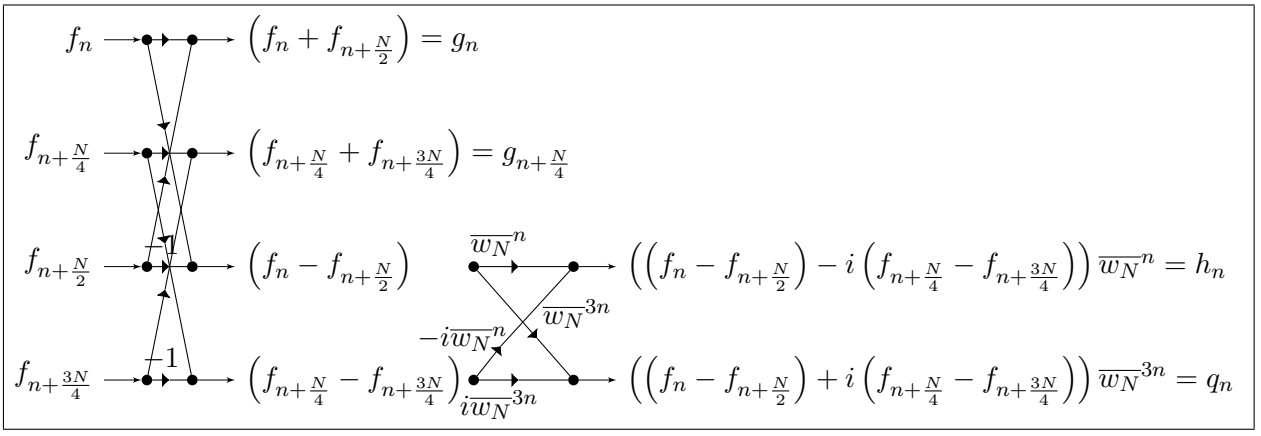


Figura 4.5: Operación mariposa, decimación en freciencia, radio split.

La implementación de estos algoritmos en Java se muestra en el Anexco C.

Coste computacinal del algoritmo:

El coste computacional de este algoritmo viene dado por $U(N) = 4N \cdot \log_2(N) - 6N + 8$.

Si comparamos este coste con el coste del algoritmo DIT FFT de radio 2, el cual recordamos que valía $T(N) = 5N \cdot \log_2 N$ tenemos:

$$\lim_{N\to\infty} \frac{U(N)}{T(N)} = \frac{4}{5}.$$

Con lo cual estamos ahorrando un 20 % respecto a los algoritmos de radio 2.

Si lo comparamos con el coste computacional de un algoritmo DIT FFT de radio 4 tenemos:

$$\lim_{N\to\infty} \frac{U(N)}{S(N)} = \frac{4}{\frac{17}{4}} = \frac{16}{17}.$$

El ahorro sería de un 5 %.

5 | Aplicaciones

En este capítulo veremos dos aplicaciones de la FFT, por un lado la multiplicación de números grandes, una aplicación totalmente prática de los algoritmos que hemos implementado en Java. Y por otro lado, el cálculo aproximado de frecuencias de señales analógicas, una aplicación más teórica que la anterior, en la que hemos usado un código en Matlab y donde incluimos algunos ejemplos concretos.

5.1. Multiplicación de números grandes

Los contenidos de esta sección hacen referencia a los libros [1], [2], [4] y a los apuntes de clase [11].

Si consederamos dos números 123 y 456, y realizamos su multiplicación usualmente hacemos los siguientes cálculos:

$$
\begin{array}{rl}
123 & \\
\times 456 & \\
\hline
738 & \Longleftarrow 3 \text{ productos de números de una cifra} \\
615 & \Longleftarrow 3 \text{ productos de números de una cifra} \\
\underline{492} & \Longleftarrow 3 \text{ productos de números de una cifra} \\
56088 & \Longleftarrow 3 \text{ sumas}
\end{array}
$$

donde podemos ver que para la multiplicación de dos números enteros de longitud tres necesitamos realizar 9 productos de números de una cifra y 3 sumas.

En general, si consideramos dos números enteros grandes de n dígitos $A = a_{n-1} \dots a_1 a_0$ y $B = b_{n-1} \dots b_1 b_0$ su multiplicación tiene una complejidad de $O(n^2)$, ya que tenemos que realizar n^2 multiplicaciones y n sumas, pero en esta sección vamos a ver que utilizando los algoritmos de la FFT podemos conseguir que la multiplicación de números grandes tenga un coste aritmético del orden de $n \log(n)$.

Para ello si fijamos una base b, cada número natural A se puede escribir en dicha base como

$$A = a_0 \cdot 1 + a_1 \cdot b + \ldots + a_{n-1} \cdot b^{n-1}$$

para ciertos dígitos $a_i \in \{0, 1, 2, \dots, b-1\}$. Es por ello que la multiplicación de números "grandes"(es decir, para los que el valor n en la fórmula anterior es elevado) está íntimamente ligada a la

multiplicación de polinomios de grado elevado, problema que como veremos también se puede abordar usando la FFT.

Así, en este capítulo resolveremos ambas cuestiones. Pero antes necesitamos introducir ciertos conceptos preliminares.

5.1.1. Polinomio interpolador

Un polinomio $p(x)$ en la variable x es una expresión del tipo:

$$p(x) = a_0 + a_1 x + \ldots + a_{n-1}x^{n-1} = \sum_{i=0}^{n-1} a_i x^i$$

donde los a_i son números reales (resp. complejos) y $a_{n-1} \neq 0$. Tomamos $a = [a_0, a_1, ..., a_{n-1}]$ el vector de coeficientes del polinomio. En estas condiciones decimos que el polinomio $p(x)$ tiene grado $n-1 \in \mathbb{N}$.

Para calcular el valor de un polinomio p de grado n-1 en un punto x_0 si usamos la relación $x_0^{k+1} = x_0 \cdot x_0^k$, necesitamos hacer $1+2\cdot(n-2) = 2n-3$ multiplicaciones (1 en a_1x_0 y 2 multiplicaciones en cada sumando $a_k x_0^k$ para $k \geq 2$) y $n-1$ sumas. En total $3n-4$ operaciones. Es decir, este proceso tiene una complejidad de $O(n)$, donde n es el grado del polinomio.

Multiplicar dos polinomios $p(x) = a_0 + a_1x^1 + ... + a_{n-1}x^{n-1}$ y $q(x) = b_0 + b_1x^1 + ... + b_{n-1}xn - 1$ da como resultado un tercer polinomio

$$p(x) \cdot q(x) = a_0b_0 + (a_0b_1 + a_1b_0)x + ... + (a_{n-1}b_{n-1})x^{2n-2}.$$

Esta operación tiene un coste aritmético $O(n^2)$, al igual que el coste de multiplicar dos números enteros. Para disminuir este coste vamos a introducir otra forma de representar un polinomio, distinta de la usual.

Definición 5.1:

Consideramos la familia $\{(x_i, y_i)\}_{i=0}^{n-1}$ de n puntos con las abscisas (x_i) distintas dos a dos. Se llama **polinomio interpolador** en los puntos $\{(x_i, y_i)\}_{i=0}^{n-1}$ a un polinomio $p(x)$ de grado menor o igual que $n-1$ tal que $p(x_i) = y_i$ para cada $i = 0, .., n-1$.

Teorema 5.1:

Dados n puntos $\{(x_i, y_i)\}_{i=0}^{n}$ con las abscisas distintas dos a dos ($x_i \neq x_j$ si $i \neq j$), existe un único polinomio interpolador en esos puntos.

Demostración. Si escribimos las ecuaciones $y_i = p(x_i)$ donde las incognitas son los coeficientes del polinomio $p(x) = a_0 + a_1x + ... + a_nx^n$, se tiene el siguiente sistema lineal de $n+1$ ecuaciones con $n+1$ incognitas:

$$\begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix} \tag{5.1}$$

donde la matriz V del sistema es la matriz de Vandermonde correspondiente a las potencias de las abcisas x_i, cuyo determinante es (ver [6]):

$$det(V) = \prod_{0 \le i < j \le n} (x_j - x_i) \neq 0,$$

que es distinto de cero ya que, por hipótesis, $x_i \neq x_j$ para $i \neq j$ y en consecuencia el sistema lineal tiene una única solución, que determina univocamente el polinomio interpolador.

Ahora construimos el polinomio interpolador por inducción:

- Para $n = 0$ tomamos el polinomio constante $P_0(x) = y_0$.
- Suponemos que $P_{n-1}(x)$ es el polinomio interpolador en n puntos $\{(x_j, y_j)\}_{j=0,\dots,n-1}$.
- Buscamos el polinomio interpolador P_n en los n+1 puntos $\{(x_j, y_j)\}_{j=0,\dots,n}$ de forma que

$$P_n(x) = P_{n-1}(x) + C_n \prod_{j=0}^{n-1} (x - x_j)$$

con independencia del valor C_n siempre se cumple que $P_n(x_j) = P_{n-1}(x_j)$ para $j = 0, ..., n-1$. Si queremos que $P_n(x_n) = y_n$ bastará con elegir

$$C_n = \frac{y_n - P_{n-1}(x_n)}{\prod_{j=0}^{n-1}(x_n - x_j)}$$

□

Este teorema nos dice que si tenemos un polinomio de grado $n-1$ y lo evaluamos en n puntos distintos, entonces la entrada de esos n puntos y los otros n puntos de la salida dan una representación única de ese polinomio. Entonces como el producto de dos polinomios p(x) y q(x) de grado n-1 nos da un polinomio de grado 2n-2, con el Teorema 5.1 sabemos que este polinomio se define de forma única con su evaluación en 2n puntos distintos. Por tanto, la idea principal para la multiplicación de polinomios usando la FFT se basaría en tres pasos: el primero de ellos sería evaluar el polinomio p en esos 2n puntos y seguidamente el polinomio q, el tercer paso consistiría en calcular los 2n productos de p y q evaluados en esos 2n puntos que nos dan 2n evaluaciones del producto de p y q en 2n puntos distintos (por el teorema de interpolación se esto defiene a $p \cdot q$ de forma única).

Pero evaluar 2n entradas diferentes sigue teniendo un coste $O(n^2)$. Para intentar disminuir este coste se utilizan las raíces de la unidad, las cuales tienen varias propiedades específicas la propiedad reflexiva, (es decir, si w es cualquiera de las raíces primitivas n-ésimas de la unidad, entonces, $w^{k+\frac{n}{2}} = -w^k$) y la de reducción, (esto es, que si w es una raiz 2n-ésima primitiva de la unidad, entonces w^2 es una raíz n-ésima primitiva de la unidad).

5.1.2. Multiplicación de polinomios con algoritmos FFT

Si recordamos que en el capítulo 1 definiamos la transformada de Fourier discreta y la transformada de Fourier inversa con las fórmulas (1,1) y (1,2) respectivamente. Podemos definir la transformada de Fourier discreta de un polinomio p(x) de grado n-1 como la evaluación del polinomio en $\overline{w_{n-1}}^0, \overline{w_{n-1}}^1, ..., \overline{w_{n-1}}^{n-1}$.

Si este cálculo lo realizamos de manera ingenua todavía tenemos un coste $O(n^2)$ pero usando los algoritmos FFT que ya tenemos podemos conseguir disminuirlo de forma sensible.

Ahora bien, como ya sabemos el producto de dos polinomios $p(x)$ y $q(x)$ de grado menor o igual que $n-1$ nos da como resultado un polinomio de grado menor o igual que $2n-1$ que para ser determinado de forma única requiere $2n$ evaluaciones de puntos distintos (por ejemplo, podemos elegir como puntos donde se evalúa el polinomio las raíces $(2n)$-ésimas de la unidad). Para usar un algoritmo FFT necesitamos vectores de coeficientes de longitud $N = 2^k$, y como con los coeficientes de cada uno de los polinomios sólo conseguimos vectores de logitud n vamos a completar con ceros la lista de dichos coeficientes hasta la posición $2n$, es decir, el vector de coeficientes de p será $a = (a_0, a_1, ..., a_{n-1})$ y el de q será $b = (b_0, b_1, ..., b_{n-1})$ y completaremos ambos hasta que tengan logitud $2n$ de la forma $a' = (a_0, a_1, ..., a_{n-1}, 0..., 0)$ y $b' = (b_0, b_1, ..., b_{n-1}, 0..., 0)$. A continuación si $2n$ es potencia de dos ya tenemos el tamaño que deseamos pero si no lo que tenemos que hacer es rellenarlos en lugar de hasta $2n$ hasta $N = 2^k$, donde k es el menor número entero tal que $2^k \geq 2n$.

Ahora, si evaluamos p en $\overline{w_N}^J$ para $J = 0..., N-1$ usando un algoritmo $DIF_{NR}FFT$ obtenemos $X = dft(a')$, es decir, la transformada de Fourier discreta del vector a' y de la misma forma con q, lo evaluamos en $\overline{w_N}^J$ para $k = 0..., N-1$ y obtenemos $Y = dft(b')$, entonces el producto de las evaluaciones de p y q en $\overline{w_N}^J$ para $J = 0..., N-1$ será, (teniendo en cuenta la propiedad número 2 de la Proposición 2.3), $Z = X \cdot Y$, y por último aplicando la transformda de Fourier inversa a Z, con un algoritmo $DIF_{RN}FFT$ ya que como X e Y son salidas de algoritmos con entrada en orden natural y salida en orden bit-reversal, están en orden bit-reversal y por tanto Z que es el producto de estos dos vectores también lo está, obtenemos los coeficientes del polinomio $p(x)q(x)$.

El coste de este proceso usando los algoritmos FFT será:

- El coste para calcular el vector $X = fft(a')$, que como ya sabemos es $5N \log_2 N$.
- El coste para calcular el vector $Y = fft(b')$, que como ya sabemos es de nuevo $5N \log_2 N$.
- El coste de la multiplicación de $X \cdot Y$ que es N.
- Calcular la transformada de Fourier inversa de $Z = X \cdot Y$, que tiene un coste de $5N \log_2 N$.

Por tanto el coste del proceso total será : $15N \log_2 N + N$ por lo que podemos decir que tiene un coste del orden de $O(N \log_2 N)$.

5.1.3. Representación de enteros como polinomios

Si queremos representar un entero como un polinomio, tenemos que elegir la base que queremos utilizar. Cualquier número entero positivo puede ser usado como base, pero nosotros por comodidad vamos a escoger como base la formada por las potencias de 10.

Por ejemplo, si tenemos el número 123456 como hemos escogido como base la formada por 10 su vector de coeficientes será $a = [6, 5, 4, 3, 2, 1]$.

Por tanto para realizar la multiplicación de números grandes aplicando los algoritmos FFT nos basaremos en la multiplicación de polinomios que hemos explicado.

Los pasos a seguir son los que siguen:

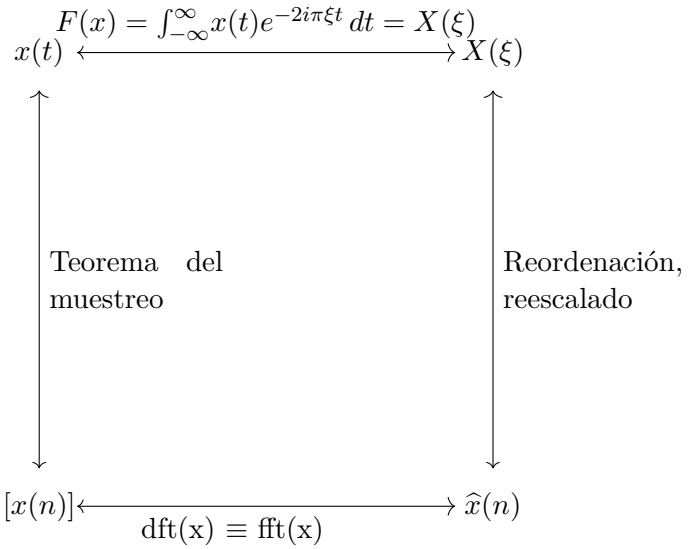
- Expresar cada uno de los números a multiplicar en base 10 y colocar los coeficientes de dicho polinomio en un vector.
- Completar, añadiendo ceros, el vector de coeficientes de cada uno de los números hasta la siguiente potencia de dos mayor que 2n.
- Aplicar la FFT a cada vector obtenido de este modo.
- Multiplicar, coordenada a coordenada, los dos vectores del punto anterior.
- Aplicar la FFT inversa al vector que resulta de la multiplicación anterior.

Los algoritmos que hemos implementado para esta aplicación los tenemos en el Anexo D.

5.2. Cálculo aproximado de frecuencias de señales analógicas

En este apartado vamos a estudiar cómo se puede obtener una aproximación de la transformada de Fourier de una señal analógica tomando muestras discretas, aplicando la transformada de Fourier rápida, reordenando datos y reescalando, es decir, vamos a ver cómo, gracias al uso del Teorema del muestreo, si se toman muestras uniformes de una señal analógica con una frecuencia de muestreo suficientemente elevada, es posible, utilizando solamente la información contenida en dichas muestras, obtener información relevante sobre el comportamiento en frecuencias (es decir, sobre la transformada de Fourier $\mathcal{F}(x) = \int_{-\infty}^{\infty} x(t)e^{-2\pi i t\xi}dt$) de una señal analógica $x(t)$.

Este problema queda representado de forma esquemática en la siguiente figura:



Los contenidos de esta sección se basan en los libros [1] y [2].

Las señales del mundo real, con las que debemos trabajar en todo tipo de aplicaciones, no son vectores finitos sino funciones definidas sobre un intervalo finito o infinito. Por ejemplo, en Medicina y en Ingeniería es frecuente que se consideren magnitudes físicas continuas, como los potenciales eléctricos (al medir electrocardiogramas o electroencefalogramas) o las diferencias de presión atmosférica (al medir sonido), etc., y estas magnitudes quedan descritas por señales que dependen del tiempo, una variable susceptible de ser evaluada en todos los puntos de la recta real. Los ingenieros denominan "analógicas" a este tipo de señales (o funciones) y, para estudiarlas en términos de frecuencias utilizan la transformada de Fourier analógica, que viene dada por la expresión:

Definición 5.2:

Llamamos **transformada de Fourier analógica o continua** de una señal $x(t) \in L^1(\mathbb{R})$, a la función $\mathcal{F}(x)$ dada por la expresión

$$\mathcal{F}(x)(\xi) := \int_{-\infty}^{\infty} x(t)e^{-2\pi it\xi}dt.$$

Para denotar la transformada de Fourier analógica también se usan las siguientes notaciones: $X(\xi) = \hat{x}(\xi) = \mathcal{F}(x)(\xi)$.

Para funciones T-periódicas tenemos la siguiente definición:

Definición 5.3:

Llamamos **transformada de Fourier analógica o continua** de una señal $x(t)$ T-periódica, a la funcion $\mathcal{F}(x)$ dada por la expresión $\mathcal{F}(x)(\xi) = \{c_k\}_{k=-\infty}^{\infty}$, donde los valores

$$c_k(x) = \frac{1}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}} x(t)e^{-\frac{2\pi ikt}{T}}dt \text{ para } k \in \mathbb{Z}$$

representan los llamados coeficientes de Fourier de la función T-periódica $x(t)$.

El Análisis de Fourier debe sus orígenes al estudio de diversos problemas físicos, especialmente en acústica (donde aparece el término de armónico puro y donde las señales sonoras se interpretan como superposición o suma de armónicos puros de forma natural) y en el estudio de la conducción del calor en sólidos conductores. Primero surgieron los coeficientes de Fourier y sus series de Fourier asociadas -que nos proporcionan el mecanismo para recuperar una función periódica a partir de sus coeficientes de Fourier (es decir, de su transformada de Fourier), que representan cómo se descompone dicha señal como suma o superposición de armónicos puros y, por tanto, a partir de su descripción en frecuencias.

Si bien los coeficientes de Fourier y las series trigonométricas asociadas estaban ya en el ambiente antes de la llegada del matemático francés, por ejemplo en el trabajo de Daniel Bernoulli, fue Fourier quien introdujo la transformada $\mathcal{F}(x)(\xi) = \int_{-\infty}^{\infty} x(t)e^{-2\pi it\xi}dt$ para funciones $x(t)$ que no son necesariamente periódicas.

La forma en la que abordó este problema, que le sirvió para motivar que la función $\mathcal{F}(x)(\xi)$ admita de forma natural una interpretación en términos de frecuencias, resulta relevante para los cálculos que queremos realizar en esta sección del capítulo.

Partiendo de una función $x \in L^2(\mathbb{R})$, tomó un valor $T > 0$ y consideró la función $x_T(t)$ que, restringida al intervalo $[-T/2, T/2)$ coincide con la función original $x(t)$ pero que, además, es T-periódica. Como se trata de una función T-periódica y, además, $x_T \in L^2(-T/2, T/2)$, dicha función queda completamente determinada a partir de sus coeficientes de Fourier y se recupera mediante la fórmula

$$x_T(t) = \sum_{k=-\infty}^{\infty} c_k(x_T) e^{\frac{2\pi i k t}{T}},$$

donde la suma se determina en el sentido de la norma de $L^2(-T/2, T/2)$.

Como es bien sabido, $c_k(x_T) = \frac{1}{T}\int_{-T/2}^{T/2} x_T(t) e^{-\frac{2\pi i k t}{T}} dt = \frac{1}{T}\int_{-T/2}^{T/2} x(t) e^{-\frac{2\pi i k t}{T}} dt$, pues $(x_T)_{[-T/2,T/2)} = x_{[-T/2,T/2)}$. Ahora bien, para valores de T elevados, tendremos que $x_{|\mathbb{R}\setminus[-T/2,T/2)}$ es muy pequeña en módulo, hasta el punto de que la integral $\int_{\mathbb{R}\setminus[-T/2,T/2)} x(t) e^{-\frac{2\pi i k t}{T}} dt$ tiene un valor despreciable, que podemos considerar nulo, por lo que

$$c_k(x_T) = \frac{1}{T}\int_{-T/2}^{T/2} x(t) e^{-\frac{2\pi i k t}{T}} dt \simeq \frac{1}{T}\int_{-\infty}^{\infty} x(t) e^{-\frac{2\pi i k t}{T}} dt = \frac{1}{T}\mathcal{F}(x)(\frac{k}{T}) = \frac{1}{T}X(\frac{k}{T}).$$

Es decir,

$$X(\frac{k}{T}) \simeq T c_k(x_T). \tag{5.2}$$

Es de este modo que la transformada de Fourier $X(\xi)$ adquiere, además, un significado en términos de frecuencias: Si $\xi = \frac{k}{T}$ para un valor de T suficientemente elevado, entonces $X(\xi)$ es aproximadamente la aportación en frecuencias a la señal $x(t)$ del k-ésimo armónico puro que interviene en su descripción como superposición de armónicos puros cuando consideramos la señal restringida al intervalo $[-T/2, T/2]$. Al perder la periodicidad aparecen frecuencias en todo el espectro real, pues k/T puede tomar cualquier valor real. Además, haciendo uso nuevamente de las series de Fourier de las señales x_T y del proceso de sumación de Riemann para las integrales, Fourier descubrió que el operador $\mathcal{F}$ tiene un inverso natural, que viene dado por

$$\mathcal{F}^{-1}(x)(t) = \int_{-\infty}^{\infty} x(\xi) e^{2\pi i \xi t} d\xi,$$

por lo que el viaje desde el dominio del tiempo al de la frecuencia es de ida y vuelta, tal como sucedía en el caso de las señales periódicas.

En la vida real existen numerosas situaciones en las que se hace necesario estudiar el comportamiento en frecuencias de una señal analógica $x(t)$. Sin embargo, típicamente no podemos abordar el cálculo directo de la transformada de Fourier de la señal $x(t)$, bien porque desconocemos la expresión analítica de la señal $x(t)$ o bien porque, aún conociendo dicha expresión, no es posible calcular una primitiva de la función $x(t)e^{-2\pi i t\xi}$. Sería estupendo tener un mecanismo que, discretizando el problema, nos permitiera hallar una buena aproximación de los valores $X(\xi)$ independientemente de quién sea la señal $x(t)$. Es en este punto donde entran en escena unas ideas que han demostrado ser de vital importancia en matemática aplicada y para conocerlas debemos definir unos conceptos previos:

Definición 5.4:

Decimos que una señal $x(t) \in L^2(\mathbb{R})$ es de **banda limitada** si la transformada de Fourier $\widehat{x}(\xi)$ posee soporte compacto. Es decir, $x(t)$ se dice de banda limitada si existe algún $b > 0$ tal que $\widehat{x}(\xi) = 0 \; \forall \; |\xi| \; > b$.

NOTA: Para la definición de la transformada de Fourier en $L^2(\mathbb{R})$, ver el Anexo E.

Definición 5.5:

El menor b verificando la definición anterior se llama **ancho de banda** de la señal x(t). El espacio de señales $x(t) \in L^2(\mathbb{R})$ tal que $supp(\widehat{x}) \subseteq [-b, b]$ se denota por PW(b) y se llama espacio de **Paley-Wiener**.

Con estos conceptos podemos pasar a enunciar el teorema del muestreo de Shannon-Whitaker-Kotelnikov, el cual nos facilita la discretización de una señal "arbitraria", simplemente tomando muestras de forma uniforme en el tiempo, sin perder por ello la posibilidad de recuperar, a partir de los valores de dichas muestras, el valor de la función en todos los puntos de la recta real. Es decir, el siguiente teorema nos permite pasar del mundo análogico al mundo digital sin perder información relevante.

Teorema 5.2:

Supongamos que $x(t) \in L^2(\mathbb{R})$ y $\widehat{x}(\xi) = 0$ para todo $|\xi| > b$. Entonces $x(t)$ queda completamente determinada a partir de sus valores en los puntos $\left\{\frac{k}{2b}\right\}_{k\in\mathbb{Z}}$. Además se satisface la fórmula:

$$x(t) = \sum_{k=-\infty}^{\infty} x(\frac{k}{2b}) \frac{\sin(\pi(-k+2bt))}{\pi(-k+2bt)},$$

donde la serie converge en $L^2(\mathbb{R})$ y también absoluta y uniformemente en $\mathbb{R}$.

Demostración. Sea $x(t) \in PW(b)$. Entonces $\widehat{x}(\xi) \in L^2(\mathbb{R})$ y $supp(\widehat{x}) \subseteq [-b, b]$. Por tanto , $\widehat{x}|_{[-b,b]} \in L^2(-b, b)$.

Tomamos $F(\xi)$, que denota extensión $2b$-periódica de $\widehat{x}|_{[-b,b]}$. Entonces F es una función $2b$-periódica y además $F(\xi) \in L^2(-b, b)$, por lo que F queda completamente determinada por sus coeficientes de Fourier y se recupera mediante la fórmula:

$$F(\xi) = \sum_{k=-\infty}^{\infty} c_k(F) \cdot e^{\frac{2i\pi k\xi}{2b}} = \sum_{k=-\infty}^{\infty} c_k(F) \cdot e^{\frac{\pi ik\xi}{b}} \qquad \text{en} \qquad L^2(-b, b).$$

Ahora bien, los coeficientes de Fourier de F están dados por:

$$c_k(F) = \langle F, e^{\frac{2\pi ik\xi}{2b}} \rangle \frac{1}{2b} = \frac{1}{2b} \int_{-b}^{b} F(\xi) e^{\frac{-2\pi ik\xi}{2b}} d\xi.$$

Como $\widehat{x}|_{[-b,b]} = F|_{[-b,b]}$ y $\widehat{x}|_{\mathbb{R}\setminus[-b,b]} = 0$ podemos escribir la siguiente igualdad:

$$c_k(F) = \frac{1}{2b} \int_{-\infty}^{\infty} \widehat{x}(\xi) e^{\frac{2\pi i(-k)\xi}{2b}} d\xi$$

Usando el teorema integral de Fourier (mirar Anexo E), se sigue que:

$$c_k(F) = \frac{1}{2b} \mathcal{F}^{-1}\left(\mathcal{F}(x)\right)\left(\frac{-k}{2b}\right) = \frac{1}{2b} x\left(\frac{-k}{2b}\right) \tag{5.3}$$

Por lo que

$$F(\xi) = \frac{1}{2b} \sum_{k=-\infty}^{\infty} x(\frac{k}{2b}) e^{\frac{-\pi i k \xi}{b}}$$

en $L^2(-b, b)$. Y por tanto

$$\widehat{x}(\xi) = \frac{1}{2b} \sum_{k=-\infty}^{\infty} x\left(\frac{k}{2b}\right) e^{\frac{-\pi i k \xi}{b}} \chi_{[-b,b]}(\xi)$$

en $L^2(\mathbb{R})$. Esto quiere decir que la transformada de Fourier de x queda completamente determinada a partir de los valores que toma la señal en los puntos $\frac{k}{2b}$ para $k \in \mathbb{Z}$ y, por tanto, haciendo uso de nuevo del teorema integral de Fourier, también la propia señal $x(t)$ queda determinada a partir de estos valores.

Aplicando de nuevo el teorema integral de Fourier (mirar Anexo E) tenemos

$$\begin{aligned}
\mathcal{F}^{-1}(e^{\frac{-i\pi k\xi}{b}} \chi_{[-b,b]}(\xi)) &= \int_{-\infty}^{\infty} e^{\frac{-i\pi k\xi}{b}} \chi_{[-b,b]}(\xi) e^{2\pi i t\xi} d\xi \\
&= \int_{-b}^{b} e^{\frac{-i\pi k\xi}{b} + 2\pi i t\xi} d\xi \\
&= \int_{-b}^{b} e^{2\pi i \xi(\frac{-k}{2b} + t)} d\xi \\
&= \frac{e^{2\pi i \xi(\frac{-k}{2b} + t)}}{2\pi i(\frac{-k}{2b} + t)} \Bigg]_{\xi=-b}^{\xi=b} \\
&= \frac{2i \sin(2\pi b(\frac{-k}{2b} + t))}{2\pi i(\frac{-k}{2b} + t)} \\
&= \frac{\sin(2\pi b(\frac{-k}{2b} + t))}{\pi(\frac{-k}{2b} + t)} \\
&= \frac{\sin(\pi(-k + 2bt))}{\frac{1}{2b}\pi(-k + 2bt)} \\
&= \frac{2b \sin(\pi(-k + 2bt))}{\pi(-k + 2bt)} \\
&= \frac{2b \sin(\pi(-k + 2bt))}{\pi(-k + 2bt)}
\end{aligned}$$

Por tanto,

$$\mathcal{F}^{-1}(e^{\frac{-i\pi k\xi}{b}} \chi_{[-b,b]}(\xi)) = \frac{2b \sin(\pi(-k + 2bt))}{\pi(-k + 2bt)}$$

y la fórmula del Teorema del muestreo queda:

$$x(t) = \sum_{k=-\infty}^{\infty} x(\frac{k}{2b}) \frac{\sin(\pi(-k + 2bt))}{\pi(-k + 2bt)}.$$

Pasamos ahora a demostrar la convergencia uniforme y absoluta de la serie:

Consideremos los errores de la serie de valores absolutos:

$$E_N(t) = \sum_{|k|>N} \left| x(\frac{k}{2b}) \frac{\sin(\pi(-k+2bt))}{\pi(-k+2bt)} \right|$$

Por la desigualdad de Cauchy-Schwarz, sabemos que

$$E_N(t) \leq \left(\sum_{|k|>N} \left| x(\frac{k}{2b}) \right|^2 \right)^{\frac{1}{2}} \left(\sum_{|k|>N} \left| \frac{\sin(\pi(-k+2bt))}{\pi(-k+2bt)} \right|^2 \right)^{\frac{1}{2}}$$

Ahora bien: como $\{x(k/(2b))\}$ son los coeficientes de Fourier de F, esta sucesión vive en $\ell^2(\mathbb{Z})$ y

$$\left(\sum_{|k|>N} \left| x(\frac{k}{2b}) \right|^2 \right)^{\frac{1}{2}} \to 0, \text{ para } N \to \infty.$$

La prueba finaliza una vez se demuestre que

$$\sup_{t\in\mathbb{R}} \left(\sum_{|k|>N} \left| \frac{\sin(\pi(-k+2bt))}{\pi(-k+2bt)} \right|^2 \right)^{\frac{1}{2}} \leq M$$

para cierta constante $M < \infty$ que no depende de t ni de N. Ahora bien, si consideramos la función

$$G(t) = \sum_{k=-\infty}^{\infty} \left| \frac{\sin(\pi(-k+2bt))}{\pi(-k+2bt)} \right|^2$$

y hacemos el cambio de variable $s = 2\pi bt$, entonces $G(t) = H(s)$ donde

$$H(s) = \sum_{k=-\infty}^{\infty} \left| \frac{\sin(-k\pi+s)}{-k\pi+s} \right|^2,$$

que es una función periódica con periodo π. Por tanto, $\sup_{t\in\mathbb{R}} G(t) = \sup_{s\in\mathbb{R}} H(s) = \sup_{s\in[-\pi/2,\pi/2]} H(s)$. Si $s \in [-\pi/2, \pi/2]$ y $k \neq 0$ entonces es claro que $|s - k\pi| \geq |k\pi| - |s| \geq |k\pi| - \pi/2 \geq |k|\pi/2$, por lo que

$$\left| \frac{\sin(-k\pi+s)}{-k\pi+s} \right|^2 \leq \frac{1}{(|k|\pi/2)^2}$$

y, por tanto,

$$\sup_{s\in\mathbb{R}} H(s) \leq 1 + \sum_{|k|\neq 0} \left| \frac{\sin(-k\pi+s)}{-k\pi+s} \right|^2 \leq 1 + \sum_{k\neq 0} \frac{1}{(|k|\pi/2)^2} = K < \infty.$$

Esto finaliza la prueba.

□

NOTA: La demostración de la convergencia absoluta y uniforme que hemos mostrado en el teorema anterior se ha tomado de [5], adaptada a la definición dada en esta memoria de la transformada de Fourier.

NOTA: Como se explica en el libro [2] todos los casos de interes registrados por cualquier sistema de medición son necesariamente de banda limitada, por lo que el Teorema del Muestreo se puede aplicar siempre.

NOTA: El teorema del muestro nos dice que si b es el ancho de banda de la señal, entonces esta se puede recuperar a partir de muestras equiespaciadas cuando el muestreo se realiza con la frecuencia mayor o igual que $2b$. Esta frecuencia se llama **frecuecia de Nyquist**.

Si bien el teorema del muestreo admite una demostración muy elegante -como acabamos de ver- cuando las funciones pertenecen al espacio de Paley-Wiener, lo cierto es que imponer que una señal $x(t)$ pertenezca a $PW(b)$ para algún $b > 0$ es, en realidad, una condición extremadamente restrictiva, sobre todo porque todas estas señales deben decaer rápidamente en el infinito. Esto impide aplicar el teorema a señales tan importantes y habituales como las funciones trigonométricas $\sin(\theta t), \cos(\theta t)$ y $e^{i\theta t}$. Sin embargo, existe una extensión del teorema que lo hace aplicable a una clase muchísimo más amplia de funciones que contiene a los polinomios trigonométricos. A continuación exponemos un esbozo de la demostración del Teorema del muestreo extendido, cuya prueba completa puede consultarse en [1]. Para lograr dicho resultado se requiere utilizar la teoría de distribuciones temperadas. Resumimos de forma breve los pasos a seguir:

- Se considera el conjunto $\mathbb{S} = \{f \in \mathcal{C}^{(\infty)}(\mathbb{R}) : \sup_{t\in\mathbb{R}} |t^n f^{(m)}(t)| < \infty \text{ para todo } n, m \in \mathbb{N}\}$ de las funciones de clase infinito de decrecimiento rápido. $\mathbb{S}$ tiene estructura de espacio vectorial topológico.

- Dada una función $f : \mathbb{R} \to \mathbb{R}$ continua a trozos y cuyo crecimiento en el infinito es más lento que el de algún polinomio (es decir, $\int_{-\infty}^{\infty} \frac{|f|}{|t^n|} < +\infty$ para algún n), se puede considerar el funcional $u_f : \mathbb{S} \to \mathbb{C}$ dado por $u_f(\phi) = \int_{-\infty}^{\infty} f(t)\phi(t)dt$, que es continuo. Lo interesante de este funcional es que determina unívocamente a la función f. Por tanto, podemos identificar f y u_f y pensar que los funcionales continuos $u : \mathbb{S} \to \mathbb{C}$ son una generalización de las funciones f. Estos funcionales se llaman distribuciones temperadas y se interpretan como "funciones generalizadas".

- Se define la transformada de Fourier de una distribución temperada u mediante la fórmula:

  $$\mathcal{F}(u)(\phi) = u(\mathcal{F}(\phi)),$$

  que explica el comportamiento de $\mathcal{F}(u)$ como funcional.

- Una distribución temperada u tendrá soporte compacto contenido en $[a, b]$ si $u(\phi) = 0$ siempre que el soporte de ϕ esté contenido en $\mathbb{R} \setminus [a, b]$.

- Una distribución u se dice de banda limitada si la distribución $\mathcal{F}(u)$ tiene soporte compacto (contenido en $[-b, b]$ para algún $b > 0$).

Resulta que las distribuciones de banda limitada son siempre del tipo u_f para alguna función ordinaria f y, por tanto, se pueden interpretar siempre como funciones ordinarias. Además, existe una demostración específica del teorema del muestreo para estas funciones. Este teorema supone una ampliación extremadamente importante del teorema que hemos probado en este TFG. En particular, se puede aplicar el teorema del muestreo, con convergencia puntual, a todos los polinomios trigonométricos.

Por otra parte, cuando la señal se ha discretizado, esta se convierte en un vector y, por tanto, disponemos para ella de una versión en frecuencias, que es su DFT (y que calculamos con el algoritmo FFT correspondiente). Veamos como la combinación de estas ideas permite el cálculo aproximado de la transformada de Fourier de una señal analógica aperiódica.

Supongamos que tenemos una señal T-periódica $x(t)$ y que esta se ha muestreado de forma uniforme con periodo de muestreo h en el intervalo $[0, T)$. Así, disponemos del vector de muestras

$$v = (x(0), x(h), x(2h), \cdots, x((N-1)h)),$$

donde $Nh = T$. Calculemos la dft del vector v: $dft(v) = (y_0, \cdots, y_{N-1})$. Entonces

$$y_k = \langle v, E_k \rangle = \sum_{j=0}^{N-1} x(jh) e^{-\frac{2\pi i k j}{N}}.$$

Como $Nh = T$, tenemos que $h/T = 1/N$ y, por tanto, $\frac{j}{N} = \frac{jh}{T}$, por lo que la identidad anterior se re-escribe como una suma de Riemann:

$$hy_k = h \sum_{j=0}^{N-1} x(jh) e^{-\frac{2\pi i k jh}{T}} \simeq \int_0^T x(t) e^{\frac{-2\pi i k t}{T}} dt,$$

lo que conduce a la relación:

$$hy_k \simeq T c_k(x). \tag{5.4}$$

Por tanto, podemos concluir que si tomamos muestras uniformes en un intervalo del tipo $[0, T)$ para una función T-periódica, la DFT del vector de muestras así calculado admite una interpretación natural en términos de los coeficientes de Fourier de dicha función.

Supongamos ahora que $x(t)$ vive en $L^2(\mathbb{R})$ y que tomamos muestras uniformes de esta señal en un intervalo simétrico con centro el origen de coordenadas y de longitud T. Nuestro vector de muestras viene dado, pues, por

$$v = (x(-T/2), x(-T/2 + h), \cdots, x(-h), x(0), x(h), \cdots, x(T/2 - h)).$$

Como la ecuación (5.2) relaciona los coeficientes de Fourier de la señal $x_T(t)$ con el valor de la transformada de Fourier de $x(t)$ en ciertos puntos, podemos utilizar la ecuación (5.4) para el cálculo aproximado de estos coeficientes de Fourier (y por tanto, también la transformada de Fourier de x). Ahora bien, para poder aplicar (5.4) necesito partir del vector de muestras uniformes de x_T en el intervalo $[0, T)$, que no es el vector v pero se obtiene a partir de él mediante una sencilla reorganización de los datos. Así, sabemos que si

$$\begin{aligned} w &= (x_T(0), x_T(h), x_T(2h), \cdots, x_T((N-1)h)) \\ &= (x(0), x(h), x(2h), \cdots, x(T/2 - h), x(-T/2), x(-T/2 + h), \cdots, x(-h)) \end{aligned}$$

(que se obtiene de v desplazando a la derecha la primera mitad del vector y dejando al principio la segunda mitad), entonces la dft de w, que representamos por $dft(w) = (y_0, \cdots, y_{N-1})$, verifica que

$$hy_k \simeq T c_k(x_T) \simeq X(k/T). \tag{5.5}$$

En particular, esto significa que una reorganización de la entrada más un reescalado de la dft nos proporciona información sobre el comportamiento en frecuencias de la señal analógica $x(t)$.

Ahora bien: algunas observaciones son aún necesarias.

Para empezar, cuando se realizan los cálculos con la dft debemos saber que estamos interpretando los vectores como sucesiones periódicas de periodo $N =$ longitud del vector y que esto se hace tanto en el dominio del tiempo como en el de las frecuencias, lo que permite considerar frecuencias positivas y negativas. La frecuencia cero (llamada, en el caso de los potenciales eléctricos,

componente de corriente continua) viene dada por y_0. Es decir, $hy_0 \simeq X(0)$. El resto de frecuencias se consideran múltiplos de la frecuencia fundamental, que denotamos por f_1 y que hacemos coincidir con y_1, adecuadamente re-escalado. Es decir, hy_1 es la aportación a la señal de la frecuencia fundamental, que colocamos en la posición f_1 del eje de las frecuencias. Es decir, $hy_1 \simeq X(f_1)$. A continuación, las frecuencias re-escaladas $hy_2, hy_3, \cdots$ hasta $hy_{N/2}$ se corresponden con las contribuciones de las frecuencias $2f_1, 3f_1, \cdots, (N/2)f_1$. Finalmente, los valores hy_{N-k} se corresponden con las contribuciones de las frecuencias $-kf_1$ para $k = 1, 2, \cdots, N/2$.

Para poder dibujar cada aportación en frecuencias en el sitio correcto es, por tanto, necesario determinar el valor preciso de la frecuencia fundamental f_1. Para ello se tiene en consideración el teorema del muestreo. Como el periodo de muestreo es h, la frecuencia de muestreo es $f_s = 1/h$ y, si queremos recuperar una señal con ancho de banda b, debe satisfacerse la desigualdad $f_s \geq 2b$, por lo que $b \leq \frac{1}{2h}$. Esto significa que la frecuencia más elevada que podemos pretender describir, si tomamos muestras con periodo de muestreo h, es $\frac{1}{2h}$. Además, las frecuencias debemos colocarlas equidistribuidas, debe haber N de ellas, y deben colocarse en un intervalo simétrico respecto del origen de coordenadas. Todas estas consideraciones conducen directamente a forzar las igualdades $-(N/2)f_1 = -1/(2h)$ y $(N/2)f_1 = 1/2(h)$. Es decir, $f_1 = 1/(Nh)$ y $f_k = k/(Nh)$ para $k = -N/2, -N/2+1, \cdots, -1, 1, 2, \cdots, N/2$. En el punto $f_k = k/(Nh)$ se debe colocar la aportación a las frecuencias asociada a y_k (que, recordemos, debe ir re-escalada y, por tanto, se corresponde en realidad con el valor hy_k) para $k = 0, 1, \cdots N/2$. Es decir, $hy_k \simeq X(kf_1) = X(f_k)$ y, finalmente, para $k = -1, -2, \cdots, -N/2$, el valor que debemos pintar en el punto f_k es $hy_{N-|k|} \simeq X(kf_1) = X(f_k)$

NOTA: Obsérvese que los cálculos anteriores son perfectamente coherentes con la fórmula (5.5), pues $Nh = T$ y, por tanto, $f_1 = \frac{1}{Nh} = \frac{1}{T}$, de modo que $hy_k \simeq X\left(\frac{k}{T}\right) = X\left(kf_1\right) = X\left(f_k\right)$.

Aunque con lo explicado hasta ahora un valor de la dft aproxima bastante bien al valor real de la transformada de Fourier analógica de la señal original, existen algunas limitaciones en los cálculos, vinculados a la naturaleza de la dft, que decribimos en lo que sigue.

Definición 5.6:

Llamamos **Aliasing** al fenómeno que se produce cuando usamos la serie cardinal que aparece en el Teorema del Muestreo para PW(b) en la reconstrucción de una señal $x(t) \notin PW(b)$ y que consiste básicamente en que la serie converge a una señal (llamada Alias) diferente a la señal original x(t).

El fenómeno Aliasing puede producirse o bien porque la señal x(t) es de banda limitada, pero su banda es mayor que b, por lo que estamos tomando menos muestras de las que realmente se necesitan para recuprar la señal o bien porque la señal no es de banda limitada.

El fenómeno de Aliasing se puede controlar esencialmente de dos modos: bien incrementando la frecuencia de muestreo, aproximandonos lo maximo posible a la frecuencia exigida por el teorema del muestreo. O bien filtrando la señal original con un filtro que se adapta a la velocidad del muestreo, y muestreamos la señal filtrada. Obviamente, ambos métodos se pueden combinar.

Por otra parte, aún en el caso en el que la señal original sea de banda limitada, y que hayamos muestreado con la frecuencia exigida en el teorema del muestreo se puede producir el siguiente fenómeno:

Definición 5.7:

Llamamos **derramamiento espectral** o **manchado espectral** al fenómeno que se produce como consecuencia de que el Teorema del Muestreo pide un conjunto infinito de muestras equiespaciadas y es algo que no podemos satisfacer debido a la naturaleza finita de los cálculos.

El derramamiento espectral se debe a que, al imponer un número finito de muestras, sustituimos la señal a la que se podría aplicar el teorema del muestreo por la que resulta de multiplicar la señal original por una función del tipo $z(t) = rect(\frac{t}{R})$ donde $rect(t) = 1$ para $0 \leq t \leq 1$ y $rect(t) = 0$ en el resto y esto puede producir discontinuidad de salto entre el principio y el final de la señal que vamos a muestrear, provocando que aparezcan, en el dominio de la frecuencia, componentes de altas frecuencias que antes no estaban. Además si tenemos en cuenta el principio de incertidumbre, en su versión cualitativa, concluimos que la nueva señal que estamos muestreando no puede ser de banda limitada. Dicho principio afirma lo siguiente:

Teorema 5.3:

Una señal $x(t) \in L^2(\mathbb{R})$ no puede ser simultáneamente de tiempo limitado y de banda limitada, a no ser que se trate de una señal idénticamente nula.

Cabe destacar que existen dos situaciones en las que el derramamiento espectral no se produce. Una de ellas es cuando la señal $x(t)$ tiene su soporte contenido en el soporte de la función $z(t)$. El otro caso es cuando $x(t)$ es periódica y la longitud R de la ventana definida mediante $z(t) = rect(\frac{t}{R})$ es un múltiplo entero del periodo. En el resto de casos el fenómeno del derramamiento espectral es inevitable y lo único que podemos hacer es reducirlo lo máximo posible.

Ahora veamos varios ejemplos prácticos de la teoría explicada en esta sección. Para ello hemos usado el código del Anexo E y las funciones que mostramos en cada uno de los ejemplos.

El programa del Anexo E funciona bien para funciones con ancho de banda finito, especialmente si sobremuestreamos, cosa que se realiza en la práctica de forma consciente. También es aplicable para señales que no son de banda limitada, especialmente si la aportación en altas frecuencias tiende rápidamente a cero en el infinito. Veámos un primer ejemplo.

Ejemplo 1:

Consideramos la función $sinc(x) = \begin{cases} \frac{sen(2\pi x)}{\pi x} & si \quad x \neq 0 \\ 1 & si \quad x = 0 \end{cases}$

y consideramos la función $rect(x) = \begin{cases} 0 & si \quad x < 0 \\ \frac{1}{2} & si \quad x = -1 \\ 1 & si \quad -1 < x < 1 \\ \frac{1}{2} & si \quad x = 1 \\ 0 & si \quad x > 1 \end{cases}$

Entre estas funciones existe la siguiente relación:

$$\begin{aligned}
\mathcal{F}(r)(\xi) &= \int_{-\infty}^{\infty} r(t)\cdot e^{-2\pi it\xi}dt \\
&= \int_{-\infty}^{\infty} 1\cdot e^{-2\pi it\xi}dt \\
&= \int_{-1}^{1} cos(2\pi t\xi)dt - i\int_{-1}^{1} sen(2\pi t\xi)dt \\
&= \int_{-1}^{1} \frac{cos(2\pi t\xi)\cdot 2\pi\xi}{2\pi\xi}dt - i\int_{-1}^{1} \frac{sen(2\pi t\xi)\cdot 2\pi\xi}{2\pi\xi}dt \\
&= \frac{1}{2\pi\xi}\left[sen(2\pi t\xi)\right]_{-1}^{1} - \frac{i}{2\pi\xi}\left[cos(2\pi t\xi)\right]_{-1}^{1} \\
&= \frac{1}{2\pi\xi}\left[2sen(2\pi\xi)\right] \\
&= \frac{2sen(2\pi\xi)}{2\pi\xi}.
\end{aligned}$$

Se sigue que $r(t) = \mathcal{F}^{-1}\left(\mathcal{F}(r)\right)(t) = \mathcal{F}^{-1}\left(\frac{2sen(2\pi\xi)}{2\pi\xi}\right)(t) = \mathcal{F}\left(\frac{2sen(2\pi\xi)}{2\pi\xi}\right)(-t)$. Por tanto, como $r(t)$ es par, $\mathcal{F}\left(\frac{2sen(2\pi\xi)}{2\pi\xi}\right)(t) = r(-t) = r(t)$.

Para aplicar el programa del Anexo E, necesitamos una función de banda limitada que tome valores reales, por lo que podemos considerar como función analógica la función $sinc(t)$ y como su transformada de Fourier inversa la función $rect(\xi)$. La función $rect(\xi)$ tiene soporte compacto, $[-1,1]$, por lo que la función $sinc(t)$ es una función de banda limitada, su ancho de banda es $b = 1$.

Para introducir estas funciones en Matlab usamos el siguiente código:

```
//definimos la función sinc
function y=rectangulof(t)
y= 2.*sinc(2.*t);
```

Función sinc

```
//definimos la transformada de sinc
function y=rectangulo(t)
y= (sign(t+1)+1)/2-(sign(t-1)+1)/2;
```

Transformada de Fourier de la función sinc

donde la función $sign(x)$, ya está implementada en Matlab de la siguiente forma:

$$sign(x) = \begin{cases} 1 & si \quad x > 0 \\ 0 & si \quad x = 0, \\ 0 & si \quad x < 0 \end{cases}$$

y la función $sinc(x)$ también está implementada.

Introduciendo como hemos explicado estas funciones en el código del Anexo E y eligiendo el periodo de muestreo (h) y el intervalo en el que deseamos que se tomen las muestras (L), por ejemplo, $h = 0{,}1$ y $L = 10$ obtenemos las siguientes gráficas:

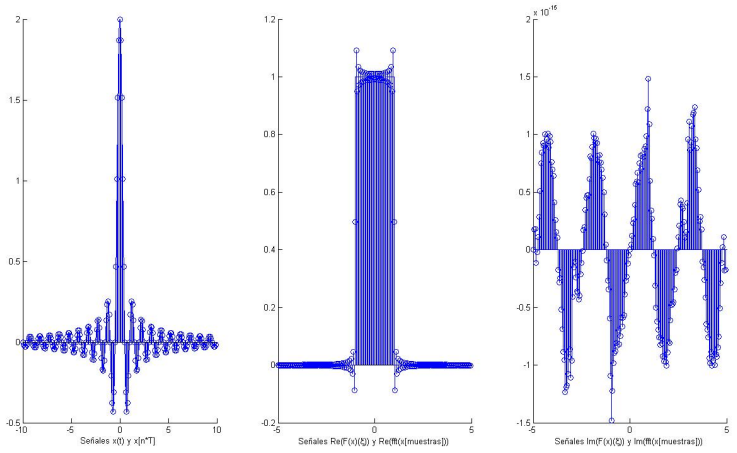


Figura 5.1: Salida del programa al introducir como función analógica la función $sinc(x)$, tomando periodo de muestreo $h = 0{,}1$ en el intervalo $[-10, 10]$.

En la Figura 5.1 mostramos tres gráficas, en la primera gráfica se ve la señal $sinc(t)$ pintada con trazo continuo y las muestras que se han tomado de la función en el intervalo de muestreo simétrico, $[-10, 10]$. En la segunda gráfica se ve la parte real de la transformada de Fourier inversa de la función $sinc(t)$ pintada con trazo continuo en el intervalo $[-5, 5] = [\frac{-1}{2 \cdot 0{,}1}, \frac{1}{2 \cdot 0{,}1}]$. Además también podemos ver los valores que se han calculado para la parte real de dicha transformada y que se aproximan muy bien al trazo continuo. Por último, en la tercera gráfica tenemos lo mismo pero para la parte imaginaria de la transformada, que es nula. Podemos decir que se recupera bastante bien la descripción de la señal.

Si en el mismo código que acabamos de usar para hacer este ejemplo, introducimos como función analógica la función $rect(x)$, la cual no es una función de banda limitada ya que su transformada de Fourier es la función $sinc(x)$, que tiene soporte en toda la recta real, con los mismos parámetros, los resultados que obtenemos son las siguientes:

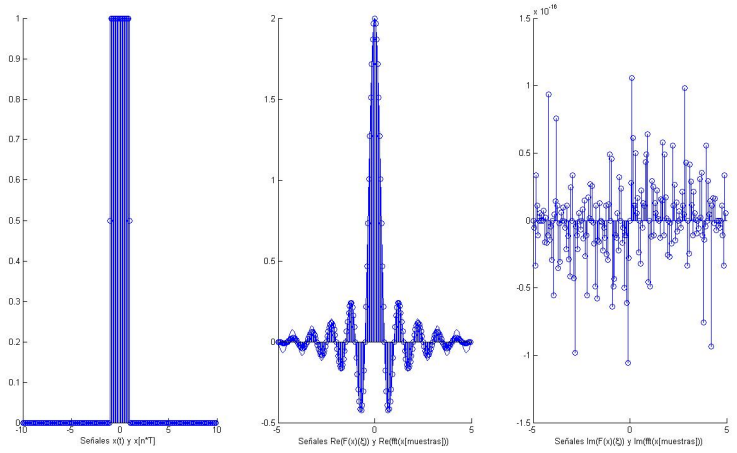


Figura 5.2: Salida del programa al introducir como función analógica $rect(x)$ y su transformada, tomando $h = 0{,}1$ y $L = 10$.

donde podemos ver que la parte real de la aproximación es mejor que en el caso anterior. Esto se debe a que la función $rect(x)$ es más fácil de integrar, aunque no sea de banda limitada.

De hecho, como ya hemos observado anteriormente, el programa del Anexo E funciona con señales que no son de banda limitada, pero que su transformada de Fourier tiende rápidamente a cero en el infinito. Esto sucede en el siguiente ejemplo.

Ejemplo 2:

Consideramos la función $s(t) = \begin{cases} \frac{1}{2}e^{-t} & si \quad t \geqslant 0 \\ 0 & si \quad t < 0 \end{cases}$

Para introducir esta función en Matlab necesitamos el siguiente código

```
//definimos la función
function y=expostep(t)
y=(1/2).*(1+sign(t)).*exp(-t);
```

Función $s(t)$

donde la función sign es la definida en el ejemplo anterior. La transformada de Fourier de la función $s(t)$ es:

$$\begin{aligned}
\mathcal{F}(s)(\xi) &= \int_{-\infty}^{\infty} s(t)\cdot e^{-2\pi i t\xi}dt \\
&= \int_{0}^{\infty} \tfrac{1}{2}e^{-t}\cdot e^{-2\pi i t\xi}dt \\
&= \tfrac{-1}{2}\int_{0}^{\infty} \frac{e^{-t\cdot(1+2\pi i\xi)}(-(1+2\pi i\xi))}{(1+2\pi i\xi)}dt \\
&= \frac{-1}{2(1+2\pi i\xi)}\left[e^{-t(1+2\pi i\xi)}\right]_0^{\infty} \\
&= \frac{-1}{2(1+2\pi i\xi)}\left[0-1\right] \\
&= \frac{1}{2(1+2\pi i\xi)} \\
&= \frac{1\cdot(1-2\pi i\xi)}{2(1+2\pi i\xi)\cdot(1-2\pi i\xi)} \\
&= \frac{1-2\pi i\xi}{2(1-2\pi i\xi+2\pi i\xi+4\pi^2\xi^2)} \\
&= \frac{1}{2(1+4\pi^2\xi^2)} - \frac{2\pi i\xi}{2(1+4\pi^2\xi^2)}.
\end{aligned}$$

La transformada de la función está definida en todo $\mathbb{R}$, por tanto no tiene sorporte compacto y la función no es de banda limitada. Pero $\mathcal{F}(s)(\xi)$ tiende rápidamente a cero en el infinito.

Para introducir la transformada en Matlab usamos el siguiente código:

```
//definimos la transformada
function y=expostepf(t)
y=complex((1+(2*pi.*t).^2).^(-1),-(2*pi.*t).*(1+(2*pi.*t).^2).^(-1));
```

Transformada de Fourier de la función $s(t)$

Introduciendo las dos funciones anteriores, con sus códigos correspondientes, en el código del Anexo E y eligiendo el periodo de muestreo (h) y el intervalo en el que deseamos que se tomen las muestras (L), por ejemplo, $h = 0{,}2$ y $L = 15$ obtenemos las siguientes gráficas:

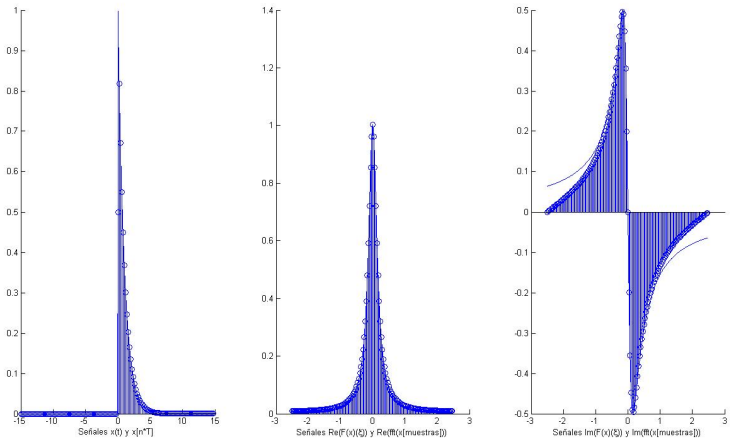


Figura 5.3: Salida del programa cuando introducimos como función analógica la función $(1/2)e^{-x}$ y su transformada de Fourier, con $h = 0{,}2, L = 15$.

En la Figura 5.3 mostramos tres gráficas, en la primera de ellas vemos la señal $s(t)$ pintada con trazo continuo y las muestras $s(ht)$ que hemos tomado. En la segunda gráfica se ve la transformada de Fourier de $s(t)$ pintada con trazo continuo en el intervalo $[-2{,}5, 2{,}5] = \left[-\frac{1}{0{,}4}, \frac{1}{0{,}4}\right]$, que es el intervalo más amplio del cual podemos obtener una descripción en frecuencias cuando el periodo de muestreo es $h = 0{,}2$. Además vemos los valores discretos que se han podido calcular para la parte real de dicha transformada con las muestras tomadas y que se aproximan muy bien al trazo continuo. En la gráfica tres, se representa la parte imaginaria de la transformada de Fourier de la función y los valores calculados con las muestras tomadas. También observamos que los valores discretos se aproximan bien al trazo continuo.

Ejemplo 3: Consideramos la función $g(x) = \begin{cases} 0 & si \quad x < \frac{-1}{2} \\ 1+2x & si \quad \frac{-1}{2} < x < 0 \\ 1 & si \quad x = 0 \\ 1-2x & si \quad 0 < x < \frac{1}{2} \\ 0 & si \quad x > \frac{1}{2} \end{cases}$

la introducimos en el programa con el siguiente código:

```
//definimos la función
function y=triangulo(t)
y= zeros(size(t));
for i = 1:length(t)
    if abs(t(i))>1/2;
          y(i) = 0;
    else
       y(i)=1-2*abs(t(i));
    end
end
```

Función triangulo

Consideramos también la transformada de Fourier de la función $g(t)$:

$$\begin{aligned}\mathcal{F}(g)(\xi) &= \int_{-\infty}^{\infty} g(t)\cdot e^{-2\pi i t\xi}dt \\ &= \int_{\frac{-1}{2}}^{0}(1+2t)\cdot e^{-2\pi i t\xi}dt + \int_{0}^{\frac{1}{2}}(1-2t)\cdot e^{-2\pi i t\xi}dt \\ &= \frac{1-e^{\pi i\xi}+\pi i\xi}{2\pi^2\xi^2} - \frac{-1+e^{-\pi i\xi}+\pi i\xi}{2\pi^2\xi^2} \\ &= \frac{2-\left(e^{\pi i\xi}+e^{-\pi i\xi}\right)}{2\pi^2\xi^2} \\ &= \frac{1-cos(\pi\xi)}{2\pi^2\xi^2}.\end{aligned}$$

Llamamos $G(\xi) = \frac{1-cos(\pi\xi)}{2\pi^2\xi^2}$.

Para inrtoducirla en Matlab usamos:

```
//definimos la transformada
function y=triangulof(t)
y= zeros(size(t));

for i = 1:length(t)
    if abs(t(i))<0.001
        y(i) = 1/2;
    else
        y(i)= (1-cos(pi.*t(i))).*(pi^2).^(-1).*t(i).^(-2);
    end
end
```

Transformada de Fourier de la función triángulo

En primer lugar, si introducimos en el algoritmo como función analógica la función $g(t)$ y como su transformada la función $G(t)$, donde $g(t)$ no es una función de banda limitada pero su transformda de Fourier, $G(t)$, tiende rápidamente a cero en el infinito. Debemos recuperar bien la señal en términos de frecuencias. Introducimos también los parámetros $h = 0{,}1$ y $L = 10$ y obtenemos:

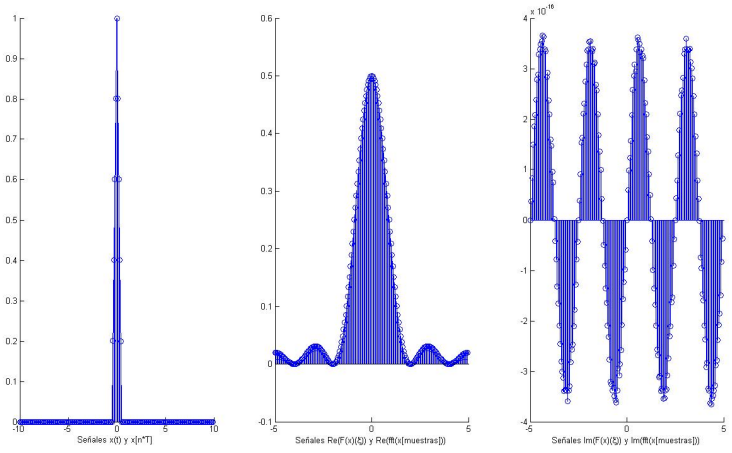


Figura 5.4: Salida del programa cuando introducimos como función analógica la función $g(t)$ y su transformada, con $h = 0{,}1$ y $L = 10$.

Observamos que, como esperabamos, la recuperación en frecuencias de la señal es buena ya que,

en la segunda gráfica vemos la parte real de la transformada de Fourier, donde la transformada de Fourier que se ha calculado con las muestras, se aproxima bastante a la la transformada de Fourier de la funcón en trazo continuo. Además la parte imaginaria de la transformada es practicamente nula.

En segundo lugar, introducimos como función analógica $G(\xi)$ y como transformada $g(t)$. Como $G(\xi)$ es una función de banda limitada finita, ya que $g(t)$ tiene soporte compacto en $[-1/2, 1/2]$, también se debe recuperar bastante bien la función en términos de frecuencias. Si introducimos los parámetros $h = 0{,}5$ y $L = 10$ obtenemos:

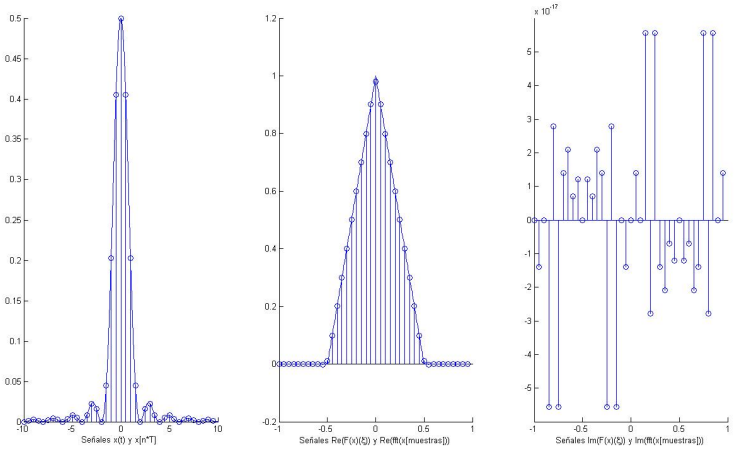


Figura 5.5: Salida del programa cuando introducimos como función analógica la función $G(t)$ y su transformada, con $h = 0{,}5$, $L = 10$.

Observamos que la recuperación es buena, ya que, como podemos ver, en la segunda gráfica, las transformada calculada a partir de las muestras tomadas se ajusta a la funcón triangulo, que como hemos visto es la transformda de la función $G(\xi)$, y por otro lado la parte imaginaria es prácticamente nula.

Los tres ejemplos anteriores corresponden o bien a funciones que están en el espacio de Paley-Wiener, con su ancho de banda correspodiente, o bien a funciones cuya transformada de Fourier converge rápido a cero en el infinito. Pero como hemos explicado anteriormente el Teorema del muestro se puede extender a polinomios trigonométricos, usando discribuciones. Veamos un ejemplo de aplicación de nuestro código para la recuperación en frecuencias de una señal que no decae a cero en el infinito.

Ejemplo 4:

Consideramos la función $f(t) = sen(2\pi t)$.

La introducimos en Matlab tecleando lo siguiente:

```
1 //definimos la función
  function y=seno(t)
3 y=sin(2.*pi.*t);
```

Funcion seno

Interpretamos la función $f(t)$ como una distribución temperada $u_{sen(2\pi t)}(\phi) = \int_{-\infty}^{\infty} sen(2\pi t)\phi(t)$. Para calcular su transformada de Fourier, aplicando la definición de la misma:

$$\begin{aligned}
\mathcal{F}(u_{sen(2\pi t)})(\phi) &= u_{sen(2\pi t)}(\mathcal{F}(\phi)) \\
&= \int_{-\infty}^{\infty} sen(2\pi t)\mathcal{F}(\phi)(\xi)d\xi \\
&= \int_{-\infty}^{\infty} \frac{e^{2\pi i\xi} - e^{-2\pi i\xi}}{2i}\mathcal{F}(\phi)(\xi)d\xi \\
&= i\left(\frac{-1}{2}\int_{-\infty}^{\infty} e^{2\pi i\xi}\mathcal{F}(\phi)(\xi)d\xi + \frac{1}{2}\int_{-\infty}^{\infty} e^{-2\pi i\xi}\mathcal{F}(\phi)(\xi)d\xi\right) \\
&= i\left(\left(\frac{-1}{2}\mathcal{F}^{-1}(\mathcal{F}(\phi)(1))\right) + \left(\frac{1}{2}\mathcal{F}^{-1}(\mathcal{F}(\phi)(-1))\right)\right) \\
&= i\left(-\pi\phi(1) + \pi\phi(-1)\right) \\
&= i(-\pi\delta_1 + \pi\delta_{-1})(\phi) \qquad \forall\phi \in \mathbb{S}.
\end{aligned}$$

Por tanto tenemos que $\mathcal{F}(u_{sen(2\pi t)}) = -i(-\pi\delta_1 + \pi\delta_{-1})$ que es una distribución con soporte compacto igual a $\{-1, 1\}$, por lo que la función $sen(2\pi t)$ tiene ancho de banda $b = 1$.

Introduciendo la función $f(t) = sin(2\pi t)$, en el código del Anexo E y eligiendo el periodo de muestreo (h) y el intervalo en el que deseamos que se tomen las muestras (L), por ejemplo, $h = 0{,}1$ y $L = 6$ obtenemos las siguientes gráficas:

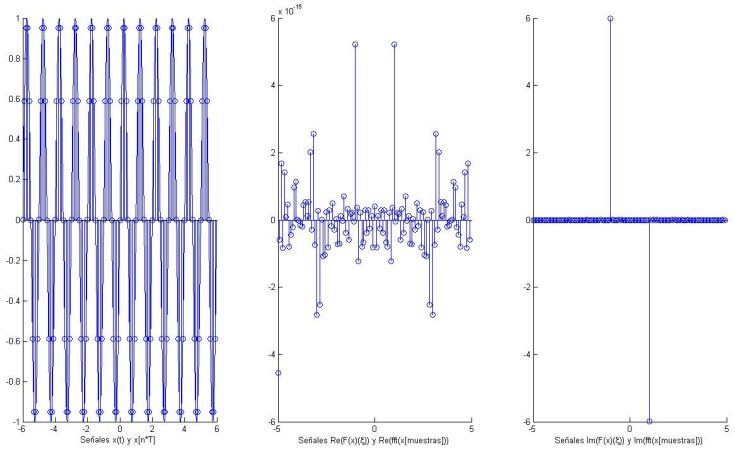


Figura 5.6: Salida del programa al introducir como función analógica la función $sen(2\pi x)$, con perido de muestreo $h = 0{,}1$.

En la primera gráfica se representa la función $f(t)$ y las muestras que se toman de ella, $f(ht)$. En la segunda gráfica la parte real de la transformada de Fourier, que es nula, tal y como esperabamos. Y en la última gráfica mostramos la parte imaginaria de la transformada de Fourier, que como era de esperar también, es nula en todos los puntos menos en 1 y -1 . Por tanto, con los valores que hemos tomado podemos decir que se recupera bien la representación en frecuencias de la función $f(t)$. Esto se debe a que el periodo de muestreo que hemos tomado $h = 0{,}1$, cumple el Teorema del muestreo, en el sentido de que la frecuencia más elevada que podemos pretender describir debe cumplir $b < \frac{1}{2h}$, en nuestro caso, $1 < \frac{1}{0{,}2} = 5$. Si por el contrario, tomamos como periodo de muestreo $h = 0{,}55$, como este no verificaría la condición anterior, ya que $1 \nleq \frac{1}{2\cdot 0{,}55}$. Entonces introduciendo nuevamente la función $sin(2\pi t)$ en el código del Anexo E y eligiendo el periodo de muestreo, $h = 0{,}55$ y el intervalo en el que deseamos que se tomen las muestras, $L = 6$, obtenemos las siguientes gráficas:

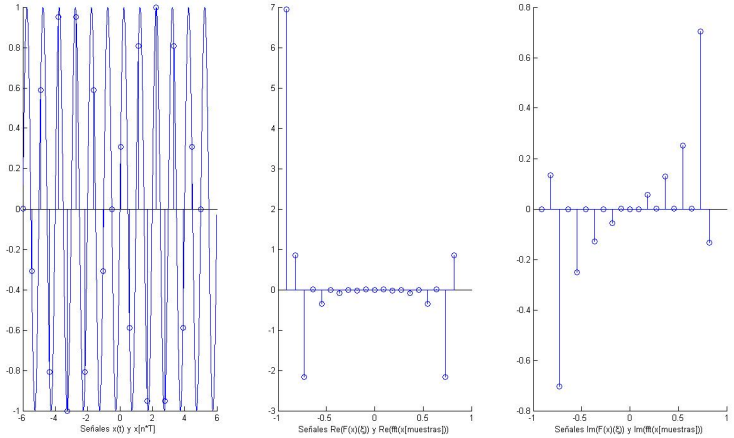


Figura 5.7: Salida del programa al introducir como función analógica la función $sen(2\pi x)$, con periodo de muestreo $h = 0{,}55$.

donde podemos observar que, como esperábamos, no se recupera bien la función en términos de frecuencias, ya que la parte real, que se representa en la segunda gráfica debería ser nula y no lo es. Y la parte imaginaria que se representa en la tercera gráfica debería ser nula en todos los puntos menos en el -1 y en el 1 y esto tampoco ocurre.

6 | Resultados

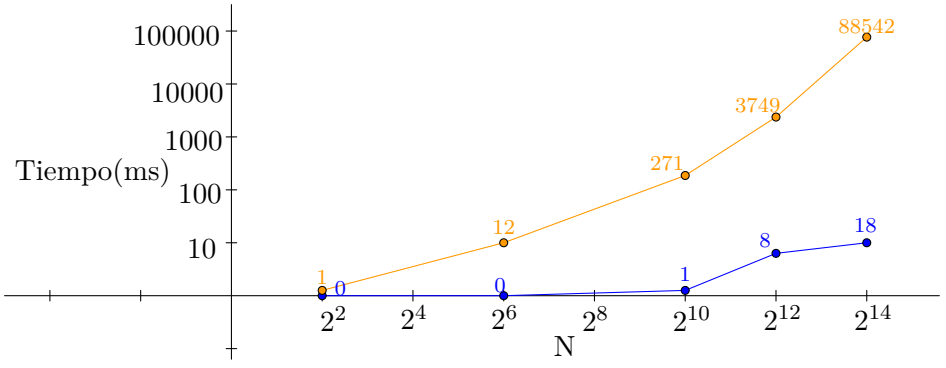
En este capítulo vamos a comprobar que los resultados de los programas que hemos implementado para el cálculo de la FFT cumplen nuestros objetivos, los cuales no solo eran que calculasen la transformada de Fourier discreta de un determinado vector, si no que además lo hiciesen en el menor tiempo posible. Los códigos que hemos implementado en Java están en el Anexo C.

Debido a que el tiempo de ejecución depende del ordenador en el que realizamos los cálculos, debemos de comentar que, los que mostramos a continuación se han realizado en un ordenador con un procesador de $1,4$ GHz Intel Core i5, con memoria 4 GB 1600 MHz DDR3 y con sistema operativo macOs High Sierra 10.13.5.

En primer lugar, si comparamos el tiempo en milisegundos que tarda en ser calculada computacionalmente la transformada de Fourier discreta, para distintos valores de N, de un determinado vector de entrada (donde las partes reales valen 1 si son menores que $\frac{N}{2}$; y 0 si no, las partes imaginarias son nulas), por definición con respecto al tiempo que tarda en ser calculada la transformada de Fourier discreta, del mismo vector de entrada, con alguno de los algoritmos FFT de radio 2 que hemos implementado ($DIFFFT_{NR}$), obtenemos la siguiente tabla:

N	2^2	2^6	2^{10}	2^{12}	2^{14}	2^{16}
FFT	0 ms	0 ms	1 ms	8ms	18 ms	36 ms
DFT	1 ms	12 ms	271 ms	3749 ms	88542 ms	+5 min

Representando gráficamente estos datos obtenemos la siguiente gráfica, donde en naranja se pintan los tiempos de ejecución en milisegundos de la FFT de radio 2, para los distintos valores de N y en azul los tiempos de ejecución en milisegundos de la DFT para los mismos valores de N.



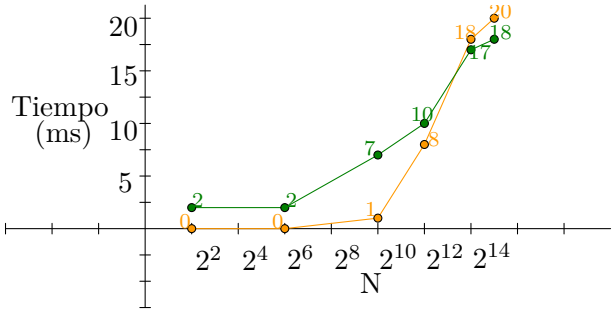
Observamos que si el tamaño del vector de entrada es $N \leqslant 2^{10}$ hay bastante diferencia de tiempo, pero para esos tamaños podríamos aplicar la DFT por definición, ya que el tiempo de ejecución

no es excesivo. Sin embargo para vectores con tamaño $N > 2^{10}$ la diferencia de tiempo es mucho mayor y además a partir de $N = 2^{16}$ el tiempo para calcular la transformada de Fourier discreta sin un algoritmo FFT es inasumible, mientras que con el algoritmo FFT se calcula muy velozmente. Podemos, por tanto, afirmar que hemos conseguido disminuir el tiempo que se necesita para calcular la transformada de Fourier discreta de un determinado vector de tamañano N, con N un número elevado, que era nuestro principal objetivo.

En segundo lugar, para mostrar que nuestros algoritmos funcionan de forma eficaz, hemos comprobado los resultados y el tiempo de ejecución de una determinada entrada (la misma que en el caso anterior), por un lado en nuestros programas y por otro lado al vector de entrada le hemos aplicado la FFT ya implementada en el programa Octave, cuyo código no es accesible (estos programas se suelen llamar cajas negras). Los resultados que hemos obtenido con respecto al tiempo de ejecución se muestran en la siguiente tabla:

N	2^2	2^6	2^{10}	2^{12}	2^{14}	2^{15}
FFT	0 ms	0 ms	1 ms	8 ms	18 ms	20 ms
FFT Octave	2 ms	2 ms	7 ms	10 ms	17 ms	18 ms

Representando gráficamente los datos de esta tabla obtenemos la siguiente gráfica, donde en naranja se pintan los tiempos de ejecución en milisegundos de la FFT de radio 2 (que hemos programado en Java), para los distintos valores de N y en verde los tiempos de ejecución en milisegundos de la FFT (ya programada en Octave) para los mismos valores de N,



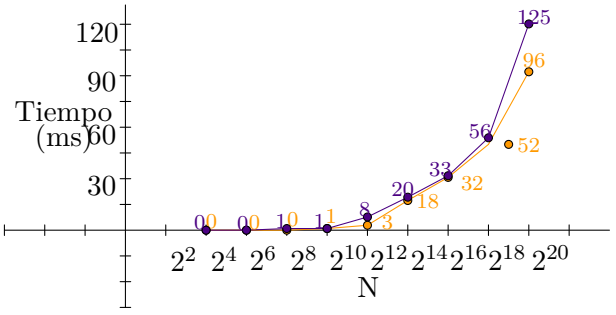
Observamos que los tiempos de ejecución son bastante parecidos, ya que se diferencian en apenas unos milisegundos. Podemos, por tanto, afirmar que los programas que hemos implementado son eficaces, al menos para las entradas probadas.

Ahora bien, cuando hemos estudiado los algoritmos FFT de radio 4, hemos dicho que el principal motivo de su desarrollo era disminuir el coste computacional de los algoritmos FFT de radio 2. Para ver si hemos logrado este objetivo, hemos comparado con determinadas entradas de distintos tamaños N, el tiempo de ejecución en milisegundos de los algoritmos FFT de radio 4 y FFT de radio 2 que hemos implementado.

El tiempo de ejecución en milisegundos, que dichos algoritmos han tardado en calcular la transformada de Fourier discreta de una determinada entrada de tamaño N (la misma que anteriormente), se muestra en la siguiente tabla:

N	2^4	2^6	2^8	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
FFT radio 4	0 ms	0 ms	0 ms	1 ms	3 ms	18 ms	32 ms	52 ms	96 ms
FFT radio 2	0 ms	0 ms	1 ms	1 ms	8 ms	20 ms	33 ms	56 ms	125 ms

De nuevo representando gráficamente los datos de la tabla obtenemos la gráfica que se muestra a continuación, donde en naranja se pintan los tiempos de ejecución en milisegundos de la FFT de radio 4, para los distintos valores de N y en morado los tiempos de ejecución en milisegundos de la FFT de radio 2 para los mismos valores de N.

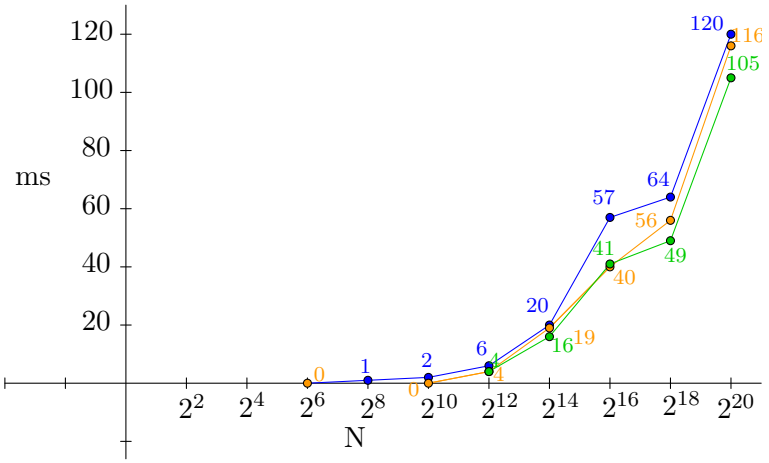


Como podemos observar, el tiempo de ejecución del algoritmo FFT de radio 4 es menor que el del algoritmo FFT de radio 2 para un determinado vector, además cuanto más grande es N, mayor es esta diferencia de tiempos de ejecución. Por tanto, podemos afirmar que también hemos cumplido nuestro objetivo con la implementación del algoritmo FFT de radio 4, al menos para las entradas probadas.

Por último, si comparamos el tiempo de ejecución de los códigos FFT de radios 2, 4 y split, para la función de entrada donde tanto la parte real como la parte imaginaria toman valores aleatorios entre $(-\frac{1}{2}, \frac{1}{2})$. Obtenemos la siguiente tabla:

N	2^2	2^4	2^6	2^8	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
FFT radio 2	0 ms	0 ms	0 ms	1 ms	2 ms	6 ms	20 ms	57 ms	64 ms	120 ms
FFT radio 4	0 ms	0 ms	0 ms	0 ms	4 ms	4 ms	19 ms	40 ms	56 ms	116 ms
FFT radio split	0 ms	0 ms	0 ms	0 ms	0 ms	4 ms	16 ms	41 ms	49 ms	105 ms

Representando gráficamente estos datos, donde en color azul se representan los tiempos de ejecución en milisegundos para los distintos tamaños de los vectores de entrada con el algoritmo FFT de radio 2, en naranja con el algoritmo FFT de radio 4 y en verde con el algoritmo FFT de radio split.



Aunque la direfencia de los tiempos de ejecución de estos algoritmos para los valores que hemos comparado no es muy grande, ya que se diferencian en unos milisegundos. Podemos observar que en general el tiempo de ejecución del algoritmo FFT de radio 4 es menor que el tiempo de ejecución del algoritmo FFT de radio 2 y que el algoritmo FFT de radio split tiene menor tiempo de ejecución que el algoritmo FFT de radio 4, al menos para los vectores introducidos.

7 | Anexo A : resolución generalizada de ecuaciones de recurrencia

Los contenidos de este anexo hacen referencia al libro [4].

Teorema 7.1:

Si el tiempo de ejecución de un algoritmo satisface

$$T(N) = \begin{cases} aT(\frac{N}{c}) + bN & \text{si } N = c^k > 1, \\ b & \text{si } N = 1 \end{cases}$$

donde $a \geq 1$, $c \geq 2$ y $b > 0$ son constantes dadas, y N es el tamaño de la entrada del algoritmo, entonces el orden de complejidad del algoritmo viene dado por:

$$T(N) = \begin{cases} \frac{bn}{c-a} \cdot N & \text{si } a < c, \\ bN \cdot \log_c N + \mathrm{O(N)} & \text{si } N = 1 \\ \frac{ab}{a-c} \cdot N^{\log_c a} & \text{si } a > c \end{cases}$$

Demostración. Asumimos que $N = c^k > 1$ con $k \in \mathbb{N}$. Entonces:

$$\begin{aligned} T(N) &= aT\left(\tfrac{N}{c}\right) + bN \\ &= a\left[aT\left(\tfrac{N}{c^2}\right) + b\tfrac{N}{c}\right] + bN \\ &= a^2T\left(\tfrac{N}{c^2}\right) + ab\tfrac{N}{c} + bN \\ &= a^2\left[aT\left(\tfrac{N}{c^3}\right) + b\tfrac{N}{c^2}\right] + ab\tfrac{N}{c} + bN \\ &= a^3T\left(\tfrac{N}{c^3}\right) + a^2b\left(\tfrac{N}{c^2}\right) + ab\left(\tfrac{N}{c}\right) + bN \\ &\vdots \\ &\vdots \\ &= a^kT\left(\tfrac{N}{c^k}\right) + a^{k-1}b\left(\tfrac{N}{c^{k-1}}\right) + a^{k-2}b\left(\tfrac{N}{c^{k-2}}\right) + \ldots + ab\left(\tfrac{N}{c}\right) + bN. \end{aligned}$$

Como $N = c^k$

$$\begin{aligned} T(N) &= a^kT(1) + a^{k-1}bc + a^{k-2}bc^2 + \ldots + abc^{k-1} + bc^k \\ &= a^k \cdot b + a^k \cdot b\left(\tfrac{c}{a}\right) + a^k \cdot b\left(\tfrac{c}{a}\right)^2 + \ldots + a^k \cdot b\left(\tfrac{c}{a}\right)^{k-1} + a^k \cdot b\left(\tfrac{c}{a}\right)^k \\ &= a^kb\textstyle\sum_{i=0}^{k}\left(\tfrac{c}{a}\right)^i. \end{aligned} \tag{7.1}$$

Distinguimos tres casos:

- Si $a = c$ entonces $\frac{c}{a} = 1$ y $N = c^k = a^k$, por lo que $k = \log_c N$.
  Por tanto la ecuación (7.1) se convierte en:

$$\begin{aligned} T(N) &= a^k b \sum_{i=0}^{k} \left(\frac{c}{a}\right)^i \\ &= Nb \sum_{i=0}^{k} 1 \\ &= Nb(k+1) \\ &= Nb(\log_c N + 1) \\ &= Nb \log_c N + Nb \\ &= Nb \log_c N + O(N). \end{aligned}$$

- Si $a > c$, entonces $\frac{c}{a} < 1$ y $\lim_{i\to\infty} \left(\frac{c}{a}\right)^i = 0$. Como $N = c^k$, tenemos $k = \log_c N$.
  Entonces la ecuación (7.1) se convierte en:

$$T(N) = a^k b \sum_{i=0}^{k} \left(\frac{c}{a}\right)^i.$$

  Usando la suma geométrica, es decir, usando que $\sum_{k=0}^{n-1} x^k = \frac{1-x^n}{1-x}$ si $x \neq 1$

$$\begin{aligned} T(N) &= a^k b \frac{1-\left(\frac{c}{a}\right)^{k+1}}{1-\left(\frac{c}{a}\right)} \\ &= a^k b \frac{1-\left(\frac{c}{a}\right)^{k+1}}{(a-c)/a} \\ &= a^k b \frac{a-c\left(\frac{c}{a}\right)^k}{a-c}. \end{aligned}$$

  Ahora bien, para k suficientemente grande

$$T(N) \approx a^k b \left(\frac{a}{a-c}\right) = a^{\log_c N} b \left(\frac{a}{a-c}\right) = \frac{ab}{a-c} N^{\log_c a}.$$

- Si $c > a$, entonces $\frac{a}{c} < 1$ y $\lim_{i\to\infty} \left(\frac{a}{c}\right)^i = 0$.
  Entonces la ecuación (7.1) se convierte en:

$$\begin{aligned} T(N) &= a^k b \sum_{i=0}^{k} \left(\frac{c}{a}\right)^i \\ &= a^k b + a^k b \left(\frac{c}{a}\right) + a^k b \left(\frac{c}{a}\right)^2 + \ldots + a^k b \left(\frac{c}{a}\right)^{k-1} + a^k b \left(\frac{c}{a}\right)^k \\ &= a^k b + a^{k-1} bc + a^{k-2} bc^2 + \ldots + abc^{k-1} + bc^k \\ &= c^k b \left(\frac{a}{c}\right)^k + c^k b \left(\frac{a}{c}\right)^{k-1} + \ldots + c^k b \left(\frac{a}{c}\right) + c^k b \\ &= c^k b \sum_{i=0}^{k} \left(\frac{a}{c}\right)^i. \end{aligned}$$

  Usando la fórmula de la suma geométrica, como se hizo anteriormente,

$$T(N) = Nb \left(\frac{1 - \left(\frac{a^{k+1}}{c^c}\right)}{1 - \frac{a}{c}}\right).$$

  Ahora bien, tomando k suficientemente grande,

$$T(N) \approx Nb \left(\frac{c}{c-a}\right) = N \frac{bc}{c-a}.$$

□

8 | Anexo B : cálculo y almacenamiento de las raíces de la unidad

En todos los capítulos donde hemos hecho uso de las raíces de la unidad hemos supuesto que las tenemos correctamente calculadas y almacenadas para su uso. En este anexo vamos a ver como realizamos los cálculos y almacemos las raíces correctamente.

8.1. Raíces algoritmo radio 2

Los contenidos de esta sección hacen referencia al libro [4].

Dado N una potencia de dos, y recordando que

$$\overline{w_N}^r = e^{-ir\theta} = cos(r\theta) - isen(r\theta)$$

con $\theta = \frac{2\pi}{N}$, donde cada $\overline{w_N}^r$ es una raíz compleja de la unidad, que como sabemos son simétricas y están igualmente espaciadas en el círculo unidad. Por lo tanto, si $a + ib$ es una raíz también lo serán $\pm a \pm ib$ y $\pm b \pm ia$.

Haciendo uso de esta propiedad, sólo necesitamos calcular los primeros $\frac{N}{8} - 1$ valores, es decir, tenemos que calcular, $\overline{w_N}^r$ para $r = 1, ..., \frac{N}{8} - 1$, teniendo en cuenta que para $r = 0$ se tiene que $\overline{w_N}^0 = 1$.

Para calcular $\overline{w_N}^r$ para $r = 0, 1, ..., \frac{N}{8} - 1$, tenemos que calcular los valores de $cos(r\theta)$ y $sen(r\theta)$ para $r = 0, 1, ..., \frac{N}{8} - 1$, para ello vamos a utilizar las siguientes funciones trigonométricas:

$$cos(a + b) = cos(a)cos(b) - sen(a)sen(b). \tag{8.1}$$

$$cos^2(a) + sen^2(a) = 1. \tag{8.2}$$

$$sen(a + b) = sen(a)cos(b) + sen(b)cos(a). \tag{8.3}$$

Tomando $\theta = \frac{2\pi}{N}$ como antes, podemos calcualar $cos((r+1)\theta)$ en términos de $cos(r\theta)$ y $sen(r\theta)$ de la siguiente forma:

$$cos((r + 1)\theta) = cos(r\theta + \theta) = cos(r\theta)cos(\theta) - sen(r\theta)sen(\theta)$$

donde hemos usado la igualdad (8.1).

Llamando $S := sen(\theta)$ y usando la igualdad (8.2) tenemos que $cos(\theta) = \sqrt{1 - S^2} := C$, con S y C constantes, entonces:

$$cos((r+1)\theta) = cos(r\theta) \cdot C + sen(r\theta) \cdot S.$$

Ahora usando las mismas constantes, C y S podemos calcular $sen((r+1)\theta)$ de nuevo en términos de $cos(r\theta)$ y $sen(r\theta)$ de la siguiente forma:

$$sen((r+1)\theta) = sen(r\theta + \theta) = sen(r\theta)cos(\theta) + cos(r\theta)sin(\theta) = sen(r\theta) \cdot C + cos(r\theta) \cdot S,$$

donde hemos usado la igualdad (8.3).

Para calcular $\overline{w_N}^r$ para $r = \frac{N}{8} - 1, ..., \frac{N}{4} - 1$, hemos hecho uso de:

$$cos\left(\frac{2\pi\left(\frac{N}{8} + r\right)}{N}\right) = -sen\left(\frac{2\pi\left(\frac{N}{8} + r\right)}{N}\right)$$
$$sen\left(\frac{2\pi\left(\frac{N}{8} + r\right)}{N}\right) = -cos\left(\frac{2\pi\left(\frac{N}{8} + r\right)}{N}\right),$$

para $r = 1, ..., \frac{N}{8} - 1$.

Ahora para calcular $\overline{w_N}^r$ para $r = \frac{N}{4} - 1, ..., N - 1$, hemos hecho uso de:

$$cos\left(\frac{2\pi\left(\frac{N}{4} + r\right)}{N}\right) = -cos\left(\frac{2\pi\left(\frac{N}{4} - r\right)}{N}\right)$$
$$sen\left(\frac{2\pi\left(\frac{N}{4} + r\right)}{N}\right) = sen\left(\frac{2\pi\left(\frac{N}{4} - r\right)}{N}\right),$$

para $r = 1, ..., \frac{N}{4} - 1$.

Por último, una vez que tenemos calculadas todas las raíces que necesitamos, como nuestro objetivo es conseguir que los algoritmos sean lo más rápido posible, estos deben acceder a los datos de la forma más eficiente. Para ello, colocamos las raíces (repitiendo las necesarias de las que ya tenemos calculadas) de forma que el algoritmo tenga que recorrer una única vez el array de principio a fin.

La función que hemos implementado para ello es la siguiente:

```
1
3   /**
      * La funcion raicesReduccion calcula las raices de la unidad necesarias
      * para aplicar un algoritmo FFT NR, es decir, los w_{N}^{r} conjugados
5     *
      * @param k 2^{k}
7     * @return W contiene las raices ordenadas en el orden que las vamos a
      * necesitar en el algoritmo NR
9     */
     public static double[] raicesReduccion(int k) {
11       int n = (int) Math.pow(2, k);
```

```
13          int n2 = n / 2;
            int n4 = n2 / 2;
15          int n8 = n4 / 2;

17          double[] W = new double[2 * n - 2];
            double arg = - 2 * Math.PI / n;
19          double S = Math.sin(arg), C = Math.sqrt(1 - S * S);

21          //calculamos las N/2 primeras raices
           W[0] = 1;
23         W[1] = 0;
            if (n >= 3) {
25              W[2] = C;
                W[3] = S;
27          }
            double ck = C;
29          double sk = S;
            for (int i = 2; i < n8; i++) {
31              double m1 = (ck + sk) * C;
                double m2 = (S + C) * sk;
33              double m3 = (S - C) * ck;
                ck = m1 - m2;
35              sk = m1 + m3;
                W[2 * i] = ck; // cos((i-1) tita) * C - sen ((i-1) tita) * S
37              W[2 * i + 1] = sk; // sen((i-1) tita) * C - cos ((i-1) tita) * S

39          }

41          if (n8 > 1) {
                W[2 * n8] = Math.sqrt(2) / 2;
43              W[2 * n8 + 1] = -Math.sqrt(2) / 2;
                for (int i = 1; i <= n8 - 1; i++) {
45
                    W[2 * (n8 + i)] = -W[2 * (n8 - i) + 1];
47                  W[2 * (n8 + i) + 1] = -W[2 * (n8 - i)];
                }
49          }

51          if (n4 > 1) {
                W[2 * n4 + 1] = -1;
53              for (int i = 1; i <= n4 - 1; i++) {

55                  W[2 * (n4 + i)] = -W[2 * (n4 - i)];

57                  W[2 * (n4 + i) + 1] = W[2 * (n4 - i) + 1];

59              }
            }
61          int c = n - 1;

63          int t = 0;
            //repetimos las raices de dos en dos
65          for (int i = 0; i < c - 1; i = i + 2) {
                W[i + n] = W[t];
67              W[n + i + 1] = W[t + 1];
                t = t + 4;
69
            }
71
```

```
        return W;
73  }
```

Función para calcular las raíces de la unidad necesarias en el algoritmo FFT de radix 2 con entrada en orden natural y salida en bitreversal.

NOTA : Para el algoritmo FFT de radio dos inverso las raíces se calculan de forma análoga, teniendo en cuenta que el algoritmo de arriba las raíces son las conjugadas, por tanto lo único que supone es una cambio de signo en la parte imaginaria (el signo del seno). Para los algoritmos $DIF_{RN}FFT$ también se calculan de forma análoga pero con las raíces colocadas en orden bit-reversal.

8.2. Raíces algoritmo radio 4

Dado N una potencia de cuatro. De nuevo como ocurre con las raíces de la unidad necesarias para aplicar algoritmos de radio 2, sólo necesitamos calcular los $\frac{N}{8}$ primeros valores, es decir, las primeras $\overline{w_N}^r$ para $r = 1, ..., \frac{N}{8} - 1$, teniendo en cuenta que $\overline{w_N}^0 = 1$.

Para calcular las primeras raíces hemos utilizado que $\overline{w_N}^r = \overline{w_N}^{r-1} \cdot \overline{w_N}^1$. Haciendo uso de estos valores, cálculamos las raíces $\overline{w_N}^r$ para $r = \frac{N}{8}, ..., \frac{N}{4} - 1$, teniendo en cuenta que $\overline{w_N}^{\frac{N}{8}+k} = \overline{w_N}^{\frac{N}{8}} \cdot \overline{w_N}^k$ Para el cálculo de las raíces de la unidad al cuadrado con superíndices $r = 0, 1, ..., \frac{N}{8} - 1$, lo que hacemos es copiar las raíces que ya tenemos cálculadas con índices pares. Para el cálculo de $\overline{w_N}^{2r}$ para $r = \frac{N}{8}, ..., \frac{N}{4} - 1$, multiplicamos por $-i$, es decir usamos que, $\overline{w_N}^{2r} = -i \cdot \overline{w_N}^{2(\frac{N}{4}+r)}$.

Ahora bien, para el cálculo del resto de raíces de la unidad y sus cuadrados, hemos usando por una lado que $(\overline{w_N}^r)^2 = \overline{w_N}^{2r}$ y por otro:

$$cos\left(\frac{4[\frac{N}{2} + (r + \frac{N}{32})]}{N}\right) = sen\left(\frac{4\pi[\frac{N}{2} + r]}{N}\right)$$

$$sen\left(\frac{4[\frac{N}{2} + (r + \frac{N}{32})]}{N}\right) = -cos\left(\frac{4\pi[\frac{N}{2} + r]}{N}\right),$$

para $r = 0, .., \frac{N}{32} - 1$.

Por último, para el cáculo de las raíces de la unidad al cubo, lo único que hemos usado es que $\overline{w_N}^{3r} = \overline{w_N}^{2r} \cdot \overline{w_N}^r$.

Como nuestro objetivo es que los algoritmos sean lo más rápido posible, estos deben acceder a los datos de forma eficiente. Por tanto en la función que hemos implementado, las raíces están colocadas en tres vectores en el orden que van a ser necesarias, para que en el cálculo de las raíces el algoritmo recorra una única vez de principio a fin cada array.

El algoritmo que hemos implementado es en siguiente:

```
1
    /**
3    * La funcion raices4k1 calcula las raices de la unidad necesarias para
     * aplicar una FFT de radio cuatro con entrada en orden natural y salida en
5    * orden bit reversal.
     *
7    * @param k N=4^{k}
     * @return w1 con las raices primitivas de la unidad, es decir, w_{n}^{r}
```

```
     * para n = N, N/4,... y 0 menor= r menor= N/4 -1 ; w2 con las raices
     * primitivas a de la unidad al cuadrado es decir, w_{n}^{2r} para n = N,
     * N/4,... y 0 menor= r menor= N/4 -1 y w3 con las raices primitivas a de la
     * unidad al cubo, es decir, w_{n}^{3r} para n = N, N/4,... y 0 menor= r
     * menor= N/4
     */
    public static double[][] raices4k1(int k) {

        double seno = Math.sin(2 * Math.PI / Math.pow(4, k));
        double coseno = Math.sqrt(1 - seno * seno);
        int N = 1;

        //Calcular el tamaño del array
        for (int i = 0; i < k; i++) {
            N *= 4;
        }
        int NG = 0;
        int Naux = N / 4;
        while (Naux > 0) {
            NG = NG + Naux;
            Naux = Naux / 4;
        }
        NG = NG * 2;

        double[] w = new double[NG];
        double[] w2 = new double[NG];
        double[] w3 = new double[NG];
        double auxr;
        double auxi;

        //w[0]
        w[0] = 1;
        w2[0] = 1;
        w3[0] = 1;
        w[1] = 0;
        w2[1] = 0;
        w3[1] = 0;
        //w[1]
        if (NG > 4) {
            w[2] = coseno;
            w[3] = -seno;
            double r2 = Math.sqrt(2) / 2;

            //calculamos las N/8 primeras raices de la unidad
            w[N / 4] = r2;
            w[N / 4 + 1] = -r2;
            w2[N / 4] = 0;
            w2[N / 4 + 1] = -1;
            w3[N / 4] = -r2;
            w3[N / 4 + 1] = -r2;
            double delta, gamma, m1, m2, m3;
            for (int i = 2; i < N / 8; i++) {
                // w_i=w_(i-1)*w_1
                delta = -seno + coseno;
                gamma = -seno - coseno;
                m1 = (w[2 * i - 2] + w[2 * i - 1]) * coseno;
                m2 = delta * w[2 * i - 1];
                m3 = gamma * w[2 * i - 2];
                w[2 * i] = m1 - m2;
                w[2 * i + 1] = m1 + m3;
```

```
69              }
                //calculamos las primeras N/8 raices primitivas de la unidad al cuadrado
71              for (int i = 1; i < N / 8; i++) {
                    // w2_i=w_(2i)
73                  w2[2 * i] = w[4 * i];
                    w2[2 * i + 1] = w[4 * i + 1];
75                  //calculamos las primeras N/8 raices primitivas de la unidad al cubo
                    //w3 = w2*w1
77                  delta = w[2 * i + 1] + w[2 * i];
                    gamma = w[2 * i + 1] - w[2 * i];
79                  m1 = (w2[2 * i] + w2[2 * i + 1]) * w[2 * i];
                    m2 = delta * w2[2 * i + 1];
81                  m3 = gamma * w2[2 * i];
                    w3[2 * i] = m1 - m2;
83                  w3[2 * i + 1] = m1 + m3;
                    //calculamos las N/4 primeras raices
85                  //w_N/8+k = w_N/8*w_k
                    m1 = (r2 - r2) * w[2 * i];
87                  m2 = delta * (-r2);
                    m3 = gamma * r2;
89
                    w[N / 4 + 2 * i] = m1 - m2;
91                  w[N / 4 + 2 * i + 1] = m1 + m3;
                    //calculamos las N/4 primeras raices al cuadrado
93                  //w2_N/4+k
                    w2[N / 4 + 2 * i] = w2[2 * i + 1];
95                  w2[N / 4 + 2 * i + 1] = -w2[2 * i];
                    //calculamos las N/4 primeras raices al cubo
97                  //w3 = w2*w1
                    w3[N / 4 + 2 * i] = -r2 * (w3[2 * i] - w3[2 * i + 1]);
99                  w3[N / 4 + 2 * i + 1] = -r2 * (w3[2 * i + 1] + w3[2 * i]);
                }
101             //rellenamos el resto de raices con las que ya tenemos calculadas
                //(las N/2 primeras raices)
103             int l = N / 4;
                int punteroI = 0;
105             //indica por donde tenemos que empezar a rellenar
                int punteroS = N / 2;
107             w[punteroS] = 1;
                w[punteroS + 1] = 0;
109             w2[punteroS] = 1;
                w2[punteroS + 1] = 0;
111
                while (l > 4) {
113                 w[punteroS] = 1;
                    w[punteroS + 1] = 0;
115                 w2[punteroS] = 1;
                    w2[punteroS + 1] = 0;
117
                    for (int i = 1; i < l / 4; i++) {
119                     w[punteroS + 2 * i] = w2[punteroI + 4 * i];
                        w[punteroS + 2 * i + 1] = w2[punteroI + 4 * i + 1];
121
                    }
123                 w2[punteroS + l / 4] = 1;
                    w2[punteroS + l / 4 + 1] = 0;
125                 for (int j = 0; j < l / 8; j++) {
                        w2[punteroS + 2 * j] = w[punteroS + 4 * j];
127                     w2[punteroS + 2 * j + 1] = w[punteroS + 4 * j + 1];
                        //multiplicar por -i
```

```
                    w2[punteroS + 2 * (j + l / 8)] = w2[punteroS + 2 * j + 1];
                    w2[punteroS + 2 * (j + l / 8) + 1] = -w2[punteroS + 2 * j];
                }

                w3[punteroS + l / 4] = 1;
                w3[punteroS + l / 4 + 1] = 0;
                for (int i = 0; i < l / 4; i++) {
                    //w3 = w2*w1
                    delta = w[punteroS + 2 * i + 1] + w[punteroS + 2 * i];
                    gamma = w[punteroS + 2 * i + 1] - w[punteroS + 2 * i];
                    m1 = (w2[punteroS + 2 * i] + w2[punteroS + 2 * i + 1]) * w[
    punteroS + 2 * i];
                    m2 = delta * w2[punteroS + 2 * i + 1];
                    m3 = gamma * w2[punteroS + 2 * i];
                    w3[punteroS + 2 * i] = m1 - m2;
                    w3[punteroS + 2 * i + 1] = m1 + m3;
                }
                //actualizamos los punteros
                punteroI = punteroS;
                punteroS = punteroS + l / 2;
                l = l / 4;

            }
            //w_{N}^{0}
            w[NG - 2] = 1;
            w[NG - 1] = 0;
            //w_{N}^{2*0}
            w2[NG - 2] = 1;
            w2[NG - 1] = 0;
            //w_{N}^{3*0}
            w3[NG - 2] = 1;
            w3[NG - 1] = 0;
        }
        double[][] dw = {w, w2, w3};

        return dw;
    }
```

Función para calcular las raíces de la unidad necesarias en el algoritmo FFT de radio 4 con entrada en orden natural y salida en bitreversal

8.3. Raíces algoritmo radio split

```
public static double[][] raicesSplitk1(int k) {

        double seno = Math.sin(2 * Math.PI / Math.pow(2, k));
        double coseno = Math.sqrt(1 - seno * seno);
        int N = 1;

        //Calcular el tamaño del array: Primer paso N/4 raices, Segundo paso N/8...
        for (int i = 0; i < k; i++) {
            N *= 2;
        }
        int NG = 0;
        int Naux = N / 4;
        while (Naux > 0) {
            NG = NG + Naux;
```

```
            Naux = Naux / 2;
        }
        NG = NG * 2;
        double[] w = new double[NG];
        double[] w2 = new double[NG];
        double[] w3 = new double[NG];
        double auxr;
        double auxi;

        //w[0]
        w[0] = 1;
        w2[0] = 1;
        w3[0] = 1;
        w[1] = 0;
        w2[1] = 0;
        w3[1] = 0;
        //w[1]
        if (NG > 4) {
            w[2] = coseno;
            w[3] = -seno;

            double r2 = Math.sqrt(2) / 2;
            w[N / 4] = r2;
            w[N / 4 + 1] = -r2;
            w2[N / 4] = 0;
            w2[N / 4 + 1] = -1;
            w3[N / 4] = -r2;
            w3[N / 4 + 1] = -r2;
            double delta, gamma, m1, m2, m3;
            for (int i = 2; i < N / 8; i++) {
                // w_i=w_(i-1)*w_1
                delta = -seno + coseno;
                gamma = -seno - coseno;
                m1 = (w[2 * i - 2] + w[2 * i - 1]) * coseno;
                m2 = delta * w[2 * i - 1];
                m3 = gamma * w[2 * i - 2];
                w[2 * i] = m1 - m2;
                w[2 * i + 1] = m1 + m3;

                //w_N/4
                w[N / 4 + 2 * i] = -r2 * (m2 + m3);
                w[N / 4 + 2 * i + 1] = r2 * (2 * m1 + m3 - m2);

            }
            for (int i = 1; i < N / 8; i++) {
                // w2_i=w_(2i)
                w2[2 * i] = w[4 * i];
                w2[2 * i + 1] = w[4 * i + 1];

                //w3 = w2*w1
                delta = w[2 * i + 1] + w[2 * i];
                gamma = w[2 * i + 1] - w[2 * i];
                m1 = (w2[2 * i] + w2[2 * i + 1]) * w[2 * i];
                m2 = delta * w2[2 * i + 1];
                m3 = gamma * w2[2 * i];
                w3[2 * i] = m1 - m2;
                w3[2 * i + 1] = m1 + m3;

                //w_N/8+k
                m1 = (r2 - r2) * w[2 * i];
```

```
                m2 = delta * (-r2);
                m3 = gamma * r2;

                w[N / 4 + 2 * i] = m1 - m2;
                w[N / 4 + 2 * i + 1] = m1 + m3;

                //w2_N/4+k
                w2[N / 4 + 2 * i] = w2[2 * i + 1];
                w2[N / 4 + 2 * i + 1] = -w2[2 * i];

                //w3 = w2*w1
                w3[N / 4 + 2 * i] = -r2 * (w3[2 * i] - w3[2 * i + 1]);
                w3[N / 4 + 2 * i + 1] = -r2 * (w3[2 * i + 1] + w3[2 * i]);
            }

            int l = N / 4; //
            int punteroI = 0;
            int punteroS = N / 2; //
            w[punteroS] = 1;
            w[punteroS + 1] = 0;
            w2[punteroS] = 1;
            w2[punteroS + 1] = 0;

            while (l > 2) {
                w[punteroS] = 1;
                w[punteroS + 1] = 0;
                w3[punteroS] = 1;
                w3[punteroS + 1] = 0;

                for (int i = 1; i < l / 2; i++) {

                    w[punteroS + 2 * i] = w[punteroI + 4 * i];
                    w[punteroS + 2 * i + 1] = w[punteroI + 4 * i + 1];
                    w3[punteroS + 2 * i] = w3[punteroI + 4 * i];
                    w3[punteroS + 2 * i + 1] = w3[punteroI + 4 * i + 1];

                }
                punteroI = punteroS;
                punteroS = punteroS + l;
                l = l / 2;

            }
            w[NG - 2] = 1;
            w[NG - 1] = 0;

            w3[NG - 2] = 1;
            w3[NG - 1] = 0;
        }
        double[][] dw = {w, w3};
        return dw;
    }
```

Función para calcular las raíces de la unidad necesarias en el algoritmo FFT de radix split con entrada en orden natural y salida en bitreversal.

9 | Anexo C : Códigos FFT

En este anexo mostramos los códigos que hemos implementado en Java, para ello debemos de tener en cuenta que, en los códigos FFT implementados, hemos tratado cada una de las entradas de los códigos, que son complejas, como pares de números, por lo que en cada array las partes reales ocupan las posiciones pares y las partes imaginarias las posiciones impares.

9.1. DFT

```
1  public static Complex[] transformadaDiscreta(int N, Complex f[]) {
       Complex DFT[] = new Complex[n];
3      for (int j = 0; j < N; j++) {

5          DFT[j] = new Complex(0, 2 * Math.PI * j / N).exp();

7      }
       Complex T;
9      Complex Taux = new Complex(0, 0);
       Complex w[] = new Complex[n];
11     for (int k = 0; k < N; k++) {
           for (int n = 0; n < N; n++) {
13             w[n] = new Complex(0, 2 * Math.PI * n * k / N).exp();
               T = Complex.times(f[n], w[n]);
15             Taux = Complex.plus(Taux, T);

17         }
           DFT[k] = Taux;
19         Taux = new Complex(0, 0);
       }
21     return DFT;
   }
```

Algoritmo DFT

9.2. Algoritmo NR FFT radio 2

```
1
   /**
3      * La funcion radix2DIFNRFFT calcular la transformada del vector a con
       * entrada en orden natural y salida en orden reversal
5      *
       * @param N el tamaño de la muestra
7      * @param a array de tamaño 2*N debido a que en los huecos pares guardamos
```

```
     * la parte real de los datos y en los huecos impares la parte imaginaria de
     * los datos
     * @param w contiene las raices de la unidad
     * @return a que ahora contiene la transformada discreta de los valores que
     * antes contenia a
     */
    public static double[] radix2DIFNRFFT(int N, double a[], double[] w) {

        int NumOfProblems = 1;
        int ProblemSize = 2 * N;
        int Jtwiddle = 0;

        while (ProblemSize > 2) {

            int HalfSize = ProblemSize / 2;

            for (int k = 0; k < NumOfProblems; k++) {

                int JFirst = k * ProblemSize;
                int JLast = JFirst + HalfSize - 1;

                //recorrer las raíces de la unidad sin dar saltos pero las veces
                // que es necesario en el algoritmo
                Jtwiddle = 2 * (N - (N / NumOfProblems));

                for (int J = JFirst; J <= JLast; J = J + 2) {

                    double Tempr = a[J];
                    double Tempi = a[J + 1];
                    //realizamos la mariposa
                    //primera operacion
                    //rellenamos la parte real
                    a[J] = Tempr + a[J + HalfSize];
                    //rellenamos la parte imaginaria
                    a[J + 1] = Tempi + a[J + HalfSize + 1];
                    //segunda operacion
                    double delta = w[Jtwiddle + 1] + w[Jtwiddle];
                    double gamma = w[Jtwiddle + 1] - w[Jtwiddle];
                    double m1 = ((Tempr - a[J + HalfSize]) + (Tempi - a[J + HalfSize
    + 1])) * w[Jtwiddle];
                    double m2 = delta * (Tempi - a[J + HalfSize + 1]);
                    double m3 = gamma * (Tempr - a[J + HalfSize]);
                    //rellenamos la parte real
                    a[J + HalfSize] = m1 - m2;
                    //rellenamos la parte imaginaria
                    a[J + HalfSize + 1] = m1 + m3;
                    //recorremos el array de las raices
                    Jtwiddle = Jtwiddle + 2;
                }

            }
            NumOfProblems = 2 * NumOfProblems;
            ProblemSize = HalfSize;
        }

        return a;
    }
```

Algoritmo DIFNR FFT

9.3. Algoritmo RN FFT radio 2

```
/**
     * La funcion radix2DIFRNFFT calcula la FFT de radio 2 para el array a con
     * entrada en orden reversal y salida en orden natural.
     *
     * @param N la dimension de los datos
     * @param a array con los datos de entrada (tiene orden 2N ya que en los
     * huecos pares guaradamos la parte real y en los impares la parte
     * imaginaria)
     * @param w array que contiene las raices de la unidad
     * @return a que ahora contiene la FFT lo que antes contenía a y en orden
     * natural
     */
     public static double[] radix2DIFRNFFT(int N, double a[], double w[]) {

        int NumOfProblems = 1;
        int ProblemSize = 2 * N;
        int Distance = 2;
        int Jtwiddle = 0;

        while (ProblemSize > 2) {

            for (int JFirst = 0; JFirst <= 2 * NumOfProblems - 1; JFirst = JFirst +
    2) {
                int J = JFirst;
                Jtwiddle = 0;

                while (J < 2 * N - 1) {

                    double tempr = a[J];
                    double tempi = a[J + 1];
                    //mariposa
                    //primera operacion

                    //rellamos la parte real
                    a[J] = tempr + a[J + Distance];
                    //rellenamos la parte imaginaria
                    a[J + 1] = tempi + a[J + Distance + 1];
                    //segunda operacion
                    double delta = w[Jtwiddle + 1] + w[Jtwiddle];
                    double gamma = w[Jtwiddle + 1] - w[Jtwiddle];
                    double m1 = ((tempr - a[J + Distance]) + (tempi - a[J + Distance
     + 1])) * w[Jtwiddle];
                    double m2 = delta * (tempi - a[J + Distance + 1]);
                    double m3 = gamma * (tempr - a[J + Distance]);
                    //rellenamos la parte real
                    a[J + Distance] = m1 - m2;
                    //rellenamos la parte imaginaria
                    a[J + Distance + 1] = m1 + m3;

                    //para recorer el vector de las raices
                    Jtwiddle = Jtwiddle + 2;
                    J = J + 4 * NumOfProblems;
                }
            }
            NumOfProblems = NumOfProblems * 2;
            ProblemSize = ProblemSize / 2;
```

```
56              Distance = Distance * 2;
            }
58
            return a;
60      }
```

Algoritmo DIFRN FFT

9.4. Algoritmo RN FFT radio 4

```
1
  /**
3      * La funcion radix4DIFNRFFT realiza la transformada de Fourier de radio 4
       * al los datos de a; con entrada en orden natural y salida en orden bit
5      * reversal
       *
7      * @param N numero de datos complejos
       * @param kexp N = 4^{k}
9      * @param a contiene los datos iniciales
       * @param w contiene las raices de la unidad
11     * @param w2 contiene las raices de la unidad al cuadrado
       * @param w3 contiene las raices de la unidad al cubo
13     * @return a con la transformada de los datos que contenia anteriormente a
       */
15     public static double[] radix4DIFNRFFT(int N, double a[], double w[], double w2
    [], double w3[]) {
            int NumOfProblems = 1;
17          int ProblemSize = 2 * N;
            int Jtwiddle = 0;
19
            while (ProblemSize > 4) {
21
                int HalfSize = ProblemSize / 2;
23              int quarterSize = ProblemSize / 4;
                int threequarterSize = (3 * ProblemSize) / 4;
25
                for (int k = 0; k < NumOfProblems; k++) {
27                  int JFirst = k * ProblemSize;
                    int JLast = JFirst + quarterSize - 1;
29
                    /*los arrays w,w2,w3 se tienen que recorrer en orden a como
31                   estan colocados, pero cada vez que empezamos un subproblema
                     nuevo, deben de volver a recorer el trozo del array correspondiente
33                   a las raices de ProblemSize correspondiente
                     */
35                  Jtwiddle = (int) (2 * N - ProblemSize) / 3;

37                  double Tempr1;
                    double Tempr2;
39                  double Tempi1;
                    double Tempi2;
41                  double A;  // A+ i B = (x_l + x_{l+HalfSize) + i(x_{l+1} + x_{l+
    HalfSize+1))
                    double B;
43                  double C;  // C+ i D = (x_{l+quarterSize} +x_{l+threequarterSize}) +
     i  (x_{l+quarterSize+1} +x_{l+threequarterSize+1})
                    double D;
```

```
45                double E;  // E+ i F = (x_l + x_{l+HalfSize) + i(x_{l+1} + x_{l+
    HalfSize+1))
                  double F;
47                double G;  // G+ i H = (x_{l+quarterSize} +x_{l+threequarterSize}) +
     i  (x_{l+quarterSize+1} +x_{l+threequarterSize+1})
                  double H;
49
                  double w2jr;  //para raices
51                double w2ji;
                  double wjr;
53                double wji;
                  double w3jr;
55                double w3ji;

57                for (int J = JFirst; J < JLast; J = J + 2) {

59                    Tempr1 = a[J];
                      Tempr2 = a[J + quarterSize];
61                    Tempi1 = a[J + 1];
                      Tempi2 = a[J + quarterSize + 1];
63
                      //primera parte de la mariposa
65                    A = Tempr1 + a[J + HalfSize];
                      B = Tempi1 + a[J + HalfSize + 1];
67
                      C = Tempr2 + a[J + threequarterSize];
69                    D = Tempi2 + a[J + threequarterSize + 1];

71                    E = Tempr1 - a[J + HalfSize];
                      F = Tempi1 - a[J + HalfSize + 1];
73
                      G = Tempr2 - a[J + threequarterSize];
75                    H = Tempi2 - a[J + threequarterSize + 1];

77                    //segunda parte de la mariposa
                      //primer cuarto
79                    a[J] = A + C;

81                    a[J + 1] = B + D;

83                    //CASO ESPECIAL : J=0
                      if (J == 0) {
85                        //segundo cuarto
                          a[J + quarterSize] = A - C;
87                        a[J + quarterSize + 1] = B - D;
                          //tercer cuarto
89                        a[J + HalfSize] = E - H;
                          a[J + HalfSize + 1] = F - G;
91                        //cuarto cuarto
                          a[J + threequarterSize] = E + H;
93                        a[J + threequarterSize + 1] = F + G;

95                        Jtwiddle = Jtwiddle + 2;
                      } else if (J == N / 4) {
97                        //segundo cuarto
                          a[J + quarterSize] = B - D;
99                        a[J + quarterSize + 1] = C - A;
                          //tercer cuarto
101                       a[J + HalfSize] = (1 / Math.sqrt(2)) * (E - H - G + F);
                          a[J + HalfSize + 1] = (-1 / Math.sqrt(2)) * (E - H + G - F);
```

```
103                         //ultimo cuarto
                            a[J + threequarterSize] = (1 / Math.sqrt(2)) * (-E - H + F +
    G);
105                         a[J + threequarterSize + 1] = (-1 / Math.sqrt(2)) * (E + H +
    F + G);
                            //avanzamos en los arrays de las raices
107                         Jtwiddle = Jtwiddle + 2;
                        } else if (J == N / 8) {
109                         //segundo cuarto
                            w2jr = w2[Jtwiddle];
111                         w2ji = w2[Jtwiddle + 1];
                            double delta = w2ji + w2jr;
113                         double gamma = w2ji - w2jr;
                            double m1 = ((A - C) + (B - D)) * w2jr;
115                         double m2 = delta * (B - D);
                            double m3 = gamma * (A - C);
117                         a[J + quarterSize] = (1 / Math.sqrt(2)) * (A - C + B - D);
                            a[J + quarterSize + 1] = (1 / Math.sqrt(2)) * (B - D - A + C
    );;
119                         //tercer cuarto
                            wjr = w[Jtwiddle];
121                         wji = w[Jtwiddle + 1];
                            delta = wji + wjr;// Jtw=0  //w_j
123                         gamma = wji - wjr;
                            m1 = ((E + H) + (F - G)) * wjr;
125                         m2 = delta * (F - G);
                            m3 = gamma * (E + H);
127                         a[J + HalfSize] = m1 - m2;
                            a[J + HalfSize + 1] = m1 + m3;
129
                            //ultimo cuarto
131                         //Aqui se pueden usar las raices anteriores
                            w3jr = w3[Jtwiddle];
133                         w3ji = w3[Jtwiddle + 1];

135                         delta = w3ji + w3jr;  // Jtw=0  //w_3j
                            gamma = w3ji - w3jr;
137                         m1 = ((E - H) + (F + G)) * w3jr;
                            m2 = delta * (F + G);
139                         m3 = gamma * (E - H);

141                         a[J + threequarterSize] = m1 - m2;
                            a[J + threequarterSize + 1] = m1 + m3;
143
                            //avanzamos en los arrays de las raices
145                         Jtwiddle = Jtwiddle + 2;

147                     } else if (J == (6 * N) / 16) {
                            //segundo cuarto
149                         a[J + quarterSize] = (-1 / Math.sqrt(2)) * (A - C + B - D);
                            a[J + quarterSize + 1] = (-1 / Math.sqrt(2)) * (A - C - B +
    D);
151
                            //tercer cuarto
153                         wjr = w[Jtwiddle];
                            wji = w[Jtwiddle + 1];
155                         double delta = wji + wjr;// Jtw=0  //w_j
                            double gamma = wji - wjr;
157                         double m1 = ((E + H) + (F - G)) * wjr;
                            double m2 = delta * (F - G);
```

```
159                     double m3 = gamma * (E + H);

161                     a[J + HalfSize] = m1 - m2;
                        a[J + HalfSize + 1] = m1 + m3;
163
                        //ultimo cuarto
165                     //Aqui se pueden usar las raices anteriores
                        w3jr = w3[Jtwiddle];
167                     w3ji = w3[Jtwiddle + 1];

169                     delta = w3ji + w3jr;  // Jtw=0  //w_3j
                        gamma = w3ji - w3jr;
171                     m1 = ((E - H) + (F + G)) * w3jr;
                        m2 = delta * (F + G);
173                     m3 = gamma * (E - H);

175                     a[J + threequarterSize] = m1 - m2;
                        a[J + threequarterSize + 1] = m1 + m3;
177
                        //avanzamos en los arrays de las raices
179                     Jtwiddle = Jtwiddle + 2;

181                 } else {

183                     //segundo cuarto
                        w2jr = w2[Jtwiddle];
185                     w2ji = w2[Jtwiddle + 1];
                        double delta = w2ji + w2jr;
187                     double gamma = w2ji - w2jr;
                        double m1 = ((A - C) + (B - D)) * w2jr;
189                     double m2 = delta * (B - D);
                        double m3 = gamma * (A - C);
191                     a[J + quarterSize] = m1 - m2;
                        a[J + quarterSize + 1] = m1 + m3;
193
                        //tercer cuarto
195                     wjr = w[Jtwiddle];
                        wji = w[Jtwiddle + 1];
197                     delta = wji + wjr;// Jtw=0  //w_j
                        gamma = wji - wjr;
199                     m1 = ((E + H) + (F - G)) * wjr;
                        m2 = delta * (F - G);
201                     m3 = gamma * (E + H);

203                     a[J + HalfSize] = m1 - m2;
                        a[J + HalfSize + 1] = m1 + m3;
205
                        //ultimo cuarto
207                     //Aqui se pueden usar las raices anteriores
                        w3jr = w3[Jtwiddle];
209                     w3ji = w3[Jtwiddle + 1];

211                     delta = w3ji + w3jr;  // Jtw=0  //w_3j
                        gamma = w3ji - w3jr;
213                     m1 = ((E - H) + (F + G)) * w3jr;
                        m2 = delta * (F + G);
215                     m3 = gamma * (E - H);

217                     a[J + threequarterSize] = m1 - m2;
                        a[J + threequarterSize + 1] = m1 + m3;
```

```
219
                        //avanzamos en los arrays de las raices
221                     Jtwiddle = Jtwiddle + 2;
                    }
223             }

225         }

227         NumOfProblems = 4 * NumOfProblems;
            ProblemSize = ProblemSize / 4;
229
        }
231     return a;
    }
```

Algoritmo DIFRN FFT radio 4

9.5. Algoritmo FFT radio split

```
/**
2     * La funcion radixSplitDIFNRFFT calcula la transformada de Fourier
    discreta mediante el algoritmo DIF FFT de radio split
     * al los datos de f; con entrada en orden natural y salida en orden bit
4    * reverse
     *
6    * @param N numero de datos complejos
     * @param a el vector de entrada, contiene los datos iniciales
8    * @param w contiene las raices de la unidad
     * @param w3 contiene las raices de la unidad al cubo
10   * @param exponente N =2^{exponente}
     * @param b es una copia del array a
12   * @param pointed punteropara controlar los diferentes niveles
     */
14
    public static double[] radixSplitDIFNRFFT(int N, double b[], double w[], double
    w3[], int exponente, byte [] pointed) {
16      int ProblemSize = N;
        byte level = (byte) exponente;
18      pointed[0] = level;
        int iniW = 0;  //inicio lista de raices en cada nivel
20      double wjr;  //para raices
        double wji;
22      double w3jr;
        double w3ji;
24      int Jtwiddle = 0;
        for (level = (byte) exponente; level > 1; level--) {
26          int i = 0;
            int HalfSize = ProblemSize / 2;
28          int quarterSize = ProblemSize / 4;
            int threequarterSize = (3 * ProblemSize) / 4;
30          while (i < N) {

32              if (pointed[i] == level) {
                    pointed[i]--;
34                  pointed[i + HalfSize] = (byte) (level - 2);
                    pointed[i + threequarterSize] = (byte) (level - 2);
36                  for (int j = 0; j < quarterSize; j++) {
```

```
                    int ind = i + j;
38                  Jtwiddle = iniW + 2 * j;
                    // Primer y segundo cuarto
40                  double A;  // A+ i B = (x_l + x_{l+HalfSize) + i(x_{l+1} +
    x_{l+HalfSize+1))
                    double B;
42                  double C;  // C+ i D = (x_{l+quarterSize} +x_{l+
    threequarterSize}) + i  (x_{l+quarterSize+1} +x_{l+threequarterSize+1})
                    double D;
44                  double E;  // E+ i F = (x_l + x_{l+HalfSize) + i(x_{l+1} +
    x_{l+HalfSize+1))
                    double F;
46                  double G;  // G+ i H = (x_{l+quarterSize} +x_{l+
    threequarterSize}) + i  (x_{l+quarterSize+1} +x_{l+threequarterSize+1})
                    double H;
48                  A = b[2 * ind] + b[2 * (ind + HalfSize)];
                    B = b[2 * ind + 1] + b[2 * (ind + HalfSize) + 1];
50
                    E = b[2 * ind] - b[2 * (ind + HalfSize)];
52                  F = b[2 * ind + 1] - b[2 * (ind + HalfSize) + 1];

54                  C = b[2 * (ind + quarterSize)] + b[2 * (ind +
    threequarterSize)];
                    D = b[2 * (ind + quarterSize) + 1] + b[2 * (ind +
    threequarterSize) + 1];
56
                    G = b[2 * (ind + quarterSize)] - b[2 * (ind +
    threequarterSize)];
58                  H = b[2 * (ind + quarterSize) + 1] - b[2 * (ind +
    threequarterSize) + 1];

60                  b[2 * ind] = A;
                    b[2 * ind + 1] = B;
62                  b[2 * (ind + quarterSize)] = C;
                    b[2 * (ind + quarterSize) + 1] = D;
64
                    //tercer cuarto
66                  wjr = w[Jtwiddle];
                    wji = w[Jtwiddle + 1];
68                  double delta = wji + wjr;// Jtw=0  //w_j
                    double gamma = wji - wjr;
70                  double m1 = ((E + H) + (F - G)) * wjr;
                    double m2 = delta * (F - G);
72                  double m3 = gamma * (E + H);

74                  b[2 * (ind + HalfSize)] = m1 - m2;

76                  b[2 * (ind + HalfSize) + 1] = m1 + m3;

78                  //ultimo cuarto
                    //Aqui se pueden usar las raices anteriores
80                  w3jr = w3[Jtwiddle];
                    w3ji = w3[Jtwiddle + 1];
82
                    delta = w3ji + w3jr;  // Jtw=0  //w_3j
84                  gamma = w3ji - w3jr;
                    m1 = ((E - H) + (F + G)) * w3jr;
86                  m2 = delta * (F + G);
                    m3 = gamma * (E - H);
88
```

```
                    b[2 * (ind + threequarterSize)] = m1 - m2;
                    b[2 * (ind + threequarterSize) + 1] = m1 + m3;

                }
                i += ProblemSize;
            } else {
                i += ProblemSize;
            }
        }

        //baja el nivel (tamaño) de los problemas
        iniW += HalfSize;
    }
    int i = 0;
    while (i < N) {
        if (pointed[i] == 1) {
            pointed[i]--;
            double A = b[2 * i], B = b[2 * (i + 1)];
            double C = b[2 * i + 1], D = b[2 * (i + 1) + 1];
            b[2 * i] = A + B;
            b[2 * i + 1] = C + D;
            b[2 * (i + 1)] = A - B;
            b[2 * (i + 1) + 1] = C - D;
            i = i + 2;
        } else {
            i++;
        }
    }
    return b;
}
```

Algoritmo DIF FFT radio split

9.6. Algoritmo para cambiar el orden de un vector

```
/**
 * Bitreversal para cambiar el orden del  vector de datos de la fft
 *
 * @param L2 array al que queremos darle la vuelta
 * @param k 2^{k}
 * @param n numero de datos complejos
 * @return LR datos con orden bitseverse
 */
public static double[] bitreversal2(double[] L2, int k, int n) {
    // la entrada es un vector de longitud n=2^k
    double[] LR = new double[2 * n];
    int[] powrev = new int[k];
    pow[0] = 1;
    powrev[0] = n / 2;
    for (int i = 1; i < k; i++) {
        powrev[i] = powrev[i - 1] / 2;
    }
    for (int m = 0; m < n; m++) {
        int j = 0;
        int mm = m;
        for (int p = 0; p < k; p++) {
            j += mm % 2 * powrev[p];
```

```
                mm /= 2;
            }
            LR[2 * m] = L2[2 * j];
            LR[2 * m + 1] = L2[2 * j + 1];
        }
        return LR;
    }
```

Algoritmo bit reversal

10 | Anexo D : Códigos multiplicación de números grandes

```
/**
 * La funcion primeraPotenciaDos calcula la primera potencia mas cercana al
 * doble de N=2^{k}
 *
 * @param k 2^{K}
 * @return sol
 */
public static int primeraPotenciaDos(int k) {
    int sol = 0;
    int candidato = 2 * k;
    double potencia = (Math.log(candidato) / Math.log(2));
    //.floor aproxima por abajo
    double ParteEntera = Math.floor(potencia);
    if (ParteEntera == potencia) {
        sol = candidato;
    } else {
        //.ceil aproxima por arriba
        sol = (int) Math.pow(2, Math.ceil(potencia));

    }
    return sol;
}
}
```

Función para calcular la potencia de dos $2^k > 2N$ donde k es lo mas pequeño posible

```

public static double[] multiplicacionGrande2(int anum, int bnum) {
    double[] a = meterNumero(anum);
    double[] b = meterNumero(bnum);

    //Si los dos numeros introducidos tienen el mismo tamaño
    if (a.length == b.length) {
        int k = (int) (Math.log((a.length) / 2) / Math.log(2));
        int N = (int) a.length / 2;

        //aplicamos la FFT con entrada en orden natural y
        //salida en orden bitreverse a los numeros
        double[] wa = AlgoritmosNR.raicesReduccion(k);
        double ffta[] = AlgoritmosNR.radix2DIFNRFFT(N, a, wa);

        double[] wb = AlgoritmosNR.raicesReduccion(k);
```

```
18              double[] fftb = AlgoritmosNR.radix2DIFNRFFT(N, b, wb);
20              //multiplicamos las salidas de la FFT
                double[] multiplicacion = new double[a.length];
22
                for (int i = 0; i < a.length; i = i + 2) {
24                  double delta = fftb[i + 1] + fftb[i];
                    double gamma = fftb[i + 1] - fftb[i];
26                  double m1 = (ffta[i] + ffta[i + 1]) * fftb[i];
                    double m2 = delta * ffta[i + 1];
28                  double m3 = gamma * ffta[i];
30                  multiplicacion[i] = (m1 - m2);
                    multiplicacion[i + 1] = (m1 + m3);
32              }
34              //aplicamos el la inversa FFT con entrada en bitreversal y
                //salida en orden natural (para no tener que aplicar bitreversal antes
    de
36              //aplicar la inversa).
                double wm[] = Inversa.raicesReduccionRN(k);
38              double auxiliar[] = Inversa.radix2DIFRNFFT(N, multiplicacion, wm);
                //la division entre N que necesita la inversa
40              double auxdiv[] = new double[auxiliar.length];
                for (int i = 0; i < auxiliar.length; i++) {
42                  auxdiv[i] = auxiliar[i] / N;
                }
44              //como estamos en base 10 puede que necesitemos dos huecos mas en el
    array
                //creamos un array auxiliar con dos huecos mas
46              double resultado[] = new double[auxiliar.length + 2];
                for (int i = 0; i < auxiliar.length; i++) {
48                  resultado[i] = auxdiv[i];
                }
50              //como estamos en base 10 los numeros deben estar en dicha base
                for (int i = 0; i < resultado.length; i = i + 2) {
52                  if (resultado[i] >= 10) {
                        resultado[i + 2] += resultado[i] / 10;
54                      resultado[i] = resultado[i] % 10;
                    }
56              }
58              double[] solucion = new double[resultado.length];
                for (int j = 0; j < resultado.length; j = j + 2) {
60                  solucion[j] = resultado[j];
                }
62              return solucion;
64              //si el numero a es mayor que el numero b necesitamos los array del tama
    ño del a
            } else if (a.length > b.length) {
66              double bAUX[] = new double[a.length];
                for (int i = 0; i < b.length; i++) {
68                  bAUX[i] = b[i];
                }
70
                for (int i = a.length - b.length; i < a.length; i++) {
72                  bAUX[i] = 0;
                }
74              int k = (int) (Math.log((a.length) / 2) / Math.log(2));
```

```
            int N = (int) a.length / 2;

            //aplicamos la FFT con entrada en orden natural y
            //salida en orden bitreversal a los numeros
            double[] wa = AlgoritmosNR.raicesReduccion(k);
            double ffta[] = AlgoritmosNR.radix2DIFNRFFT(N, a, wa);

            double[] wb = AlgoritmosNR.raicesReduccion(k);
            double[] fftb = AlgoritmosNR.radix2DIFNRFFT(N, bAUX, wb);

            //multiplicamos las salidas de la FFT
            double[] multiplicacion = new double[a.length];

            for (int i = 0; i < a.length; i = i + 2) {
                double delta = fftb[i + 1] + fftb[i];
                double gamma = fftb[i + 1] - fftb[i];
                double m1 = (ffta[i] + ffta[i + 1]) * fftb[i];
                double m2 = delta * ffta[i + 1];
                double m3 = gamma * ffta[i];

                multiplicacion[i] = (m1 - m2);
                multiplicacion[i + 1] = (m1 + m3);
            }

            //aplicamos el la inversa FFT con entrada en bitreversal y
            //salida en orden natural (para no tener que aplicar bitreversal antes
    de
            //aplicar la inversa).
            double wm[] = Inversa.raicesReduccionRN(k);
            double auxiliar[] = Inversa.radix2DIFRNFFT(N, multiplicacion, wm);
            double auxdiv[] = new double[auxiliar.length];
            for (int i = 0; i < auxiliar.length; i++) {
                auxdiv[i] = auxiliar[i] / N;
            }
            //como estamos en base 10 puede que necesitemos dos huecos mas en el
    array
            //creamos un array auxiliar con dos huecos mas
            double resultado[] = new double[auxiliar.length + 2];
            for (int i = 0; i < auxiliar.length; i++) {
                resultado[i] = auxdiv[i];
            }
            //como estamos en base 10 los numeros deben estar en dicha base
            for (int i = 0; i < resultado.length; i = i + 2) {
                if (resultado[i] >= 10) {
                    resultado[i + 2] += resultado[i] / 10;
                    resultado[i] = resultado[i] % 10;
                }
            }

            double[] solucion = new double[resultado.length];
            for (int j = 0; j < resultado.length; j = j + 2) {
                solucion[j] = resultado[j];
            }
            return solucion;
        } else {
            //si el numero b es mas grande que el numero a
            double aAUX[] = new double[b.length];
            for (int i = 0; i < a.length; i++) {
                aAUX[i] = a[i];
            }
```

```
134             for (int i = b.length - a.length; i < b.length; i++) {
                    aAUX[i] = 0;
136             }
                int k = (int) (Math.log((b.length) / 2) / Math.log(2));
138             int N = (int) b.length / 2;

140             //aplicamos la FFT con entrada en orden natural y
                //salida en orden bitreverse a los numeros
142             double[] wa = AlgoritmosNR.raicesReduccion(k);
                double ffta[] = AlgoritmosNR.radix2DIFNRFFT(N, aAUX, wa);
144
                double[] wb = AlgoritmosNR.raicesReduccion(k);
146             double[] fftb = AlgoritmosNR.radix2DIFNRFFT(N, b, wb);

148             //multiplicamos las salidas de la FFT
                double[] multiplicacion = new double[b.length];
150
                for (int i = 0; i < b.length; i = i + 2) {
152                 double delta = fftb[i + 1] + fftb[i];
                    double gamma = fftb[i + 1] - fftb[i];
154                 double m1 = (ffta[i] + ffta[i + 1]) * fftb[i];
                    double m2 = delta * ffta[i + 1];
156                 double m3 = gamma * ffta[i];

158                 multiplicacion[i] = (m1 - m2);
                    multiplicacion[i + 1] = (m1 + m3);
160             }

162             //aplicamos el la inversa FFT con entrada en bitreversal y
                //salida en orden natural (para no tener que aplicar bitreversal antes
    de
164             //aplicar la inversa).
                double wm[] = Inversa.raicesReduccionRN(k);
166             double auxiliar[] = Inversa.radix2DIFRNFFT(N, multiplicacion, wm);
                double auxdiv[] = new double[auxiliar.length];
168             for (int i = 0; i < auxiliar.length; i++) {
                    auxdiv[i] = auxiliar[i] / N;
170             }
                //como estamos en base 10 puede que necesitemos dos huecos mas en el
    array
172             //creamos un array auxiliar con dos huecos mas
                double resultado[] = new double[auxiliar.length + 2];
174             for (int i = 0; i < auxiliar.length; i++) {
                    resultado[i] = auxdiv[i];
176             }
                //como estamos en base 10 los numeros deben estar en dicha base
178             for (int i = 0; i < resultado.length; i = i + 2) {
                    if (resultado[i] >= 10) {
180                     resultado[i + 2] += resultado[i] / 10;
                        resultado[i] = resultado[i] % 10;
182                 }
                }
184
                double[] solucion = new double[resultado.length];
186             for (int j = 0; j < resultado.length; j = j + 2) {
                    solucion[j] = resultado[j];
188             }
                return solucion;
190         }
```

Función para multiplicar los números con los algorimos FFT

```
1
  /**
3     * La funcion multiplicar multiplica los coeficientes de la solucion por las
      * potencias de diez correspondientes para obtener el resultado de la
5     * multiplicacion
      *
7     * @param coeficientes el array que contiene los coeficientes de la FFT
      * @return sol
9     */
    public static double multiplicar(double[] coeficientes) {
11      double sol = 0;
        int t = 0;
13      for (int i = 0; i < coeficientes.length; i = i + 2) {
            //Math.round para redondear los numeros
15          sol = sol + (Math.round(coeficientes[i]) * Math.pow(10, t));
            t = t + 1;
17      }
        return sol;
19  }
    }
```

Función para multiplicar los números de un array por potencias de diez

11 | Anexo E : Traformada de Fourier en $L^2(\mathbb{R})$

Los contenidos de este capítulo hacen referencia al libro [13].

No es trivial definir la transformada de Fourier en $L^2(\mathbb{R})$. Una forma "curiosa" de hacerlo (debida a Norbert Wiener) es la que vamos a describir a continuación:

Definición 11.1:

Se definen los **Polinomios de Hermite** en $L^1(\mathbb{R})$ como:

$$H_n(x) = (-1)^n e^{x^2} \frac{d^n}{dx^n}\left(e^{-x^2}\right).$$

Definición 11.2:

Se definen las **funciones de Hermite** como:

$$h_n(x) = H_n(x)e^{\frac{-x^2}{2}}.$$

Es evidente que $h_n \in L^1(\mathbb{R}) \cap L^2(\mathbb{R})$ para todo $n \in \mathbb{R}$, pues son funciones de decrecimiento exponencial.

A partir de estas definiciones se verifican los siguientes resultado:

Lema 11.1:

span$\{h_n(x) : n = 0, 1, \ldots\}$ es denso en $L^2(\mathbb{R})$.

Lema 11.2:

$\langle h_n, h_m\rangle = \int_{-\infty}^{\infty} h_n(x)h_m(x)dx = 2^n n\sqrt{\pi}\delta_{n,m}$, donde $\delta_{n,m} = \begin{cases} 1 & si \quad n = m \\ 0 & si \quad n \neq m. \end{cases}$

Lema 11.3:

Sea

$$W(x)(\xi) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} x(t) e^{-it\xi} dt$$

$$W^{-1}(x)(\xi) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} x(\xi) e^{it\xi} d\xi$$

donde $x \in L^1(\mathbb{R})$. Entonces $W(h_n) = (-i)^n h_n$ y $W^{-1}(h_n) = i^n h_n$.

Ahora bien, por el Lema 11.1 y por el Lema 11.2 tenemos que todo $x \in L^2(\mathbb{R})$ se puede escribir como

$$x = \sum_{n=0}^{\infty} \frac{\langle x, h_n \rangle}{\langle h_n, h_n \rangle} \cdot h_n$$

con convergencia de la serie en $L^2(\mathbb{R})$, lo cual motiva la siguiente defición de la transformada de Fourier en $L^2(\mathbb{R})$:

Definición 11.3:

Dada $x \in L^2(\mathbb{R})$ definimos la transformada de Fourier, y su inversa, como:

$$W(x) = \sum_{n=0}^{\infty} \frac{\langle x, h_n \rangle}{\langle h_n, h_n \rangle} (-i)^n h_n$$

$$W^{-1}(x) = \sum_{n=0}^{\infty} \frac{\langle x, h_n \rangle}{\langle h_n, h_n \rangle} (i)^n h_n$$

Teorema 11.1:

Si $x \in L^2(\mathbb{R})$, entonces se cumple la igualdad $W^{-1}(W(x)) = x = W(W^{-1}(x))$.

Demostración. Para la demostración del teorema, vamos a probar la igualdad $W(W(x(t))) = x(-t)$.

Sea $x \in L^2(\mathbb{R})$. Por un lado, teniendo en cuenta el Lema 12.3:

$$\begin{aligned} W(W(x(t))) &= W\left(\textstyle\sum_{n=0}^{\infty} \frac{\langle x, h_n \rangle}{\langle h_n, h_n \rangle} (-i)^n h_n\right) \\ &= \textstyle\sum_{n=0}^{\infty} \frac{\langle x, h_n \rangle}{\langle h_n, h_n \rangle} (-i)^n (-i)^n h_n \\ &= \textstyle\sum_{n=0}^{\infty} \frac{\langle x, h_n \rangle}{\langle h_n, h_n \rangle} (-1)^n h_n \end{aligned}$$

Por otro lado:

$$x(-t) = \sum_{n=0}^{\infty} \frac{\langle x(-t), h_n \rangle}{\langle h_n, h_n \rangle} h_n(-t)$$

.

Entonces como

$$\begin{aligned} h_n(-t) &= (-1)^n \cdot e^{t^2} \cdot \tfrac{d^n}{dt^n}(e^{-t^2})e^{\frac{-t^2}{2}} \\ &= (-1)^n \cdot e^{\frac{t^2}{2}} \cdot \tfrac{d^n}{dt^n}(e^{-t^2}) \\ &= (-1)^n h_n(t). \end{aligned}$$

Sustituyendo esto último en la igualdad anterior:

$$\begin{aligned} x(-t) &= \textstyle\sum_{n=0}^{\infty} \frac{\langle x(-t), h_n\rangle}{\langle h_n\rangle} h_n(-t) \\ &= \textstyle\sum_{n=0}^{\infty} \frac{\langle x(-t), h_n\rangle}{\langle h_n, h_n\rangle} (-1)^n h_n(t), \end{aligned}$$

$$\begin{aligned} \langle x(-t), h_n(t)\rangle &= \textstyle\int_{-\infty}^{\infty} x(-t)(-1)^n e^{t^2} \frac{d^n}{dt^n}(e^{-t^2})e^{\frac{-t^2}{2}} dt \\ &= \textstyle\int_{-\infty}^{\infty} x(-t)(-1)^n e^{\frac{t^2}{2}} \frac{d^n}{dt^n}(e^{-t^2})e^{\frac{-t^2}{2}} dt. \end{aligned}$$

Por tanto la demostración finalizará cuando probemos que $\langle x(-t), h_n\rangle = \langle x(t), h_n\rangle$ Haciendo el cambio de variable $u = -t \to du = -dt$, nos queda:

$$\begin{aligned} \langle x(-t), h_n(t)\rangle &= \textstyle\int_{-\infty}^{\infty} x(u)(-1)^n e^{\frac{u^2}{2}} \frac{d^n}{du^n}(e^{-u^2}) du \\ &= \langle x(u), h_n(u)\rangle. \end{aligned}$$

Por lo que tenemos que $x(-t) = \sum_{n=0}^{\infty} \frac{\langle x(t), h_n\rangle}{\langle h_n, h_n\rangle}(-1)^n h_n(t) = W(W(x(t)))$,

Lo que finaliza la prueba, ya que como hemos probado que $x(-t) = W(W(x))$ y por tanto $x(-t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} W(x)(\xi)e^{-it\xi} d\xi$, entonces

$$\begin{aligned} x(t) &= \tfrac{1}{\sqrt{2\pi}} \textstyle\int_{-\infty}^{\infty} W(x)(\xi)e^{-i\xi(-t)} d\xi \\ &= \tfrac{1}{\sqrt{2\pi}} \textstyle\int_{-\infty}^{\infty} W(x)(\xi)e^{i\xi t} d\xi \\ &= W^{-1}(W(x))(t). \end{aligned}$$

□

NOTA: Si partimos de la definición de transformada de Fourier que se ha usado para la demostración del teorema del muestreo, $\mathcal{F}(x)(\xi) = \int_{-\infty}^{\infty} x(t)e^{-2\pi it\xi} dt$, entonces podemos usar el teorema anterior para demostrar que $\mathcal{F}^{-1}(x)(t) = \int_{-\infty}^{\infty} x(\xi)e^{2\pi it\xi} d\xi$.

En efecto: Como $W^{-1}(W(x(t))) = x(t)$ entonces,

$$x(t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \left[\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} x(t)e^{-it\xi} dt \right] e^{i\xi t} d\xi = \frac{1}{2\pi} \int_{-\infty}^{\infty} \left[\int_{-\infty}^{\infty} x(t)e^{-it\xi} dt \right] e^{i\xi t} d\xi$$

Haciendo el cambio de variable $\xi = 2\pi u$, obtenemos que

$$x(t) = \int_{-\infty}^{\infty} \left[\int_{-\infty}^{\infty} x(t)e^{-2\pi itu} dt \right] e^{2i\pi tu} du$$

Por lo que $\mathcal{F}^{-1} = \int_{-\infty}^{\infty} x(t)e^{2\pi it\xi} d\xi$.

Por tanto, hemos demostrado el teorema integral de Fourier para funciones en $L^2(\mathbb{R})$, que es el siguiente:

Teorema 11.2:

Los operadores $\mathcal{F} : L^2(\mathbb{R}) \to L^2(\mathbb{R})$ y $\mathcal{F}^{-1} : L^2(\mathbb{R}) \to L^2(\mathbb{R})$ definidos por

$$\mathcal{F}(x)(\xi) = \int_{-\infty}^{\infty} x(t) e^{-2\pi i \xi t} dt$$

$$\mathcal{F}^{-1}(x)(\xi) = \int_{-\infty}^{\infty} x(t) e^{2\pi i \xi t} d\xi$$

Son inversos el uno del otro: $\mathcal{F}\mathcal{F}^{-1} = Id = \mathcal{F}^{-1}\mathcal{F}$.

12 | Anexo F : Código Matlab

Los contenidos de este Anexo son de los libros [2] y [3].

```
 /**
* La funcion recuperarFrecuencias calcula la transformada de Fourier discreta, y
    muestra tres graficas, en la primera se muestra la señal x(t) en trazo continuo
    y las muestras x(kh) de la señal que se toman, en la segunda grafica la parte
    real de la transformada de Fourier de la señal en el intervalo [-1/(2h) , 1/(2h)
    ] y los valores que se han calculado para la parte real de dicha transformada a
    partir de las muestras tomadas y en la tercera grafica lo mismo que en la dos
    pero para la parte imaginaria de la transformada.
     * @param funcion es la funcion analogica
     * @param transformada transformada de Fourier de la funcion analogica
     * @param h perido de muestreo
     * @param L indica que las muestras se toman en el intervalo simetrico
      [-L,L]
     * @return tres graficas
     */

function y=recuperarFrecuencias(funcion,transformada,h,L)

close all;


funcionAnalogica = str2func(funcion);
transformadaFourier = str2func(transformada);

t=(-L):(0.01*h):L; //tiempo
muestras= (-L):h:(L-h); //puntos para tomar las muestras
n=1:1:length(muestras); //longitud del vector [-L,L]

x=funcionAnalogica(t);
v=funcionAnalogica(muestras); //muestras en un intervalo simetrico con el centro del
    origen de coordenadas y de longitud 2L


//Dibujamos la funcion x[t] y las muestras tomadas v

figure;

 subplot(1,3,1);
 hold on;
 stem(muestras,v);
 plot(t,x);
 xlabel('x[t] y x[k*h]')
  hold off

```

```
/*reordenamos el vector de muestras de forma que : se obtiene de v desplazando a la derecha la primera mitad del vector y dejando al principio
la segunda mitad*/

 ind=length(v)
 w=[v((ind/2+1):1:(ind)), v(1:1:(ind/2))];

 subplot(1,3,2);

 /*Aplicamos la fft al vector reoordenado y reescalamos la funcion multplicando por el periodo de muestreo h*/

 frecuencias=h*fft(w);



//Reordenamos las frecuencias
 indd=length(frecuencias);
 frecuencias=[frecuencias((indd/2+1):1:(indd)), frecuencias(1:1:(indd/2))];


/*Dibujamos la parte real de la transformada de Fourier, que le hemos introducido al programa como transformada. la cual dibujamos en trazo continuo y se superponen los resultados colocando la frecuencia hallada en el lugar correspondiente (hy_{k} es equivalente a X(f_{k}) y hy_{N-k} es equivalente a X(f_{-k}))
*/
 hold on
 stem((-1/(2*h)):(1/(length(w)*h)):(1/(2*h)-1/(length(w)*h)),real(frecuencias));
 plot((-1/(2*h)):(0.01*h):(1/(2*h)-1/(length(w)*h)),transformadaFourier((-1/(2*h)):(0.01*h):(1/(2*h)-1/(length(w)*h))));
 xlabel('Señales Re(F(x)(\xi)) y Re(fft(x[muestras]))');


/*Dibuamos la parte imaginaria teniendo en cuenta lo mismo que para la parte real*/

subplot(1,3,3);

hold on
 stem((-1/(2*h)):(1/(length(w)*h)):(1/(2*h)-1/(length(w)*h)),imag(frecuencias));
 plot((-1/(2*h)):(0.01*h):(1/(2*h)-1/(length(w)*h)),imag(transformadaFourier((-1/(2*h)):(0.01*h):(1/(2*h)-1/(length(w)*h)))));
xlabel('Señales Im(F(x)(\xi)) y Im(fft(x[muestras]))');
```

Código aplicación 2

Bibliografía

[1] Almira, J.M *Matemáticas para la recuperación de señales : una introducción*. G.E.U. Granada. (2005).

[2] Almira, J.M y Aguilar-Domingo.M *Neuromatemáticas, el leguaje eléctrico del cerebro*. CSIC y Libros de la Catarata. (2016).

[3] Amidror, I. *Mastering the Discrete Fourier Tranform in one, two or several dimensions: Pitfalls and Artifacts*. Springer.(2013).

[4] Chu, E, George, A. *INSIDE the FFT BLACK BOX, Serial and Parallel Fast Fourier Transform Algorithms*. CRC press LLC. (2000).

[5] Cabuzo, A.R. *La transformada de Fourier discreta y el formato JPEG*.(2018).

[6] Ph.J.Davis. *Interpolation and aproximation*. Dover. New York. (2014).

[7] Gasquet, C , Witomski, P. *Fourier Analysis and Applications*. Springer. New York. (1999).

[8] Hassanieh, H. *The Sparse Fourier Tansform : Theory and Practice*. ACM books. NY, USA (2018).

[9] Hassanieh, H. Indyk, P., Price, E. *Nearly Optimal Sparse Fourier Transform. In proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing*, STOC'12, pages 563-578. NY, USA (2012) : ACM.

[10] Lara, T.*Matrices Circulantes*, Divulcaciones Matemáticas, Vol. 9, Num. 1, (2001), pág 85-102.

[11] Pallares, A.J. *Apuntes de la asignatura Cálculo numérico de una variable* (2016). Disponible online : http://webs.um.es/apall/archivos/cn1v/cn1v1112.pdf.

[12] Paredes, M., Rodríguez, D., Villamizar-Morales, J. *La estructura algebraica del espacio de señales unidimensionales*, Escula de matemáticas, Vol. 23, Num. 2, pág 15-39, (2001).

[13] Wiener,N. *The Fourier integral and certain of its applications*. Cuo Archive. New York. (1993).

[14] W.Kammler, D. *A first course in Fourier Analysis*. Cambridge University Press. New York (2007).

[15] http://fdetonline.com/cambio-coordenadas-explicar-mundo-2-2/