Universidad Católica San Pablo

Análisis y diseño de algoritmos
Ciencias de la Computación

Transformada Rápida de Fourier

Autores:
Paolo Delgado Vidal
Fabian Concha Sifuentes
Joaquin Casusol Escalante

20 de junio de 2022

Índice

1. Abstract

Existen distintos tipos de problemas más que todo en la ingeniería como tal, ya sea resolver ecuaciones diferenciales o diseñar controladores clásicos de sistemas realimentados o generar diseños de filtros de radiotransitores. Yéndonos a otro apartado de la ingeniería también está el tratamiento digital de imágenes para mejorar ciertas zonas de una imagen para ciertas necesidades. El objetivo de esta investigación es determinar lo que es la Transformada de Fourier, y a partir de esto encontrar un método de solución de esta misma de una forma más simple utilizando la Transformada Rápida de Fourier (FFT) para poder pasar una señal de audio inicialmente siendo su dominio el tiempo a otro espacio en el cual su dominio vaya a ser la frecuencia y así obtener información no tan evidente en el dominio inicial.

2. Introducción

El descubrimiento de la transformada rápida de Fourier junto con su aplicación recursiva la podemos atribuir de alguna forma a Carl Friedrich Gauss el cuál en 1805 usó este algoritmo para interpolar las trayectorias de los asteroides Palas y Juno sin embargo, y es por esto que el descubrimiento no se le atribuyé completamente a él, su investigación no fue ampliamente reconocida y casi no se llegó a conocer como tal, es por esto que existen muchas fuentes que dicen que ni siquiera lo llegó a publicar.

Fue en 1965 donde la transformada rápida de Fourier se hizo popular debido a James Cooley de IBM y John Tukey de Princeton los cuales publicaron un artículo en 1965 reinventando el algoritmo y describiendo cómo realizarlo convenientemente en una computadora. Se dice que a Tukey se le ocurrió la idea durante una reunión con el presidente Kennedy, en la cual se estaba buscando formas para detectar pruebas de armas nucleares de la Unión Soviética usando sismómetros. Lo que pasaba es que estos sismómetros generarían series temporales sismológicas y debido a la gran cantidad de sensores y a la cantidad de tiempo se necesitaba un algoritmo rápido para detectar este tipo de pruebas.

Ahora, hablando del algoritmo como tal, la transformada rápida de fourier es un algoritmo que nos permite multiplicar dos polinomios de longitud n en un tiempo $O(nlogn)$, esto evidencia que es más rápido que la multiplicación convencional, la cual realiza las operaciones en un tiempo de $O(n^2)$. Así mismo, se puede multiplicar dos números grandes al reducirlos a multiplicar polinomios, por lo que también se pueden multiplicar dos números largos en un tiempo $O(nlogn)$ (donde n es el número de dígitos en los números). Este algoritmo de FFT se ejecuta en el ya mencionado tiempo $O(nlogn)$, pero bajo la condición de ser polinomios no mayores a un tamaño 10^5, pues al superar este número, la exactitud del algoritmo en sus valores flotantes se ve afectada.

3. Desarrollo

3.1. DFT - TRANSFORMADA DISCRETA DE FOURIER

Considerando para polinomios de grado $n-1$:

$$A(x) = a_0x^0 + a_1x^1 + a_2x^2 + ... + a_{n-1}x^{n-1}$$

Suponemos que n es una potencia de 2, en caso no sea así, realizamos la suma de términos faltantes a_ix^i, y declaramos los coeficientes a_i con valores 0.
La teoría de los números complejos nos dice que la ecuación $x^n = 1$ posee n soluciones complejas (llamadas n-ésimas raíces de la unidad), y las soluciones son de la forma:

$$w_{n,k} = e^{\frac{2k\pi i}{n}} conk = 0, ..., n-1$$

La transformada discreta de Fourier (DFT) del polinomio $A(x)$ (o de manera equivalente el vector de coeficientes $[a_0, a_1, a_2, ..., a_{n-1}]$ se define como los valores del polinomio en los puntos $x = w_{n,k}$, es decir, es el vector:

$$\begin{aligned} DFT(a_0, a_1, ..., a_{n-1}) &= (y_0, y_1, ..., y_{n-1}) \\ &= (A(w_{n,0}), A(w_{n,1}), ..., A(w_{n,n-1})) \\ &= (A(w_n^0), A(w_n^1), ..., A(w_n^{n-1})) \end{aligned}$$

3.2. Inversa DFT

De manera similar , se define la transformada de Fourier Discreta Inversa : La DFT inversa de valores del polinomio $[a_1, a_2, a_3, ..., a_{n-1}]$ son los coeficientes del polinomio

$$InversaDFT(y_0, y_1, ..., y_{n-1}) = (a_0, a_1, ..., a_{n-1})$$

Por lo tanto, si una DFT calcula los valores del polinomio en los puntos en las n-ésimas raíces, la DFT inversa puede restaurar los coeficientes del polinomio usando esos valores.

3.3. FFT

Entendamos que la Transformada Rápida de Fourier (FFT) es un método que permite calcular la DFT en tiempo $O(nlogn)$. Esto es debido a la técnica de recursividad y de dividir y conquistar.
Entonces, sea un polinomio $A(x)$, donde n es una potencia de 2 y $n > 1$, teniendo el vector:

$$A(x) = a_0x^0 + a_1x^1 + a_2x^2 + ... + a_{n-1}x^{n-1}$$

Lo dividiremos en dos partes, es decir, en dos sub-vectores. Dichos sub-vectores serán de los coeficientes de las posiciones pares y de los coeficientes de las posiciones impares:

$$A(x) = a_0x^0 + a_2x^1 + ... + a_{n-2}x^{\frac{n}{2}-1}$$

$$A(x) = a_1x^0 + a_3x^1 + ... + a_{n-1}x^{\frac{n}{2}-1}$$

A esos dos sub-vectores se les aplica el DFT mostrado anteriormente, dando como resultado el vector modificado tras el FFT mediante la unión de ambos representada en la siguientes fórmulas:
Para la primera mitad:

$$A(x) = A_0(x^2) + xA_1(x^2)\ en\ \frac{n}{2}$$

Que representado como vectores donde y_k *es* $A(x), y_k^0$ *es* A_0, *y* y_k^1 *es* A_1:

$$y_k = y_k^0 + w_n^k * y_k^1$$

Para la segunda mitad:

$$A(x) = A_0(x^2) - xA_1(x^2) \; en \; k + \frac{n}{2}$$

Que representado como vectores donde $y_{k+\frac{n}{2}}$ *es* $A(x)$, y_k^0 *es* A_0, *y* y_k^1 *es* A_1:

$$y_{k+\frac{n}{2}} = y_k^0 - w_n^k * y_k^1$$

3.4. Inversa FFT

Dado el vector $[y_0, y_1, y_2, ..., y_{n-1}]$ los valores del polinomio A de grado $n-1$ en los puntos $x = w_n^k$.
Queremos restaurar los coeficientes $[a_0, a_1, a_2, a_3, ..., a_{n-1}]$ del polinomio. A este problema se le llama interpolación. En este caso, dado que conocemos los valores de los puntos en las raíces de la unidad, podemos obtener un algoritmo, que es prácticamente igual a la FFT.
Según la definición del DFT, lo representaremos de forma matricial, de modo que quedaría de la siguiente manera: (Matriz de Vandermonde)

$$\begin{pmatrix} w_n^0 & w_n^0 & w_n^0 & w_n 0 & ... & w_n 0 \\ w_n^0 & w_n^1 & w_n^2 & w_n^3 & ... & w_n^{n-1} \\ w_n^0 & w_n^2 & w_n^4 & w_n^6 & ... & w_n^{2(n-1)} \\ w_n^0 & w_n^3 & w_n^6 & w_n^9 & ... & w_n^{3(n-1)} \\ . & . & . & . & ... & . \\ . & . & . & . & ... & . \\ . & . & . & . & ... & . \\ w_n^0 & w_n^{n-1} & w_n^{2(n-1)} & w_n^{3(n-1)} & ... & w_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ . \\ . \\ . \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ . \\ . \\ . \\ y_{n-1} \end{pmatrix}$$

- En álgebra lineal, es considerada una matriz que presenta una progresión geométrica en cada fila
- Estas matrices son útiles en la interpolación de polinomios, ya que resolviendo un sistema de ecuaciones, la matriz de Vandermonde de orden $n \times n$ es equivalente a encontrar los coeficientes a_j del polinomio

De esta manera, podemos calcular el vector $[a_0, a_1, a_2, a_3, ..., a_{n-1}]$ al multiplicar el vector $[y_0, y_1, y_2, ..., y_{n-1}]$ desde la izquierda con la inversa de la matriz:

$$\begin{pmatrix} \mathrm{a}_0 \\ \mathrm{a}_1 \\ \mathrm{a}_2 \\ \mathrm{a}_3 \\ . \\ . \\ . \\ \mathrm{a}_{n-1} \end{pmatrix} = \begin{pmatrix} \mathrm{w}_n^0 & \mathrm{w}_n^0 & \mathrm{w}_n^0 & \mathrm{w}_n 0 & ... & \mathrm{w}_n 0 \\ \mathrm{w}_n^0 & \mathrm{w}_n^1 & \mathrm{w}_n^2 & \mathrm{w}_n^3 & ... & \mathrm{w}_n^{n-1} \\ \mathrm{w}_n^0 & \mathrm{w}_n^2 & \mathrm{w}_n^4 & \mathrm{w}_n^6 & ... & \mathrm{w}_n^{2(n-1)} \\ \mathrm{w}_n^0 & \mathrm{w}_n^3 & \mathrm{w}_n^6 & \mathrm{w}_n^9 & ... & \mathrm{w}_n^{3(n-1)} \\ . & . & . & . & ... & . \\ . & . & . & . & ... & . \\ . & . & . & . & ... & . \\ \mathrm{w}_n^0 & \mathrm{w}_n^{n-1} & \mathrm{w}_n^{2(n-1)} & \mathrm{w}_n^{3(n-1)} & ... & \mathrm{w}_n^{(n-1)(n-1)} \end{pmatrix}^{-1} \begin{pmatrix} \mathrm{y}_0 \\ \mathrm{y}_1 \\ \mathrm{y}_2 \\ \mathrm{y}_3 \\ . \\ . \\ . \\ \mathrm{y}_{n-1} \end{pmatrix}$$

De este modo, conseguimos la siguiente fórmula:

$$a_k = \frac{1}{n} \sum_{j=0}^{n-1} y_j w_n^{-kj}$$

Dicha fórmula es similar a la fórmula para obtener los coeficientes de una Matriz de Vandermonde:

$$y_k = \sum_{j=0}^{n-1} a_j w_n^{kj}$$

Al ser casi iguales podemos interpretar que los coeficientes a_k se pueden encontrar con el mismo algoritmo de divide y vencerás, así como con la FFT, solo que en lugar de w_n^k,tenemos que usar w_n^{-k}, finalmente necesitamos dividir los coeficientes resultantes por n.
Por lo tanto, el cálculo de la DFT inversa es casi el mismo que el cálculo de la DFT, y también se puede realizar en un tiempo de $O(nlogn)$.

3.5. Convolución

En matemáticas, la convolución es una operación matemática en dos funciones (f y g) que produce una tercera función ($f * g$) que expresa cómo la forma de una es modificada por la otra. El término convolución se refiere tanto a la función de resultado como al proceso de cálculo. Se define como la

integral del producto de las dos funciones después de que una se invierte y se desplaza. La integral se evalúa para todos los valores de desplazamiento, produciendo la función de convolución.

En nuestro caso, realizamos la convolución a los dos polinomios que incluirán los coeficientes de esas funciones, a los cuales trataremos como señales, siendo la de entrada (audio), la repuesta al impulso o función impulso, y la salida (la función convolucionada), con la finalidad de la señal de salida del audio tenga las mismas características de espacio acústico que nuestra señal de entrada, pero modificadas de forma que se haya aplicado un "filtro" (alterar frecuencias,amplitud,etc) siendo que mantiene la esencia del audio original mas presente cambios en su escucha.

4. Código

Listing 1: fourier

```
#include <iostream>
#include <complex>
#include <vector>
#include "AudioFile.h"

using namespace std;
using cd = complex<double>;

const double PI = acos(-1);


void fft(vector<cd>& audio_samples, bool invert) {
    int n = audio_samples.size();
    //paso base para saber cuando la recursividad termina
    if (n == 1)
    {
        return;
    }
    //creamos los dos vectores para dividir el vector
       ingresante en 2
    vector<cd> audio_samples1(n / 2);
    vector<cd> audio_samples2(n / 2);
    //copiamos la informacion
    for (int i = 0; i < n / 2; i++)
    {
```

```
        audio_samples1[i] = audio_samples[2 * i];
        audio_samples2[i] = audio_samples[2 * i + 1];
    }
    //divide y venceras
    fft(audio_samples1, invert);
    fft(audio_samples2, invert);

    //formula FFT
    double ang = 2 * PI / n * (invert ? -1 : 1);
    //Variables complejas para guardar los resultados
    cd w(1), wn(cos(ang), sin(ang));


    for (int i = 0; 2 * i < n; i++) {
        audio_samples[i] = audio_samples1[i] + w *
           audio_samples2[i];
        audio_samples[i + n / 2] = audio_samples1[i] - w *
           audio_samples2[i];
        if (invert) {
            audio_samples[i] /= 2;
            audio_samples[i + n / 2] /= 2;
        }
        w *= wn;
    }
}
//filtro de convolucion
void convolucion(vector<cd>& vector_samples, vector<cd>&
   filtro_conv, vector<cd>& resultante,int sizex, int sizeh)
{
    int i;
    int j;
    for (i = 0; i < sizex; i++)
    {
        for (j = 0; j < sizeh; j++)
        {
            resultante[i + j] = resultante[i + j] +
               vector_samples[i] * filtro_conv[j];
        }
    }

}

int main()
{
    AudioFile<double> audio;
```

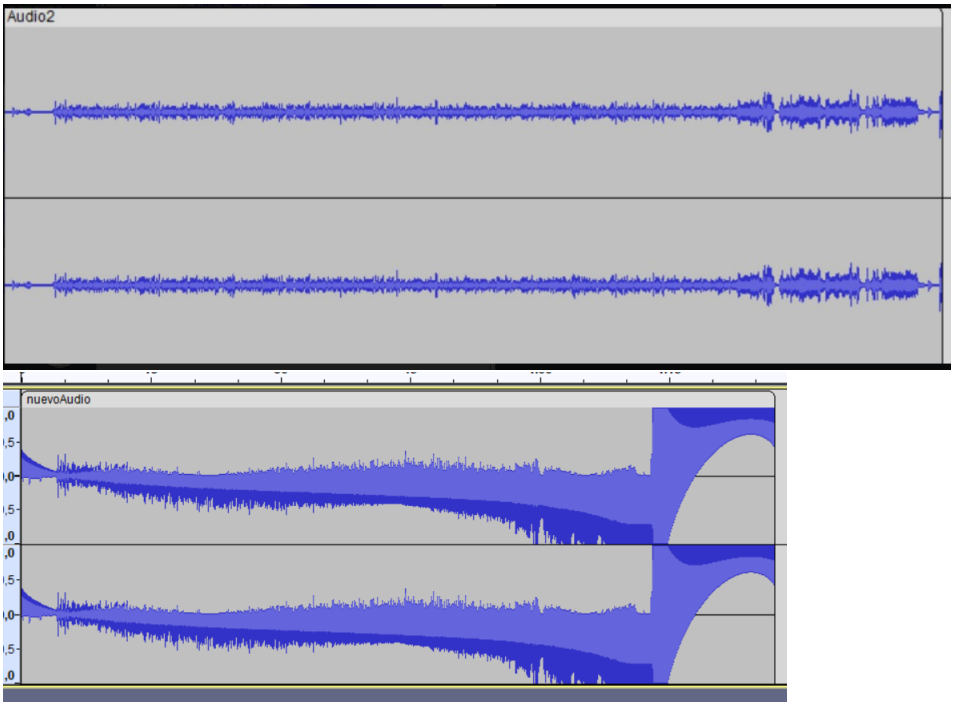
```
audio.load("C:/Users/Admin/Documents/ADA/Audio2.wav");
////////////////////lectura de audio y obtencion de
    samples//////////////
int numero_samples = audio.getNumSamplesPerChannel();
int conv = 3;
/////////////////llevamos el tamanho del array a la
    potencia de dos mas cercana al nro de samples//
int potencia = 2;
int potencia2 = 2;

while (numero_samples > potencia)
{
    potencia = potencia * 2;
}
while (conv + potencia - 1 > potencia2)
{
    potencia2 = potencia2 * 2;
}

//definimos los vectores que van a guardar los samples
//filtro de convolucion
//y el resultante de la convolucion de ambos
vector<cd> x(potencia);
vector<cd> h(conv);
vector<cd> y(potencia2);
//los llevamos a potencia de 2
for (int i = numero_samples; i < potencia; i++)
{
    x[i] = 1;
}
//le ponemos valores al vector que entrara a convolucion
    con los samples
h[0] = -0.01;
h[1] = 10;
h[2] = -0.01;
//copiamos los samples
for (int i = 0; i < numero_samples; i++)
{
    x[i] = audio.samples[0][i];
}
//los llevamos a potencia de 2
for (int i = conv + potencia - 1; i < potencia2; i++)
{
    y[i] = 1;
}
```

```
    ///pasarle la transformada rapida de fourier a estos
        valores
    fft(x, false);
    //cambiar los valores de las frecuencias
    convolucion(x, h, y, potencia, conv);

    //pasarle la inversa con los valores cambiados
    fft(y, true);
    audio.setNumSamplesPerChannel(potencia2);
    //exportar el nuevo audio
    for (int i = 0; i < potencia2; i++)
    {
        audio.samples[0][i]= y[i].real();
    }
    for (int i = 0; i < potencia2; i++)
    {
        audio.samples[1][i] = y[i].real();
    }
    audio.save("C:/Users/Admin/Documents/ADA/Audio_modificado
        .wav");
}
```



5. Conclusión

Con la Transformada Rápida de Fourier aplicada a nuestra muestra de audio hemos transformado el dominio de nuestros datos de un dominio de tiempo a un dominio de frecuencias, para que cierta información que pasaba desapercibida o que quizás no era lo suficientemente clara sea más entendible para nosotros, posteriormente mediante un filtro de convolución hemos modificado los datos numéricos del audio almacenados en un vector para después devolverle a esta muestra de audio el dominio de tiempo que tenía originalmente con la ayuda de la inversa de la Transformada Rápida de Fourier y poder ser exportada, escuchada y comparada con la original. En las pruebas mostradas nos damos cuenta de la evidente modificación de la frecuencia y amplitud de los valores del audio original, generando así una pista nueva y modificada.

6. Bibliografía

6.1. Librería AudioFile

Para la implementación del código, hemos empleado la librería AudioFile, la cuál fue implementada por Adam Stark y permite tanto la lectura como la escritura de archivos de audio WAV y AIFF, siendo incluida en el código mediante un archivo de cabecera (.h) e utilizada mediante el objeto *AudioFile*.
Esta librería trabaja a base de canales de audio y muestras del mismo, siendo para ello que tendremos que almacenar dentro de una matriz las mismas por cada canal.

6.2. Fuentes de Información

Referencias

[1] Marta Ruiz Costa-jussàHelenca Duxans Barrobés y H.D. Sherali, *Diseño y análisis de filtros en procesamiento de audio*, España, Catalunya, 2016.

[2] http://www.disca.upv.es/adomenec/IASPA/tema4/ES-Filtros.html,
Efectos de sonido Filtrado y ecualización.

[3] Adam Stark https://github.com/adamstark/AudioFile/blob/master/README.md,
Librería AudioFile.