CIENCIA DE LA COMPUTACIÓN

Analisis y Diseño de Algoritmos

Transformada Rápida de Fourier

Aarón Misash Apaza Coaquira
Eileen Karin Apaza Coaquira
Brigham Jeffrey Caceres Gutierrez

CCOMP 5-1

"Los alumnos declaran haber realizado el presente trabajo de acuerdo a las normas de la Universidad Católica San Pablo"

1 Introducción

La transformada rápida de Fourier(FFT por sus siglas en inglés - fast Fourier transform), es una transformación matemática aplicada a curvas de señales sinusoidales. En palabras de G. D. Bergland:

"El algoritmo de la transformada rápida de Fourier puede reducir el tiempo necesario para encontrar la transformada discreta de Fourier de varios minutos a menos de un segundo, y puede reducir los costos de varios dólares a varios centavos"

Nos permite transformar una señal que está entre el dominio del tiempo y el dominio de la frecuencia.Es reversible ya que las señales se pueden transformar entre dominios. Es uno de los algoritmos más importantes, se usa para la compresión de imagen, compresión de audio, procesamiento de señal, computación científica de alto desempeño, etc.
El profesor Gilbert Strang del MIT describió la Transformada rápida de Fourier como:

" El algoritmo numérico más importante de la vida"

Pero, ¿Por qué la transformación de Fourier es tan central en nuestro mundo moderno? Cada vez que miras una imagen digital, cada vez que escuchas música digital y cada vez que reproduces un vídeo digital, un algoritmo de FFT se ejecuta en segundo plano.

2 Marco teórico

2.1 Serie de Fourier

- Ecuación original:

$$f(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} \left[a_n cos\left(\frac{n\pi t}{L}\right) + b_n sen\left(\frac{n\pi t}{L}\right) \right] \tag{1}$$

- Notación por integración:

$$F(\xi) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i x \xi} dx \tag{2}$$

$$f(x) = \int_{-\infty}^{\infty} F(\xi) e^{-2\pi i x \xi} \xi \tag{3}$$

Dónde $F(\xi)$ representa una función en el espacio de frecuencia, ξ representa un valor en el espacio de frecuencia, $f(x)$ representa una función en el espacio real, y x representa un valor en el espacio real.

2.2 DFT

- Para determinar el DTF de una señal discreta (donde es el tamaño de su dominio), multiplicamos cada uno de sus valores por elevado a alguna función. Luego, sumamos los resultados obtenidos para un . Si usáramos una computadora para calcular la Transformada Discreta de Fourier de una señal, necesitaría realizar N (multiplicaciones) x N (sumas) = $O(N^2)$ operaciones.

$$x\left[K\right]=\sum_{n=0}^{N-1}x\left[n\right]e^{\frac{-j2\pi kn}{N}} \tag{4}$$

$$\text{Donde} \quad k=0,....,N-1$$

3 Algoritmo de Cooley-Tukey

El problema con el uso de un DFT estándar es que requiere una gran multiplicación de matrices y sumas sobre todos los elementos, que son operaciones prohibitivamente complejas. El algoritmo de Cooley-Tukey calcula el DFT directamente con menos sumas y sin multiplicaciones matriciales.
Supongamos que separamos la Transformada de Fourier en subsecuencias indexadas pares e impares.

$$\begin{cases} \text{n=2r , si es par} \\ \text{n=2r+1, si es impar} \end{cases} \tag{5}$$

$$\text{Donde } r=1,2,...,\tfrac{N}{2}-1$$

Después de realizar un poco de álgebra, terminamos con la suma de dos términos. La ventaja de este enfoque radica en el hecho de que las subsecuencias indexadas pares e impares se pueden calcular simultáneamente.

$$x\left[k\right]=\sum_{r=0}^{\frac{N}{2}-1}x\left[2r\right]e^{\frac{-j2\pi k}{N}}+x\left[k\right]=e^{\frac{-j2\pi k}{N}}\sum_{r=0}^{\frac{N}{2}-1}x\left[2r+1\right]e^{\frac{-j2\pi k}{N/2}}$$

$$x\left[k\right]=x_{par}\left[k\right]+e^{\frac{-j2\pi k}{N}}x_{impar}\left[k\right] \tag{6}$$

El truco para el algoritmo de Cooley-Tukey es la recursión.

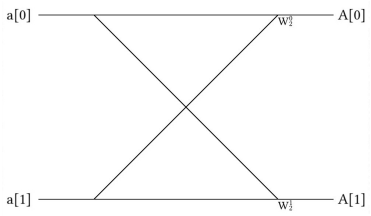
Con la recursión, podemos reducir la complejidad a O(nlogn), que es una operación factible.

Como nota al margen, estamos aplicando que la matriz debe tener una potencia de 2 para que la operación funcione. Esta es una limitación del hecho de que estamos usando la recursión y dividiendo la matriz en 2 cada vez; sin embargo, si su matriz no tiene una potencia de 2, simplemente puede rellenar el espacio sobrante con 0 hasta que su matriz tenga una potencia de 2.

4 Diagrama de mariposa

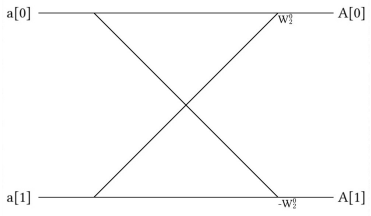
Es una porcion de la computación que combina los resultados de pequenas transformadas discretas de Fourier (DFT) en una DFT mas grande, o viceversa

Imagina que necesitamos realizar un FFT de una matriz de solo 2 elementos. Podemos representar esta adición con la siguiente mariposa (radix-2):



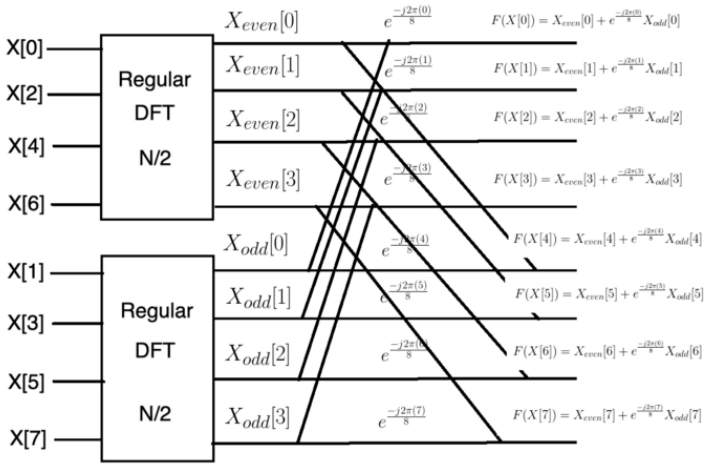
$$\begin{aligned} b_0 &= a_0 + \omega_0^2 a_1 \\ b_1 &= a_0 + \omega_1^2 a_1 \end{aligned} \tag{7}$$

Sin embargo, resulta que la segunda mitad de nuestra matriz de los valores son siempre los negativos de la primera mitad, por lo que 20=21, por lo que podemos utilizar el siguiente diagrama de mariposas:



$$\begin{aligned} b_0 &= a_0 + \omega_0^2 a_1 \\ b_1 &= a_0 - \omega_0^2 a_1 \end{aligned} \tag{8}$$

Y en cada paso, usamos el término apropiado. Por ejemplo, imagina que necesitamos realizar un FFT de una matriz de solo 2 elementos. Podemos representar esta adición con la siguiente mariposa (radix-2):



Podemos expresar las ganancias en términos de notación Big O de la siguiente manera. El primer término proviene del hecho de que calculamos la Transformada discreta de Fourier dos veces. Multiplicamos este último por el tiempo necesario para calcular la Transformada discreta de Fourier en la mitad de la entrada original. En el paso final, toma pasos para sumar la Transformada de Fourier para un . Explicamos esto agregando al producto final.

(Coste de DFT)

$$
\begin{aligned}
&2x\left(\frac{N}{2}\right)^2 + N \\
&= \frac{N^2}{2} + N \\
&O\left(\frac{N}{2}^2 + N\right) \\
&= O\left(N^2\right)
\end{aligned}
$$

$\therefore$ El coste es de $\mathrm{O}\left(N^2\right)$

Donde:

$2 \Rightarrow$ 2 DFT

$\left(\frac{N}{2}\right)^2 \Rightarrow$ DFT con N/2 elementos

$N \Rightarrow x\left[k\right] = x_{par}\left[k\right] + e^{\frac{-j2\pi k}{N}} x_{impar}\left[k\right]$

Observe cómo pudimos reducir el tiempo necesario para calcular la transformada de Fourier en un factor de 2. Podemos mejorar aún más el algoritmo aplicando el enfoque de dividir y conquistar, reduciendo a la mitad el costo computacional cada vez. En otras palabras, podemos continuar dividiendo el tamaño del problema hasta que nos quedemos con grupos de dos y luego calcular directamente las Transformadas discretas

de Fourier para cada uno de esos pares.

(Coste de FFT)

$$\frac{N}{2} \to 2(\frac{N}{2})^2 + N = \frac{N^2}{2} + N$$
$$\frac{N}{4} \to 2(2(\frac{N}{4})^2 + \frac{N}{2}) + N = \frac{N^2}{4} + 2N$$
$$\frac{N}{8} \to 2(2(2(\frac{N}{8})^2 + \frac{N}{2}) + \frac{N}{2}) + N = \frac{N^2}{8} + 3N$$
$$\vdots$$
$$\frac{N}{2^P} \to \frac{N^2}{2^P} + P^N = \frac{N^2}{N} + (\log_2 N)N = N + (\log_2 N)N$$
$$\sim O(N + N\log_2 N)$$
$$\sim O(N\log_2 N)$$

$\therefore$ El coste es de $O(N\log_2 N)$

5 Codigo

- DFT
  - Pseudocódigo:

Input: $s_{x,y-M}$, $s_{x-M,y}$, $d_{x,y-M}$, $S_{x-1,y}(u,v)$, $D_{x,y}(v)$
Output: $D_{x,y}(v)$, $S_{x,y}(u,v)$

Paso 1 $R_A = 1$
Computar $d_{x,y} = s_{x,y} - s_{x-M,y}$
Computar $c_{x,y} = d_{x,y} - d_{x,y-M}$
$C_M = M, R_A = M$
for v=0,1,..., M-1 **do**
    Computar $D_{x,y}(v) usando D_{x,y}(v) y e_{x,y}$
for v=0,1,...,M-1 **do**
    for u=0,1,...,M-1 do **do**
        Computar $S_{x,y}(u,v) usando S_{x-1,y}(u,v) y D_{x,y}(v)$

  - Código en c++:

```
#include <complex>
#include <windows.h>
#include <vector>

using namespace std;
```

```
6
7  const double PI = 3.14159265;
8
9  vector<complex<double>> m_DFT(vector<complex<double>>
       m_input) {
10   vector<complex<double>> m_output;
11   int m_N = m_input.size();
12
13   complex<double> m_suma(0, 0);
14   for (int m_k = 0; m_k < m_N; m_k++)
15   {
16     m_suma.real(0);
17     m_suma.imag(0);
18     for (int m_n = 0; m_n < m_N; m_n++)
19     {
20       double m_theta = -2 * PI * m_k * m_n / m_N;
21       complex<double> W_N(exp(complex<double>(0, m_theta))
       );
22       m_suma += m_input[m_n] * W_N;
23     }
24     m_output.push_back(m_suma);
25   }
26   return m_output;
27 }
```

- Cooley-Tukey recursivo
  - Pseudocódigo:

```
RECURSIVE-FFT(a)
 1  n = a.length              // n is a power of 2
 2  if n == 1
 3      return a
 4  ω_n = e^(2πi/n)
 5  ω = 1
 6  a^[0] = (a_0, a_2, ..., a_{n-2})
 7  a^[1] = (a_1, a_3, ..., a_{n-1})
 8  y^[0] = RECURSIVE-FFT(a^[0])
 9  y^[1] = RECURSIVE-FFT(a^[1])
10  for k = 0 to n/2 - 1
11      y_k = y_k^[0] + ω y_k^[1]
12      y_{k+(n/2)} = y_k^[0] - ω y_k^[1]
13      ω = ω ω_n
14  return y                  // y is assumed to be a column vector
```

  - Código en c++:

```
1
2  #include <complex>
3  #include <windows.h>
```

```cpp
#include <vector>

using namespace std;

const double PI = 3.14159265;

vector<complex<double>> rec_FFT(vector<complex<double>> m_input)
{
    int m_N = m_input.size();
    vector<complex<double>> m_output(m_N);

    if (m_N == 1)
    return m_input;

    complex<double> w_n(exp(complex<double>(0, (-2) * PI / m_N)));
    complex<double> w(1, 0);

    ////DIVIDE
    vector<complex<double>> pares(m_N / 2);
    vector<complex<double>> impares(m_N / 2);

    int indx = 0;
    for (int indxpar = 0; indxpar < m_N; indxpar += 2)
    {
      pares[indx] = m_input[indxpar];
      indx++;
    }

    indx = 0;
    for (int indximpar = 1; indximpar < m_N; indximpar +=2)
    {
      impares[indx] = m_input[indximpar];
      indx++;
    }

    //CONQUISTA
    vector<complex<double>> y_pares = rec_FFT(pares);
    vector<complex<double>> y_impares = rec_FFT(impares);

    //VENCERAS o COMBINARAS o MEZCLARAS
    vector<complex<double>> y_rec(m_N);
    for (int k = 0; k < m_N / 2; k++)
    {
      y_rec[k] = y_pares[k] + w * y_impares[k];
      y_rec[k + (m_N / 2)] = y_pares[k] - w * y_impares[k];
      w = w * w_n;
    }
    return y_rec;
}
```

- Cooley-Tukey iterativo
  - Pseudocódigo:

RECURSIVE-FFT(a)

```
1  n = a.length                    // n is a power of 2
2  if n == 1
3      return a
4  ω_n = e^(2πi/n)
5  ω = 1
6  a^[0] = (a_0, a_2, ..., a_{n-2})
7  a^[1] = (a_1, a_3, ..., a_{n-1})
8  y^[0] = RECURSIVE-FFT(a^[0])
9  y^[1] = RECURSIVE-FFT(a^[1])
10 for k = 0 to n/2 - 1
11     y_k = y_k^[0] + ω y_k^[1]
12     y_{k+(n/2)} = y_k^[0] - ω y_k^[1]
13     ω = ω ω_n
14 return y                        // y is assumed to be a column vector
```

– Código en c++:

```
#include <complex>
#include <vector>
#include <bitset>
#include <windows.h>

using namespace std;

const double PI = 3.14159265;

string invertirCadena(string cadena)
{
  string invertido = cadena;
  int tamano = cadena.size();
  for (int x = 0, y = tamano - 1; x < tamano; x++, y--)
    invertido[x] = cadena[y];
  return invertido;
}

void bitReverseCopy(vector<complex<double>> &a, vector<
    complex<double>> &A)
  {
  int n = a.size();
  for (int k = 0; k < n; k++)
  {
    string binary = bitset<2>(k).to_string();
    binary = invertirCadena(binary);
    int decimal = int(bitset<2>(binary).to_ulong());
    A[decimal] = a[k];
  }
}

vector<complex<double>> ite_FFT(vector<complex<double>>
    m_input)
{
```

```
  int n = m_input.size();
  vector<complex<double>> A(n);
  bitReverseCopy(m_input, A);

  for (int s = 1; s <= log2(n); s++)
  {
    double m = pow(2, s);
    complex<double> w_m(exp(complex<double>(0, -2 * PI / m
    )));
    for (int k = 0; k < n; k+=m)
    {
      complex<double> w(1, 0);
      for (int j = 0; j < m / 2; j++)
      {
        complex<double> t = w * A[k + j + (m / 2)];
        complex<double> u = A[k + j];
        A[k + j] = u + t;
        A[k + j + (m / 2)] = u - t;
        w = w * w_m;
      }
    }
  }

  return A;
}
```

6 Conclusión

- La Transformada Rápida de Fourier ha llevado a una revolución en el procesamiento de senales digitales.
- FFT es un algoritmo optimizado para implementar la "Transformación Discreta de Fourier". En este proceso, una sección limitada en el tiempo de una señal se descompone en sus componentes individuales para poder analizarlos.
- La DFT nos brinda una manera segura de transformar secuencias con variable independiente discreta y de duracion finita, en otras secuencias discretas finitas tambien, sin embargo al recibir como entrada una secuencia de numeros reales y complejos, se requiere una matriz muy grande de multiplicaciones y sumas sobre todos los elementos, siendo un conjunto de operaciones complejas,

References

[1] What makes a Fourier Transform Fast?,Url: https://www.algorithm-archive.org/contents/cooley$_t$ukey/cooley$_t$ukey.html

[2] Youtube, url:https://www.youtube.com/watch?v=7JIiQiO-wqU

[3] Transformada Rápida de Fourier,url:https://www.researchgate.net/publication/336339137$_T ransformada_R apida_d eF ou$

[4] Fast Fourier Transform,url:https://towardsdatascience.com/fast-fourier-transform-937926e591cb