Transformada Rápida de Fourier

Fast Fourier Transform

Integrantes:
Joaquin Casusol
Paolo Delgado
Fabián Concha

Índice

1. Introducción
2. Transformada discreta de Fourier (DFT)
3. Inversa DFT
4. Transformada Rápida de Fourier (FFT)
5. Inversa FFT
6. Código y librerías

Introducción

- El origen y primer uso de la transformada rápida de Fourier se le atribuye a Carl Friedrich Gauss en 1805 usó este algoritmo para interpolar las trayectorias de los asteroides Palas y Juno.
- Pero fue en 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
- La transformada rápida de Fourier tiene una gran amplitud de usos, ya que permite obtener la representación de información en el espacio de frecuencias y aplicando un operador en éste dominio, permite un mejor manejo de la información obtenida.

Transformada discreta de Fourier (DFT)

La transformada de Fourier es una operación matemática que transforma a una función del dominio del tiempo al dominio de la frecuencia, pero sin alterar su contenido de información, que nos permite multiplicar dos polinomios de longitud n.

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 a n como una potencia de 2, y en caso de no serlo, se haría una serie de sumas con los términos faltantes con un coeficiente de 0.

Transformada discreta de Fourier (DFT)

La transformada discreta de Fourier (DFT) del polinomio A(x) (o de manera equivalente el vector de coeficientes [a0, a1, a2, ..., an−1]) se define como los valores del polinomio en los puntos $x = w_n^k$, es decir, es el vector:

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

Transformada discreta de Fourier (DFT)

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:

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

Inversa de DFT

La inversa del DFT consiste en los coeficientes del polinomio:

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

Por ende, si el DFT calcula los valores del polinomio en las n-ésimas raíces, su inversa podrá restaurar los coeficientes del mismo empleando dichos valores.

Transformada rápida de Fourier (FFT)

Es un método que permite calcular la DFT en tiempo O(n log n).

Esto es debido a la técnica de recursividad y de dividir y conquistar.

En un polinomio A(x), donde n es una potencia de 2 y n > 1:

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

Lo dividiremos en dos partes, serán de los coeficientes de las posiciones pares y de los coeficientes de las posiciones impares:

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

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

Transformada rápida de Fourier (FFT)

Posteriormente, buscaremos una forma de hallar el DFT del vector total mediante la unión de los DFT de A0 y A1, empleando las siguientes fórmulas para ambas mitades:

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

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

Lo cual podemos expresarlo como vectores de yk(A(x)), yk0(A0), yk1(A1) y ykn/2(A(x)):

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

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

Inversa de 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 $[a_0, a_1, a_2, ..., a_{n-1}]$ del polinomio. A este problema se le llama interpolación.

Según la definición del DFT, lo representaremos de forma matricial, de modo que quedaría de la siguiente manera: (Matriz de Vandermonde)

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

Inversa de FFT

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 podremos encontrar los coeficientes a_j del polinomio,

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

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

Los coeficientes a_k se pueden encontrar con el mismo algoritmo de divide y vencerás, así como con la FFT.

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(n log n).

Código

```
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;
    }
}
```

Código

```
//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];
        }
    }

}
```

Código

```
///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/fabia/OneDrive/Escritorio/Fourier/nuevoAudio.wav");
```

```
AudioFile<double> audio;
audio.load("C:/Users/fabia/OneDrive/Escritorio/Fourier/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;
}
```

Conclusión

- Hemos transformado el dominio de nuestros datos de un dominio de tiempo a un dominio de frecuencias
- Modificamos los datos numéricos del audio almacenados en un vector para después devolverle a esta muestra de audio el dominio de tiempo que tenía originalmente
- Luego de este procedimiento hemos podido sacar a la luz información que no era tan clara en el audio original, sin embargo si en el audio exportado finalmente.