Transformación Rápida de Fourier

Paz Ballón Sebastian André, Ugarte Concha Sebastian, Valdivia Begazo Sharon Daniela
Email: sebastian.paz.ballon@ucsp.edu.pe
Email: sebastian.ugarte@ucsp.edu.pe
Email: sharon.valdivia@ucsp.edu.pe
Escuela profesional de Ciencia de la Computación
Universidad Católica San Pablo
Arequipa, Peru

20 de junio de 2022

ÍNDICE

I. INTRODUCCIÓN

En el año 1805 Gauss creó el primer algoritmo de "divide y vencerás", este fue utilizado para calcular la Transformada de Fourier Discreta (DFT por sus siglas en inglés- Discrete Fourier Transform) de manera recursiva. Un siglo y medio después, exactamente en abril de 1965, J.W. Cooley y John Tukey, publicaron un artículo titulado 'Un algoritmo para el cálculo automático de series complejas de Fourier', donde se plantea un algoritmo más eficiente para realizar los cálculos de la DFT, al cual llamaron Transformada Rápida de Fourier (FFT por sus siglas en inglés - Fast Fourier Transform). La motivación de este algoritmo surge por el Dr. Richard L. Garwin en IBM Watson Research, quién quería verificar si se estaba cumpliendo con un tratado entre la Unión Soviética y Estados Unidos, creado para limitar el uso de armas nucleares en ciertos territorios, Cooley y Tukey, plantearon la idea de la FFT, e instalaron sensores que podían identificar explosiones nucleares en un rango de 15 kilómetros.

La Transformada de Fourier (FT) es una transformación matemática utilizada para mutar señales de un dominio de tiempo o espacio a uno de frecuencia, y viceversa. Esta FT utiliza conceptos como: la operación de transformación de una función, la función resultante de esta y el espectro de frecuencias de una función.

El algoritmo Cooley-Tukey FFT permite la reducción del costo computacional durante el cálculo de una DFT de N puntos, reduciendo el orden de operaciones de $O(n^2)$ a $O(n \cdot log_2 N)$ multiplicaciones y sumas complejas.

Más allá de lo matemático, la FFT es utilizada en las áreas de ingeniería, medicina, telecomunicaciones, etc. Es empleada para comprimir audio, imágenes o videos, o limpiar el ruido de archivos. En estos últimos años, ha surgido una nueva aplicación de la transformada, la cual tiene que ver con el reconocimiento de voz, que es una tecnología que abre las puertas a la identificación biométrica por sonido.

II. EXPLICACIÓN DEL ALGORITMO

Como se mencionó anteriormente, la transformada Rápida de Fourier permite calcular la transformada discreta de Fourier con una complejidad menor, de $O(n^2)$ a $O(n \cdot log_2 N)$. Este algoritmo está basado en el paradigma "divide y vencerász solo puede ser aplicado con un número de muestras igual a una potencia de 2.

Partimos de la Transformada Discreta de Fourier (DFT) :

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn}$$

$$W_N = e^{-j\frac{2\pi}{N}}$$

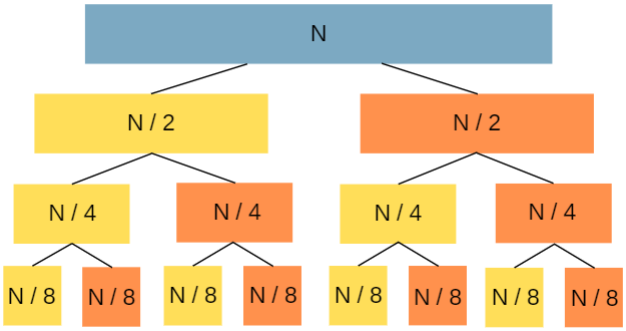
Donde se describe el cálculo de N ecuaciones. Por ejemplo si tomamos $N = 4$ la ecuación puede ser descrita como:

$$\begin{aligned}
X(0) &= x_0(0)W^0 + x_0(1)W^0 + x_0(2)W^0 + x_0(3)W^0 \\
X(1) &= x_0(0)W^0 + x_0(1)W^1 + x_0(2)W^2 + x_0(3)W^3 \\
X(2) &= x_0(0)W^0 + x_0(1)W^2 + x_0(2)W^4 + x_0(3)W^6 \\
X(1) &= x_0(0)W^0 + x_0(1)W^3 + x_0(2)W^6 + x_0(3)W^9
\end{aligned}$$

Y estas ecuaciones pueden ser representadas de forma matricial como:

$$\begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \end{bmatrix} = \begin{bmatrix} W^0 & W^0 & W^0 & W^0 \\ W^0 & W^1 & W^2 & W^3 \\ W^0 & W^2 & W^4 & W^6 \\ W^0 & W^3 & W^6 & W^9 \end{bmatrix} \begin{bmatrix} x_0(0) \\ x_0(1) \\ x_0(2) \\ x_0(3) \end{bmatrix}$$

Se observa que debido aque la matriz de W y posiblemente $x_0(k)$ son complejos, son necesarias N^2 multiplicaciones complejas y $(N)(N-1)$ adiciones complejas para desarrollar el calculo matricial requerido. La FFT debe su exito al hecho de que el algoritmo reduce el número de multiplicaciones y adiciones requeridas.
El FFT está basado en "divide y vencerás"porque descompone la DFT original $(x(k))$ en un número de DFTs más pequeñas $(g(k)$ y $h(k))$ de un tamaño equivalente a la mitad de la muestra. Este procedimiento se realiza de manera recursiva hasta que se obtengan DFTs de dos puntos, para que combinadas y operadas de manera adecuada, den origen a una DFT más grande $(x(k))$.



A continuación se explicará el algoritmo basado en divide y vencerás, a partir de la ecuación utilizada en DFT:

Supogamos que tenemos una señal $x(n)$ de N puntos o muestras, donde N es una potencia entera de 2.

$x(n)$ está compuesta por las muestras $x(0), x(1), x(2).....x(N-1)$.

- Por ello primero debemos identificar las muestras de índice par $(x(0), x(2), x(4), x(2r))$, y las muestras de índice impar $(x(1), x(3), x(4), x(2r+1))$. Por lo que a partir de la ecuación DFT de $x(n)$:

$$X(k) = \sum_{n=0}^{N-1} x(n)W_N^{kn}$$
$$k = 0, 1, 2, 3...N-1$$

- Podemos desglosarla entre una sumatoria que solo considere a los índices pares, y una que solo considere a los índices impares

$$X(k) = \sum_{r=0}^{\frac{N}{2}-1} x(2r)W_N^{2rk} + \sum_{r=0}^{\frac{N}{2}-1} x(2r+1)W_N^{(2r+1)k}$$

Se factorizan los exponentes de W_N para poder simplificar la dificultad a la hora de hacer el cálculo:

$$X(k) = \sum_{r=0}^{\frac{N}{2}-1} x(2r)(W_N^2)^{rk} + W_N^k \sum_{r=0}^{\frac{N}{2}-1} x(2r+1)(W_N^2)^{rk}$$

- Sabiendo además que:

$$W_N^n = (e^{-j\frac{2\pi}{N}})^n = e^{-j\frac{2\pi n}{N}}$$

Se puede escribir colocando a la n como inversa

$$W_N^n = e^{-j\frac{2\pi}{\frac{N}{n}}}$$

A partir de este paso, se busca las equivalencias de los nuevos valores. Por ejemplo para un $n = 2$:

$$W_N^2 = e^{-j\frac{2\pi}{\frac{N}{2}}} = W_{\frac{N}{2}}$$

- Reemplazando en la ecuación, tenemos lo siguiente:

$$X(k) = \sum_{r=0}^{\frac{N}{2}-1} x(2r)(W_{\frac{N}{2}})^{rk} + W_N^k \sum_{r=0}^{\frac{N}{2}-1} x(2r+1)(W_{\frac{N}{2}})^{rk}$$

Por lo que ahora tenemos dos sumatorias con la forma de un DFT. Se realizará la demostración de que $X(k)$ es un DFT de N muestras que se compone por dos DFTs de $\frac{N}{2}$ muestras cada una.

Aplicándolo a un ejemplo:

- Tenemos 8 muestras: $x(k) = [0, 1, 2, 3, 4, 5, 6, 7]$
  De las cuales separamos las muestras pares en un conjunto $g(k) = [0, 2, 4, 6]$
  De igual manera, separamos las muestras impares en un conjunto $h(k) = [1, 3, 5, 7]$

- Entonces igualamos la parte par de la DFT con el conjunto $g(k)$, que evalúa las muestras pares.

$$\sum_{r=0}^{\frac{N}{2}-1} x(2r)(W_{\frac{N}{2}})^{rk} = \sum_{n=0}^{N-1} g(n)(W_N)^{nk}$$

Se concluye que la transformada discreta de Fourier es: $G(k) = \sum_{n=0}^{N-1} g(n)(W_N)^{nk}$

- De igual forma, igualamos la parte impar de la transformada discreta de Fourier con el conjunto $h(k)$, que evalúa las muestras impares.

$$\sum_{r=0}^{\frac{N}{2}-1} x(2r+1)(W_{\frac{N}{2}})^{rk} = \sum_{n=0}^{N-1} h(n)(W_N)^{nk}$$

Se concluye que la transformada discreta de Fourier es: $H(k) = \sum_{n=0}^{N-1} h(n)(W_N)^{nk}$

- Ahora sabemos que $X(k)$ es un DFT de N muestras que se compone por dos DFTs de $\frac{N}{2}$ muestras cada una.
  $X(k)$ finalmente se representa de la siguiente forma:

$$X(k) = G(k) + W_N^k H(k)$$

- Como $G(k)$ y $H(k)$ tienen periodo $\frac{N}{2}$ y $X(k)$ tiene periodo N, esta última tiene 2 periodos de cada una, así que se calculará en 2 mitades de la forma:
  Donde k recorre de 0 a $\frac{N}{2}$
  -Primera mitad: $X(k) = G(k) + W_N^k H(k)$
  -Segunda mitad: $X(k + \frac{N}{2}) = G(k + \frac{N}{2}) + W_N^{k+\frac{N}{2}} H(k + \frac{N}{2})$

Como $G(k)$ y $H(k)$ son periódicas y su periodo es igual a $\frac{N}{2}$, se tienen 2 periodos de cada una para extraer la $X(k)$:

$$X(k) = G(k) + W_N^k H(k)$$

Debido a la periodicidad se puede dividir $X(k)$ y calcular la primera y segunda mitad de los valores así:
Evaluando primero de 0 a $\frac{N-1}{2}$

$$X(k+\frac{N}{2}) = G(k+\frac{N}{2}) + W_N^{k+\frac{N}{2}} H(k+\frac{N}{2})$$

Para la segunda mitad de muestras:
Sabemos que la siguiente equivalencia:

$$W_N = e^{-j\frac{2\pi}{N}} \rightarrow W_N^{k+\frac{N}{2}} = (e^{-j\frac{2\pi}{N}})^{k+\frac{N}{2}}$$

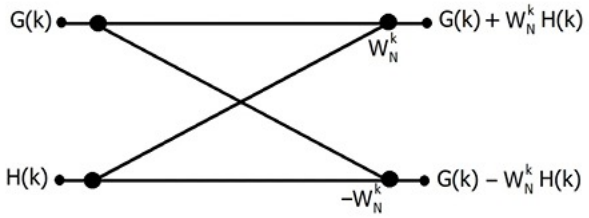
$$W_N^{k+\frac{N}{2}} = e^{-j\frac{2\pi}{N}k} \cdot e^{-j\frac{2\pi N}{N2}}$$

$$W_N^{k+\frac{N}{2}} = W_N^k \cdot e^{-j\pi}$$

Según la identidad de Euler: $e^{-j\pi} = -1$

$$W_N^{k+\frac{N}{2}} = -W_N^k$$

- Y al reemplazar la equivalencia obtenida, y separando la primera y segunda mitad de los valores de x(k) tenemos:
  -Primera mitad: $X(k) = G(k) + W_N^k H(k)$
  -Segunda mitad: $X(k) = G(k) - W_N^k H(k)$
- Podemos representar estas ecuaciones a través de un diagrama de flujo o también conocido como mariposa básica de Fourier, operando las últimas 2 muestras que obtuvimos a través de la recursividad



III. Ejemplo práctico

Utilizamos un archivo .py para leer el audio y convertirlo en un vector, este se exporta en un archivo txt y en un programa .c se realiza la transformada de Fourier para luego retornar el vector del audio y poder visualizar la gráfica en el archivo .py.

Código fourier.py

```
import numpy as np
import matplotlib.pyplot as plt
import scipy

def normalizeAudio(data):
    return np.float32((data / max(data)))

SAMPLE_FOR = 1 # in seconds
samplerate, data = scipy.io.wavfile.read(r'audio1.wav')
data = normalizeAudio(data[0:int(samplerate*SAMPLE_FOR)])


fft_out = np.fft.fft(data[0:100])
freq_vector = np.arange(0, 44100, 44100 / 100)
plt.plot(freq_vector, np.abs(fft_out))
plt.show()
```

Código fft.h

```
#ifndef _FFT_H_
#define _FFT_H_


#include "complex.h"


template <class T = double> class TFFT
{
public:
        static bool Forward(const complex<T>* const Input, complex<T>* const Output, const uns

        static bool Forward(complex<T>* const Data, const unsigned int N);
        static bool Inverse(const complex<T>* const Input, complex<T>* const Output, const uns
        static bool Inverse(complex<T>* const Data, const unsigned int N, const bool Scale = t

protected:

        static void Rearrange(const complex<T>* const Input, complex<T>* const Output, const u
        static void Rearrange(complex<T>* const Data, const unsigned int N);

        static void Perform(complex<T>* const Data, const unsigned int N, const bool Inverse =

        static void Scale(complex<T>* const Data, const unsigned int N);
};

#include "fft.cpp"

#endif
```

Código complex.h

```
#ifndef _COMPLEX_H_
#define _COMPLEX_H_

template <class T = double> class complex
{
protected:

        T m_re;
        T m_im;

public:
        static const complex<T> i;
        static const complex<T> j;


        complex() : m_re(T(0.)), m_im(T(0.)) {}
        complex(T re, T im) : m_re(re), m_im(im) {}
        complex(T val) : m_re(val), m_im(T(0.)) {}


        complex& operator= (const T val)
        {
                m_re = val;
```

```
        m_im = T(0.);
        return *this;
}


T re() const { return m_re; }
T im() const { return m_im; }


complex<T> conjugate() const
{
        return complex<T>(m_re, -m_im);
}

T norm() const
{
        return m_re * m_re + m_im * m_im;
}


complex<T> operator+ (const complex<T>& other) const
{
        return complex<T>(m_re + other.m_re, m_im + other.m_im);
}

complex<T> operator- (const complex<T>& other) const
{
        return complex<T>(m_re - other.m_re, m_im - other.m_im);
}

complex<T> operator* (const complex<T>& other) const
{
        return complex<T>(m_re * other.m_re - m_im * other.m_im,
                m_re * other.m_im + m_im * other.m_re);
}

complex<T> operator/ (const complex<T>& other) const
{
        const T denominator = other.m_re * other.m_re + other.m_im * other.m_im;
        return complex<T>((m_re * other.m_re + m_im * other.m_im) / denominator,
                (m_im * other.m_re - m_re * other.m_im) / denominator);
}

complex<T>& operator+= (const complex<T>& other)
{
        m_re += other.m_re;
        m_im += other.m_im;
        return *this;
}

complex<T>& operator-= (const complex<T>& other)
{
        m_re -= other.m_re;
        m_im -= other.m_im;
        return *this;
}

complex<T>& operator*= (const complex<T>& other)
```

```
{
        const T temp = m_re;
        m_re = m_re * other.m_re - m_im * other.m_im;
        m_im = m_im * other.m_re + temp * other.m_im;
        return *this;
}

complex<T>& operator/= (const complex<T>& other)
{
        const T denominator = other.m_re * other.m_re + other.m_im * other.m_im;
        const T temp = m_re;
        m_re = (m_re * other.m_re + m_im * other.m_im) / denominator;
        m_im = (m_im * other.m_re - temp * other.m_im) / denominator;
        return *this;
}

complex<T>& operator++ ()
{
        ++m_re;
        return *this;
}

complex<T> operator++ (int)
{
        complex<T> temp(*this);
        ++m_re;
        return temp;
}

complex<T>& operator-- ()
{
        --m_re;
        return *this;
}

complex<T> operator-- (int)
{
        complex<T> temp(*this);
        --m_re;
        return temp;
}

complex<T> operator+ (const T val) const
{
        return complex<T>(m_re + val, m_im);
}

complex<T> operator- (const T val) const
{
        return complex<T>(m_re - val, m_im);
}

complex<T> operator* (const T val) const
{
        return complex<T>(m_re * val, m_im * val);
}

complex<T> operator/ (const T val) const
```

```
{
        return complex<T>(m_re / val, m_im / val);
}

complex<T>& operator+= (const T val)
{
        m_re += val;
        return *this;
}

complex<T>& operator-= (const T val)
{
        m_re -= val;
        return *this;
}

complex<T>& operator*= (const T val)
{
        m_re *= val;
        m_im *= val;
        return *this;
}

complex<T>& operator/= (const T val)
{
        m_re /= val;
        m_im /= val;
        return *this;
}

friend complex<T> operator+ (const T left, const complex<T>& right)
{
        return complex<T>(left + right.m_re, right.m_im);
}

friend complex<T> operator- (const T left, const complex<T>& right)
{
        return complex<T>(left - right.m_re, -right.m_im);
}

friend complex<T> operator* (const T left, const complex<T>& right)
{
        return complex<T>(left * right.m_re, left * right.m_im);
}

friend complex<T> operator/ (const T left, const complex<T>& right)
{
        const T denominator = right.m_re * right.m_re + right.m_im * right.m_im;
        return complex<T>(left * right.m_re / denominator,
                -left * right.m_im / denominator);
}


bool operator== (const complex<T>& other) const
{
        return m_re == other.m_re && m_im == other.m_im;
}
```

```
        bool operator!= (const complex<T>& other) const
        {
                return m_re != other.m_re || m_im != other.m_im;
        }

        bool operator== (const T val) const
        {
                return m_re == val && m_im == (0.);
        }

        bool operator!= (const T val) const
        {
                return m_re != val || m_im != T(0.);
        }

        friend bool operator== (const T left, const complex<T>& right)
        {
                return left == right.m_re && right.m_im == T(0.);
        }

        friend bool operator!= (const T left, const complex<T>& right)
        {
                return left != right.m_re || right.m_im != T(0.);
        }
};

#include "complex.cpp"

#endif
#pragma once
```

Código complex.cpp

```
#ifndef _COMPLEX_CPP_
#define _COMPLEX_CPP_

#include "complex.h"

template<class T> const complex<T> complex<T>::i(T(0.), T(1.));
template<class T> const complex<T> complex<T>::j(T(0.), T(1.));

#endif
```

Código fourierSample.cpp

```
#include "fft.h"

#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <vector>

using namespace std;


int main(int argc, char* argv[])
{
```

```
        vector<double> data;
        string line;
        ifstream infile("test.txt");
        double numero;
        while (getline(infile,line)) {
                istringstream iss(line);

                if (iss >> numero) {
                        data.push_back(numero);
                }
        }


        complex<double> Signal1[100];
        for (int i = 0; i < data.size(); i++) {
                Signal1[i] = data[i];
        }


        complex<double> Signal[8] = {0., 1., 2., 3., 4., 5., 6., 7.};


        TFFT<double>::Forward(Signal1, 100);


        std::cout << "re: ";
        for (unsigned int i = 0; i < 100; ++i)
                std::cout << Signal1[i].re() << " ";

        std::cout << std::endl << "im: ";
        for (unsigned int i = 0; i < 100; ++i)
                std::cout << Signal1[i].im() << " ";
        std::cout << std::endl << std::endl;

        return 0;
}
```

Código fft.cpp

```
#ifndef _FFT_CPP_
#define _FFT_CPP_

#include "fft.h"
#include <math.h>


template <class T> bool TFFT<T>::Forward(const complex<T>* const Input, complex<T>* const Outp
{

        if (!Input || !Output || N < 1 || N & (N - 1))
                return false;
        Rearrange(Input, Output, N);
        Perform(Output, N);
        return true;
}


template <class T> bool TFFT<T>::Forward(complex<T>* const Data, const unsigned int N)
```

```
{

        if (!Data || N < 1 || N & (N - 1))
                return false;
        Rearrange(Data, N);
        Perform(Data, N);
        return true;
}


template <class T> bool TFFT<T>::Inverse(const complex<T>* const Input, complex<T>* const Outp
{

        if (!Input || !Output || N < 1 || N & (N - 1))
                return false;
        Rearrange(Input, Output, N);
        Perform(Output, N, true);
        if (Scale)
                TFFT<T>::Scale(Output, N);
        return true;
}


template <class T> bool TFFT<T>::Inverse(complex<T>* const Data, const unsigned int N, const b
{
        if (!Data || N < 1 || N & (N - 1))
                return false;
        Rearrange(Data, N);
        Perform(Data, N, true);
        if (Scale)
                TFFT<T>::Scale(Data, N);
        return true;
}

template <class T> void TFFT<T>::Rearrange(const complex<T>* const Input, complex<T>* const Ou
{
        unsigned int Target = 0;
        for (unsigned int Position = 0; Position < N; ++Position)
        {
                Output[Target] = Input[Position];
                unsigned int Mask = N;
                while (Target & (Mask >>= 1))
                        Target &= ~Mask;
                Target |= Mask;
        }
}

template <class T> void TFFT<T>::Rearrange(complex<T>* const Data, const unsigned int N)
{
        unsigned int Target = 0;
        for (unsigned int Position = 0; Position < N; ++Position)
        {
                if (Target > Position)
                {
                        const complex<T> Temp(Data[Target]);
                        Data[Target] = Data[Position];
                        Data[Position] = Temp;
                }
```

```
                unsigned int Mask = N;
                while (Target & (Mask >>= 1))
                        Target &= ~Mask;
                Target |= Mask;
        }
}

template <class T> void TFFT<T>::Perform(complex<T>* const Data, const unsigned int N, const b
{
        const T pi = Inverse ? T(3.14159265358979323846) : T(-3.14159265358979323846);

        for (unsigned int Step = 1; Step < N; Step <<= 1)
        {
                const unsigned int Jump = Step << 1;
                const T delta = pi / T(Step);
                const T Sine = sin(delta * T(.5));
                const complex<T> Multiplier(T(-2.) * Sine * Sine, sin(delta));
                complex<T> Factor(T(1.));
                for (unsigned int Group = 0; Group < Step; ++Group)
                {
                        for (unsigned int Pair = Group; Pair < N; Pair += Jump)
                        {
                                const unsigned int Match = Pair + Step;
                                const complex<T> Product(Factor * Data[Match]);
                                Data[Match] = Data[Pair] - Product;
                                Data[Pair] += Product;
                        }
                        Factor = Multiplier * Factor + Factor;
                }
        }
}


template <class T> void TFFT<T>::Scale(complex<T>* const Data, const unsigned int N)
{

        const T Factor = T(1.) / T(N);}
        for (unsigned int Position = 0; Position < N; ++Position)
                Data[Position] *= Factor;
}

#endif
```

REFERENCIAS

[1] M. Martínez (2021, Mayo 12). ¿Qué es la transformada de Fourier y para qué sirve? [online]. Available: https://www.nobbot.com/educacion/que-es-la-transformada-de-fourier-y-para-que-sirve/

[2] INTI AUDIO. Transformación rápida de Fourier FFT - Conceptos básicos. [online] Available: https://www.nti-audio.com/es/servicio/conocimientos/transformacion-rapida-de-fourier-fft

[3] A.L Schmidt. "FFT: Transformada Rápida de Fourier", Universidad Nacional del Sur, no. 3, pp. 1-3, Marzo 2013 [online] Available: http://lcr.uns.edu.ar/fvc/NotasDeAplicacion/FVC-Schmidt%20Ana%20Luc%C3%ADa.pdf

[4] ETHW. (2016, Marzo, 31). James W. Cooley [online] Available:https://ethw.org/James$_W$.$_C$$ooley$ $M.J.Spilsbury, A.Euceda(2016, Mayo, 10)TransformadaRápidadeFourier[online]Available$: $https$: $//www.camjol.info/index.php/fisica/article/view/8276/8495$

[6] A. M. Manzano. Transformada rápida de Fourier Implementación y algunas aplicaciones [online] Available: https://www.um.es/documents/118351/9850722/Martínez+Manzano+TF$_4$8705250$_v$2.$pdf/c44507c8 - e990 - 4aac - b282 - 927acadcedd1$
$J.A.Gordillo.TransformadaRápidadeFourierparte1.[online]Available : https : //www.youtube.com/watch?v = VM1i6 -_h G4kab_channel = JacquelineArzateGordillo$

[7] J. A. Gordillo. Transformada Rápida de Fourier (FFT) Parte2. [online] Available: https://www.youtube.com/watch?v=2d$_X auKxRDEab_channel = JacquelineArzateGordillo$
$J.A.Gordillo.TransformadaRápidadeFourier(FFT)Parte3.[online]Available : https : //www.youtube.com/watch?v = R_DNOVJWBBMab_channel = JacquelineArzateGordillo$

[8] J. A. Gordillo. Transformada Rápida de Fourier (FFT), Parte 4. [online] Available: https://www.youtube.com/watch?v=7-x-tMUmgQQab$_channel = JacquelineArzateGordillo$