Proyecto CPP2020

Daniel Reyes Barrera

18 de diciembre de 2020

Resumen

En este documento se ha programado dos algoritmos para calcular la Tranformada de Fourier, la DFT (Discrete Fourier Transform) y la FFT (Fast Fourier Transform) comparando los tiempos de ejecucion para distintos N números de muestras.

1. Discrete Fourier Transform

La DFT transforma una función matemática en otra, obteniendo una representación en el dominio de la frecuencia, siendo la función original una función en el dominio del tiempo.

La secuencia de N números complejos $x_0, ..., x_{N-1}$ se transforma en la secuencia de N números complejos $X_0, ..., X_{N-1}$ mediante la DFT con la fórmula

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-\frac{i2\pi}{N}kn} \qquad k = 0, ..., N-1$$

1.1. Problema computacional.

Objetivo: Calcular la transformada de fourier da una función dada.

Entrada: Un número entero N representando la cantidad de muestras.

Salida: El tiempo de ejecución de cada N muestra.

1.2. Algoritmo.

Para el algoritmo se utilizó una función de prueba $f(x) = 2sin(2\pi x) + 5cos(2\pi x)$ y se usó la libreria `complex` para manipular números complejo. Siguiendo la formula como se define la DFT el algoritmo es sensillo de programar.

El código fuente del programa se muestra en el apéndice 5.

1.3. Instancia del problema.

Como prueba de escritorio, se seleccionó la siguiente instancia del problema. Entrada: $N = 65536$. La salida del programa se observa en la Figura 1.

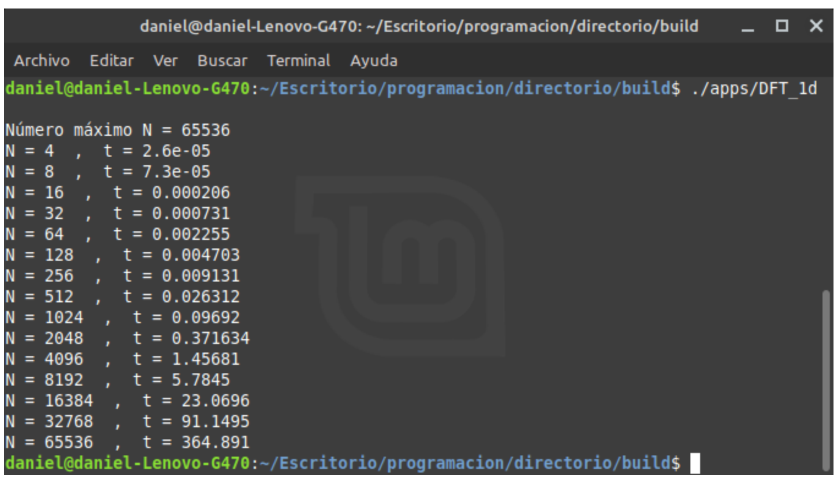

Figura 1: Ejecución del programa.

2. Fast Fourier Transform

La FFT no es una nueva transformada sino que se trata de un algoritmo para el cálculo de la Transformada Discreta de Fourier (DFT). Su importancia radica en el hecho que elimina una gran parte de los cálculos repetitivos a que está sometida la DFT, por lo tanto se logra un cálculo más rápido. Además, la FFT generalmente permite una mayor precisión en el cálculo de la DFT disminuyendo los errores de redondeo.

2.1. Problema computacional.

Objetivo: Calcular Transformada de fourier mediante FFT.

Entrada: Un número entero N representando la cantidad de muestras.

Salida: El tiempo de ejecución de cada N muestra.

2.2. Algoritmo.

Al igual que el DFT se utilizó la misma función de prueba y mismas librerias junto con los metodos de la clase `Complex` para la programación del algoritmo FFT.

2.3. Instancia del problema.

Como prueba de escritorio, se seleccionó la siguiente instancia del problema. Entrada: $N = 33554432$, el cual es significativamente mayo que la instancia en el DFT ya que el algoritmo FFT es mucho más eficiente. La salida del programa se observa en la Figura 1.

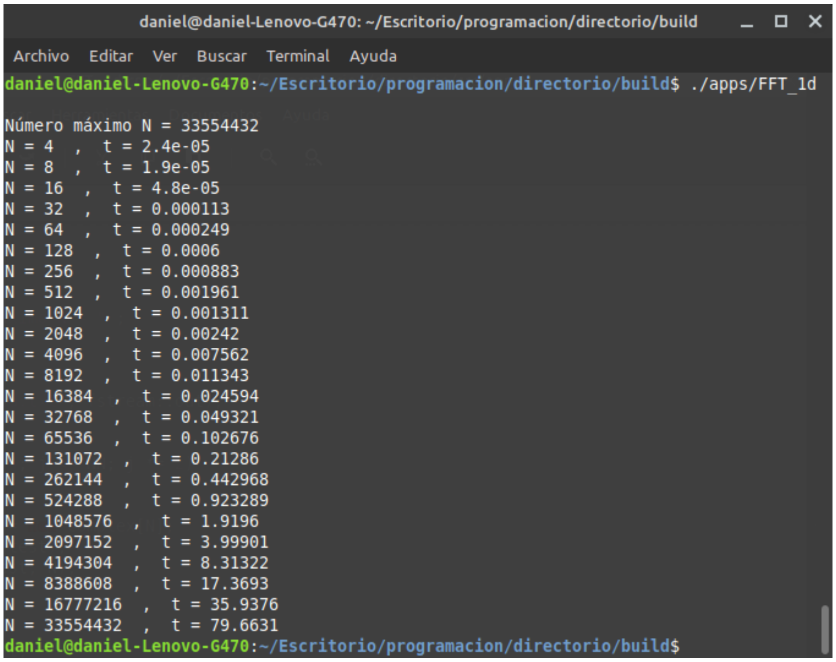

Figura 2: Ejecución del programa.

3. Comparación entre DFT y FFT

Programe el algoritmo para determinar si el número es palíndromo y el algoritmo para validar la entrada en métodos independientes. Sugerencia: Haga uso de los operadores módulo y división para separar el número tecleado en unidades, decenas, centenas, etc.

3.1. Instancia del problema.

Como prueba de escritorio, se seleccionaron las siguientes instancias del problema. Entrada: 12344321, 2234321, 76567 y 12324. La salida del programa se observa en la Figura **??**.

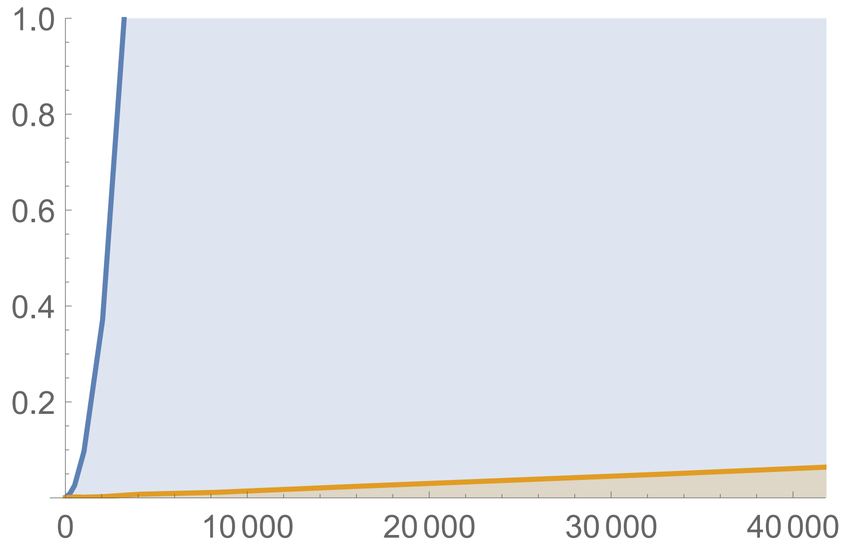

Figura 3: Comparación de datos con un rango menor.

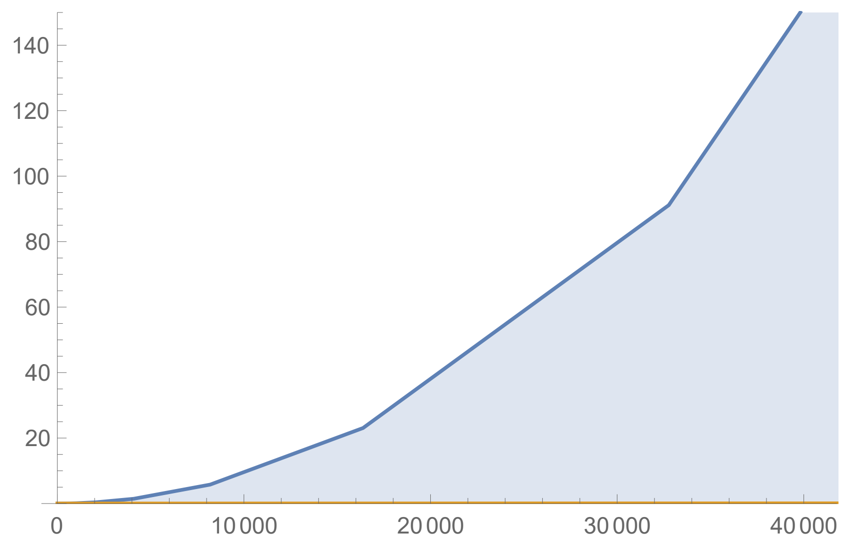

Figura 4: Comparación de datos con un rango mayor.

4. Conclusiones.

La FFT es de gran importancia en una amplia variedad de aplicaciones, desde el tratamiento digital de señales y filtrado digital en

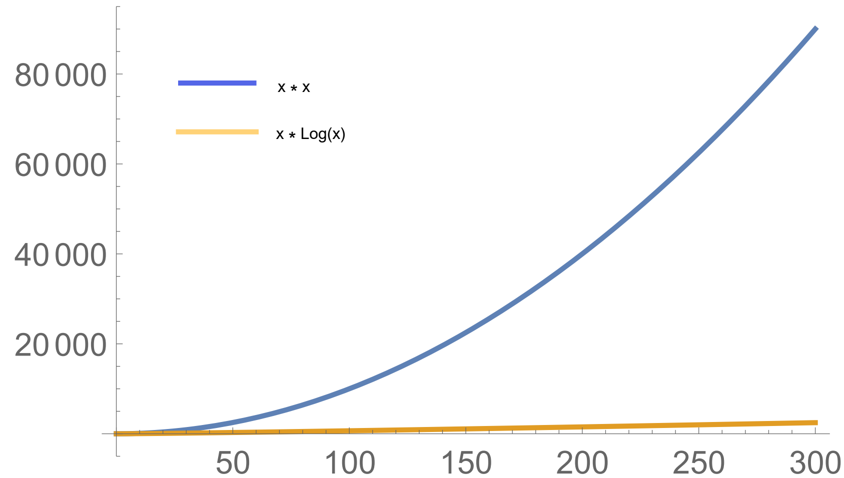

Figura 5: Comparación entre dos funciones.

general a la resolución de ecuaciones en derivadas parciales o los algoritmos de multiplicación rápida de grandes enteros.

5. Código fuente de DFT

```
#include <complex>
#include <iostream>
#include <valarray>
#include <time.h>
#include<fstream>

using namespace std;

//! Declarar una variable global N para ser utilizada en todo el program
int N = 0;

//! Definimos el tipo de variable compleja.
typedef complex<double> Complex;
//! Definimos el tipo de array que utilizaremos.
typedef valarray<Complex> CArray;


double funcion(double);
```

```
//! Obtenemos todos los pontos de nuestra funcion dado N.
template <class T>
void obtener_datos(T data[])
{
    for (int i = 0; i < N; i++)
    {
        data[i] = funcion(i);
    }
}

//! Declaramos la funcion DFT.
void DFT(CArray &, Complex[]);

int main()
{
    cout << "Numero maximo N = " ;
    int NN;
    cin >> NN;
    fstream myfile;
    myfile.open("DFT_id.txt", fstream::out);
    clock_t t;
    for (int j = 4; j <= NN; j *=2)

    {
        N = j;
        Complex *test = new Complex[N];
        obtener_datos(test);
        CArray data(test, N);
        t = clock();
        DFT(data, test);
        t = clock() - t;
        cout << "N = " << N << "  ,  t = " << ((float)t) / CLOCKS_PER_SEC
        myfile << "{" << N << ", " << ((float)t) / CLOCKS_PER_SEC << " }
        delete [] test;
    }
    myfile.close();

    return 0;
}
```

```

//! Definimos una funcion como prueba.
/*!
\param[double] Un punto sobre el dominio de la funcion.
\return Un valor tipo double que es el valor de la funcion en tal punto.
*/
double funcion(double x)
{
    return 2 * sin(2 * M_PI / N * x) + 5 * cos(2 * M_PI / N * x);
}

//! Definimos la estructura de la funcion DFT.
/*!
\param[CArray] Dos arrays, el primero como referencia para escribir los
*/
void DFT(CArray &x, Complex test[])
{

    for (int i = 0; i < N; i++)
    {
        double suma_r = 0;
        double suma_i = 0;
        for (int k = 0; k < N; k++)
        {
            suma_r += test[k].real() * cos(2 * M_PI / N * k * i);
            suma_i -= test[k].real() * sin(2 * M_PI / N * k * i);
        }
        x[i].real(suma_r);
        x[i].imag(suma_i);
    }
}
```

6. Código fuente de FFT

```
#include <complex>
#include <iostream>
#include <valarray>
#include <time.h>
#include <fstream>

```

```
using namespace std;

//! Declarar una variable global N para ser utilizada en todo el program
int N = 8;

//! Definimos el tipo de variable compleja.
typedef complex<double> Complex;
//! Definimos el tipo de array que utilizaremos.
typedef valarray<Complex> CArray;

double funcion(double);

//! Obtenemos todos los pontos de nuestra funcion dado N.
template <class T>
void obtener_datos(T data[])
{
    for (int i = 0; i < N; i++)
    {
        data[i] = funcion(i);
    }
}

//! Declaramos la funcion FFT.
void FFT(CArray &);

int main()
{
    cout << "Numero maximo N = " ;
    long long int NN;
    cin >> NN;
    fstream myfile;
    myfile.open("FFT_1d.txt", fstream::out);
    clock_t t;

    for (long int j = 4; j <= NN; j *= 2)
    {
        N = j;
        Complex *test = new Complex[N];
        obtener_datos(test);
        CArray data(test, N);
```

```

        t = clock();
        FFT(data);
        t = clock() - t;
        cout << "N = " << N << "  ,  t = " << ((float)t) / CLOCKS_PER_SE
        myfile << "{" << N << ", " << ((float)t) / CLOCKS_PER_SEC << " }
        delete [] test;
    }
    myfile.close();

    return 0;
}

//! Definimos una funcion como prueba.
/*!
\param[double] Un punto sobre el dominio de la funcion.
\return Un valor tipo double que es el valor de la funcion en tal punto.
*/
double funcion(double x)
{
    return 2 * sin(2 * M_PI / N * x) + 5 * cos(2 * M_PI / N * x);
}

//! Definimos la estructura de la funcion FFT.
/*!
\param[CArray] Un array complejo como referencia para hacer los calculos
*/
void FFT(CArray &x)
{
    const size_t N = x.size();
    if (N <= 1)
        return;

    //! Dividimos
    CArray even = x[slice(0, N / 2, 2)];
    CArray odd = x[slice(1, N / 2, 2)];

    //! Recurcividad
    FFT(even);
    FFT(odd);
```

```

    //! Combinamos
    for (size_t k = 0; k < N / 2; ++k)
    {
        Complex t = polar(1.0, -2 * M_PI * k / N) * odd[k];
        x[k] = even[k] + t;
        x[k + N / 2] = even[k] - t;
    }
}
```