***TP: Teoría de la Computación***

***Comparación de Algoritmos de Transformaciones de Fourier***

**Nombre y Apellido: ANTONIO JOSE ROMERO PABON**

**Legajo: 110397**

**Teléfono: +1 561-720-3429**

**e-mail: antonio.jose.ro.pa2@gmail.com**

**Carrera: LICENCIATURA EN INFORMÁTICA**

**Docente: MARIAL ALEGRE**

**Fecha de entrega: 08-11-2023**

Transformadas de Fourier y comparación de algoritmos

# ***Problema a analizar***

## Para la realización de este trabajo, utilizaremos una señal de escalón unitario definida de la siguiente manera:

## A graph of mathematical equations and formulas Description automatically generated

## Esta señal será utilizada para evaluar las distintas implementaciones de las Transformadas de Fourier que presentaremos a continuación.

# ***Implementaciones a evaluar***

## 1.- Transformada de Fourier Tradicional

La transformada de Fourier es un algoritmo para la separación de distintas frecuencias dentro de una señal. Su fórmula matemática vendría

Donde N es el tamaño del array recibido para el tren de impulso y k es un numero entero entre 0 y N-1 que representa cada uno de los índices del array.

Este algoritmo tiene una complejidad de O(n2) donde n es el tamaño del array. Esto debido a que el cálculo de esta sumatoria se realiza para cada uno de los valores del array (O(n)) y dentro de la sumatoria se realiza el cálculo n veces (O(n)), por tanto, la multiplicación de estas complejidades nos da O(n2).

En el código, este algoritmo está escrito en la siguiente función:

A computer code on a black background

Description automatically generated

(Archivo testManual.py)

Esta función representa el mismo calculo matemático, y en el caso del código, la complejidad queda expresada de manera más visual en forma de un for loop dentro de otro for loop, ambos con un rango dependiente a n.

## 2.- Transformada de Fourier separando en pares e impares

La transformada de Fourier separada en pares e impares es un algoritmo que utiliza el principio de divide y vencerás para optimizar la transformada de Fourier tradicional al separar el cálculo de cada índice en 2 grupos, los números pares y los números impares, y luego agrupando las operaciones en común para reducir la cantidad de cálculos repetidos. La fórmula matemática para este algoritmo es la siguiente:

Donde la sumatoria de la izquierda representa los números pares del array (2n) y la sumatoria de la izquierda representa los números impares del array (2n+1).

Este algoritmo también tiene una complejidad de O(n2) donde n es el tamaño del array. El cálculo de las sumatorias se realizan para cada uno de los valores del array (O(n)) y dentro de las sumatorias se realiza el cálculo n/2 veces (O(n)), por tanto, la multiplicación de estas complejidades nos da O(n2).

En el código, este algoritmo está escrito en la siguiente función:

A screen shot of a computer code

Description automatically generated

(Archivo testParImpar.py)

Esta función representa el mismo cálculo matemático.

En el caso de esta función, si bien la complejidad es la misma, podemos notar que en la transformada tradicional tiene una complejidad cuadrática debido a la multiplicación n\*n de los for loops, mientras que en este caso la complejidad cuadrática viene dada por la multiplicación n\*n/2, por lo que si bien la complejidad es la misma, el algoritmo de separar pares e impares tiene un escalado menor gracias a que la sumatoria solo itera n/2 veces (además de que se optimiza la cantidad de cálculos realizados al usar las variables Euler\_externo y Euler\_interno).

## 3.- Transformada de Fourier separando en pares e impares y usando el principio de periodicidad

La transformada de Fourier separada en pares e impares compleja es un algoritmo que nuevamente utiliza el principio de divide y vencerás para reducir la cantidad de cálculos requeridos al aprovecharse del principio de periodicidad de Euler para calcular 2 índices de la transformada al mismo tiempo. Esta transformada se expresa en las siguientes operaciones matemáticas:

Donde Xk y Xk+N/2 se calculan al mismo tiempo aprovechando que las sumatorias son las mismas, con la diferencia de que la constante de Euler afuera de la sumatoria impar es negativa (debido al principio de periodicidad, ya que se la multiplicación e-π es igual a -1, por lo que cambia el signo).

Este algoritmo también tiene una complejidad de O(n2) donde n es el tamaño del array. El cálculo de las sumatorias se realizan para la mitad de los valores del array (O(n)) y dentro de las sumatorias se realiza el cálculo n/2 veces (O(n)), por tanto, la multiplicación de estas complejidades nos da O(n2).

En el código, este algoritmo está escrito en la siguiente función:

A screen shot of a computer code

Description automatically generated

(Archivo testParImparComplejo.py)

Esta función representa el mismo cálculo matemático.

En el caso de esta función, si bien la complejidad sigue siendo la misma, la complejidad de este algoritmo escala en menor medida que los 2 algoritmos anteriores dado que su complejidad es de n/2 \* n/2 o n2/4, expresado en los rangos de los for iterando n/2 veces cada uno.

## 4.- Transformada rápida de Fourier usando el Algoritmo de Cooley Tukey

Por último, está el algoritmo de Cooley Tukey para la transformada rápida de Fourier. Este algoritmo usa los mismos principios matemáticos que la transformada anterior, con la diferencia de que introducimos conceptos de computación como la recursividad y los árboles binarios para optimizar el cálculo.

Este algoritmo a diferencia de los anteriores tiene una complejidad de O(nlog(n)) gracias a que, en vez de iterar cada valor del array de manera individual, este algoritmo se llama a si mismo de manera recursiva, separando en una hoja izquierda los valores pares y en una hoja derecha los impares, con un caso base de cuando el array recibido tiene una longitud de 1 (es importante mencionar que para que el algoritmo funcione, es necesario que el array sea del tamaño de una potencia de 2, o en su defecto, agregar valores al array para que llegue a tener un tamaño que sea una potencia de 2).

El código de este algoritmo es el siguiente:

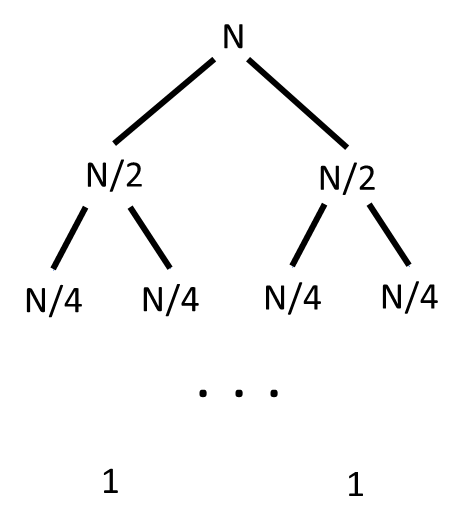
A screen shot of a computer code

Description automatically generated

(Archivo testCooleyFFT.py)

Esta función como dijimos previamente es un algoritmo recursivo donde el caso base ocurre si el array es de tamaño 1, en cuyo caso retorna el valor dentro del array, y en caso de no ser el caso base, y en los demás casos el algoritmo funciona como un árbol binario donde el nodo hoja izquierda son los valores pares y el nodo hoja derecha los impares, y luego con los valores devueltos por los 2 nodos hoja se utilizan para armar un array resultado.

Al hacer esto, dividimos el array en pares e impares de la siguiente manera:



Como este árbol binario escala en función de potencias de n, nos da una complejidad O(log n) con un logaritmo base a 2.

Luego realizamos la suma de ambos valores que al ser una única operación es complejidad O(1).

Por último, cada nivel de recursión tiene una complejidad de O(N) debido a la multiplicación y suma de las salidas pares e impares que se realiza N/2 veces.

Al multiplicar todas estas complejidades, nos da una complejidad de O(n log n).

# ***Resultados Obtenidos***

Para la evaluación de los algoritmos, estamos usando los siguientes valores para generar el tren de impulso:

A black screen with white text

Description automatically generated

(Archivo comparacionDFT.py)

Donde TIEMPO\_ENTRE\_IMPULSOS siempre será igual a 1, MAGNITUD\_TREN es la magnitud del tren de impulso y CANTIDAD\_DE\_PUNTOS es la cantidad de valores que se le esta ingresando al tren de impulsos.

Para realizar la prueba, utilizamos varios valores de cantidad de puntos y comparamos cuanto tiempo tardo cada ejecución. Los resultados obtenidos son los siguientes:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Cantidad de Puntos | | | | |
|  | 2\*\*8' | 2\*\*9' | 2\*\*10' | 2\*\*11' | 2\*\*12' |
| Tradicional | 0.098 seg | 0.330 seg | 1.445 seg | 5.151 seg | 20.759 seg |
| Pares e impares | 0.055 seg | 0.204 seg | 0.852 seg | 3.483 seg | 12.925 seg |
| Pares e impares complejo | 0.036 seg | 0.120 seg | 0.493 seg | 1.968 seg | 7.650 seg |
| Cooley Tukey FFT | 0.001 seg | 0.003 seg | 0.007 seg | 0.015 seg | 0.031 seg |

A screen shot of a computer code

Description automatically generated

(Extracto de la terminal tras correr el archivo comparacionDFT.py usando una cantidad de puntos = 2\*\*12)

Los resultados se observan más claramente en la cantidad de puntos más alta, donde podemos ver que los resultados son los esperados según los principios matemáticos, con la transformada de Fourier tradicional siendo el algoritmo más lento con 20.759 segundos, la transformada separando en pares e impares de tercera con 12.925 segundos, la transformada separando en pares e impares complejos de segunda con 7.650 segundos y la transformada rápida usando el algoritmo de Cooley Tukey de primera con 0.031 segundos.

# ***Conclusiones***

Estos casos nos sirven para demostrar la utilidad del método divide y vencerás, permitiendo que a pesar de que 3 de los algoritmos evaluados tienen una misma complejidad, el nivel de implementación del método de divide y vencerás puede generar resultados sustanciales (en el caso más extremo, la transformada tradicional tarda casi 3 veces más que la transformada separando en pares e impares compleja).

También queda demostrado el impacto que puede tener la complejidad de un algoritmo, ya que a pesar de que se ve una optimización considerable dentro de un mismo orden de complejidad, el algoritmo de Cooley y Tukey es ordenes de magnitud más rápido que los demás algoritmos por el simple hecho de a que ritmo de N escalan las iteraciones del algoritmo, y si bien en números más pequeños de N la diferencia entre los algoritmos es menor, en números mayores (los arrays en Python pueden llegar a tener un tamaño de 2\*\*23), la diferencia en eficiencia de los algoritmos se vuelve cada vez más significativa.