TP1 de Algoritmos y Programación:  
Complejidad Computacional - Transformada Rápida de Fourier

Diego Esp. Desp@FI.UBA.AR

Facultad de Ingeniería

Universidad de Buenos Aires

Marcelo Bz mBz@FI.UBA.AR

Facultad de Ingeniería

Universidad de Buenos Aires

Resumen

El objetivo de este TP es ganar familiaridad con los conceptos propios de la programación en C++, análisis de la complejidad temporal y análisis de la complejidad espacial, implementando un programa que permita computar la transformada discreta de Fourier (DFT), la transformada discreta de Fourier inversa (iDFT), la transformada rápida discreta de Fourier (FFT), y la transofrmada rápida inversa discreta de Fourier (iFFT) para un conjunto de señales discretas complejas de variable entera, y su inversa correspondiente.

# Introducción

Se pide dar implementaciones de la DFT (Discrete Fourier Transform) y la iDFT (Inverse Discrete Fourier Transform) [1], mediante el algoritmo tradicional (cfr. TP0) y mediante un nuevo algoritmo, que emplea la estrategia de Dividir y Conquistar, llamado Algoritmo de la Transformada Rápida de Fourier (FFT).

**DFT**Recordando la definición de la DFT, tenemos por definición la transformación (DFT) de una secuencia de N puntos, x0, ..., xN-1 definida formalmente mediante:

,

con xn números complejos. Observar que la DFT es una función F: (N→C) → (N→C), es decir, dada una secuencia de números complejos, devuelve otra secuencia de números complejos.  
  
**iDFT**

Análogamente, tenemos la iDFT, la transformación inversa a la anterior- para una secuencia de N puntos, x0, ..., xN-1, definida mediante:

,

con xn números complejos. Observar que la iDFT es también una función F-1: (N→C) → (N→C), es decir, dada una secuencia de números complejos, devuelve otra secuencia de números complejos. Ambas verifican además la propiedad:

F[ F-1[ (xk) ] ] = F-1[ F[ (xk) ] ] = (xk) [1], la cual se conoce como "Definición de la inversa".

Mencionamos nuevamente que la DFT verifica las siguientes propiedades:

F[ F [ (xk) ] ] = (x-k) [2] (principio de dualidad)

|| (xk) || = n.|| F(x-k) || [3] (identidad de Parseval)

Tanto la DFT como la iDFT verifican las mismas propiedades de simetría conjugada, linealidad, desplazamiento y modulación que su contraparte continua.

En el TP0 se hizo el análisis de la complejidad temporal de la implementación de la DFT y de la iDFT mediante fuerza bruta, arrojando que la complejidad temporal de estas implementaciones por fuerza bruta son O(n2). Desde luego, tal tiempo es impráctico cuanto menos, lo cual motiva la búsqueda de algoritmos de mejor performance.

A continuación presentaremos los algoritmos de la FFT y la iFFT.

**FFT y iFFT: Versiones rápidas de la transformada discreta de Fourier**

**FFT (Fast Fourier Transform)**

Se realiza la siguiente implementación de la DFT, la cual permite eficientemente calcular la DFT de una secuencia de N números, (xk) que verifique la fórmula:

Se aplica la técnica recursiva de Dividir y Conquistar separando el vector de entrada x en dos subvectores, p y q, de índices pares e impares respectivamente, y se puede verificar la siguiente ecuación recursiva:

X(k) = P(k) + WkN . Q(k), k=0,1, ..., N-1 (\*1)

donde P es la transformada de p y Q es la transformada de q. Aplicando recursivamente el algoritmo, tenemos:

vector fft(vector x)

{

if ((N = x.length()) >= 2)

{

p = componentes de x con índice par;

q = componentes de x con índice impar;

P = fft(p);

Q = fft(q);

for (k = 0; k < N; ++k)

X[k] = P[k] + W(k, N) \* Q[k];

}

else

{

X = x;

}

return X;

}[[1]](#footnote-2)

**iFFT (Inverse Fast Fourier Transform)**

Formalmente, se define la iDFT como la transformada G que satisface:

F[G[a] ] = G[ F[a] ] = a para todo vector de números complejos a, donde F es la transformada discreta de Fourier. La iFFT es una implementación rápida nuevamente de la iDFT.

Similar al caso anterior, tendremos:

vector ifft(vector x)

{

if ((N = x.length()) >= 2)

{

p = componentes de x con índice par;

q = componentes de x con índice impar;

P = ifft(p);

Q = ifft(q);

for (k = 0; k < N; ++k)

X[k] = P[k] + W(k, N) \* Q[k];

}

else

{

X = x;

}

return X;

}

lo cual arroja la misma ecuación para X(k):

X(k) = P(k) + WkN . Q(k), k=0,1, ..., N-1 (\*2)

Luego, para determinar la complejidad de este método basta determinar la complejidad de (\*1).

# Descripción de la solución

Claramente vamos a querer separar en esta solución la lógica concerniente a las siguientes entidades:

*Complejo* Entidad encargada de manejar la aritmética, representación e impresión de números complejos.

*Vector* Contenedor para manejo de secuencias indizables de números complejos

*Vector* Entidad encargada de manejar las operaciones sobre conjuntos indizables de datos, en una sola dimensión, definida por la axiomática formal de [4]

*DFTcalculator* Entidad encargada de efectuar las dos operaciones que pide el TP, calcular la DFT y la iDFT. Esta clase contiene además las implementaciones de la FFT y la iFFT.

*cmdline* Entidad encargada de manejar los parámetros de entrada y salida a través de la consola

*UnitTests* Clase de pruebas unitarias con los diversos métodos de pruebas sobre las distintas implementaciones de la transformada.

(main) El punto de partida del progama en sí.

# Instrucciones de compilación

## Bajo plataforma Windows

La implementación sólo hace uso de características ANSI, no nativas de Windows. Para compilar con Visual Studio (2005 en adelante), efectuar:

cl /EHsc main.cpp

desde la consola Visual Studio Command Prompt ([2]), lo cual produce el archivo ejecutable main.exe.

## Bajo plataforma Linux/Unix-like

Nuevamente, la implementación sólo hace uso de características ANSI, portables a cualquier implementación ANSI de C++. Con lo cual, para compilar con gcc basta con efectuar:

gcc -g main.cpp -o main

lo cual producirá el archivo ejecutable main [3].

## Instrucciones de uso

Las opciones de uso son:

main -i *stream\_entrada* -o *stream\_salida* -m *método*

donde *stream\_entrada* es el archivo físico o stream de entrada sobre el cual se va a calcular el método especificado en *método*, y stream\_salida es el archivo físico o stream de salida sobre el cual se va a calcular el método especificado en *método*.

*método* es alguno de {fft, ifft, dft, idft}.

# Estructura de la solución

Damos a continuación el gráfico con la estructura de la solución.

# Análisis de la Complejidad Temporal

## bruteForceAlgorithm (DFT/iDFT)

Tenemos la siguiente implementación:

static void bruteForceAlgorithm(const vector<complex> & data , vector<complex> & result , string algorithm){

int N = data.length(); //O(1)

int sign = 1; //O(1)

if(algorithm == "idft"){ //O(1)

sign = -1;

}

for(int i = 0 ; i < N ; i++){

complex sum = 0; //O(1)

for(int j = 0 ; j < N ; j++){

double re = cos(2\*M\_PI\*j\*i\*sign/N);

double im = sin(2\*M\_PI\*j\*i\*sign/N);

complex W = complex(re,im);

sum = sum + (data[j]\*W);

}

if(algorithm == "idft"){

sum = sum / N;

}

result.pushBack(sum);

}

}

En particular, si definimos N=Tamaño(data) como el tamaño del problema en cuestión, buscamos caracterizar T(N), la función de tiempo de N. Marcamos en el código con O(1) todas las operaciones de complejidad constante (que no depende de N). En particular, vemos que las primeras tres sentencias son operaciones que no dependen del tamaño de la entrada, con lo cual son operaciones O(1).

Luego tenemos un bucle for, cuya condición de corte viene dada por:

for(int i = 0 ; i < N ; i++)....

Esto es, este for será recorrido exactamente N veces. Dentro de ese for tenemos primero una operación de complejidad temporal constante (a saber, la instanciación de un complejo llamado sum e inicialización al complejo cero, la cual es una operación que no depende de N). Inmediatamente después, tenemos otro for anidado, cuya condición de corte/recorrido viene dada por:

for(int j = 0 ; j < N ; j++)...

Es decir, este for será recorrido nuevamente unas N veces exactamente. Vale aclarar que la variable j no depende de i. Luego, vale el principio de multiplicación y tenemos que:

Si la complejidad dentro del segundo for es O(f), luego la complejidad del bloque completo que incluye a los dos for:

for(int i = 0 ; i < N ; i++){ //O(n2)

complex sum = 0; //O(1)

for(int j = 0 ; j < N ; j++){ //O(n)

double re = cos(2\*M\_PI\*j\*i\*sign/N); //O(1)

double im = sin(2\*M\_PI\*j\*i\*sign/N); //O(1)

complex W = complex(re,im); //O(1)

sum = sum + (data[j]\*W); //O(1)

}

if(algorithm == "idft"){ //O(1)

sum = sum / N;

}

result.pushBack(sum); //O(1)

}

será N.N.O(f) = N2.O(f) (principio de multiplicación).

Es trivial darse cuenta que la complejidad dentro de este segundo for anidado es O(1). En efecto, las operaciones:

double re = cos(2\*M\_PI\*j\*i\*sign/N); //O(1)

double im = sin(2\*M\_PI\*j\*i\*sign/N); //O(1)

complex W = complex(re,im); //O(1)

sum = sum + (data[j]\*W); //O(1)

son todas operaciones de complejidad constante O(1).

Finalmente, provisto que la implementación del contenedor vector implemente la operación pushBack() en O(1)[[2]](#footnote-3), tenemos que la complejidad total dentro del bloque es O(1).

Luego, retomando, por principio de multiplicación, los dos for's anidados tienen complejidad N2, y por tanto el algoritmo *bruteForceAlgorithm* tiene complejidad temporal N2.

## FFTAlgorithm (FFT, iFFT)

Tenemos la siguiente implementación:

static void FFTAlgorithm(const vector<complex> & data , vector<complex> & result , vector<int> & indexes,string algorithm){ //T(n)

int sign = 1; //O(1)

if(algorithm == "ifft"){ //O(1)

sign = -1;

}

// Si todavía no llegué al caso base llamo recursivamente

if(indexes.length() != 1){

vector<int> evenIndexes = vector<int>(); //O(1)

vector<int> oddIndexes = vector<int>(); //O(1)

//Guardo los índices pares e impares en los respectivos vectores

for(int i = 0 ; i < indexes.length() ; i = i + 2){ //O(n/2)

evenIndexes.pushBack(indexes[i]); //O(1)

oddIndexes.pushBack(indexes[i+1]); //O(1)

}

//Creo dos vectores que almacenarán los resultados de las transformadas

//de los índices pares e impares.

vector<complex> evenResult = vector<complex>(); //O(1)

vector<complex> oddResult = vector<complex>(); //O(1)

//Llamo recursivamente para calcular la DFT sobre la sub secuencia

FFTAlgorithm(data,evenResult,evenIndexes,algorithm); //T(n/2)

FFTAlgorithm(data,oddResult,oddIndexes,algorithm); //T(n/2)

//Uno la DFT de ambas sub secuencias

int N = evenResult.length() + oddResult.length(); //O(1)

for(int i = 0 ; i < N ; i++){ //O(n)

double re = cos(sign\*2\*M\_PI\*i/N); //O(1)

double im = sin(sign\*2\*M\_PI\*i/N); //O(1)

complex W(re,im); //O(1)

complex Xk = W\*oddResult.getCircular(i) + evenResult.getCircular(i); //O(1)

result.pushBack(Xk); //O(1)

}

}

else{

//Obtengo el dato que necesito a partir del vector de índices

int dataIndex = indexes[0]; //O(1)

complex dataComplex = data[dataIndex]; //O(1)

result.pushBack(dataComplex); //O(1)

}

//CUando terminan todas las llamadas recursivas, si era idft divido por N

if(algorithm == "ifft" && result.length() == data.length()){ //O(1)

result = result/result.length();

}

}

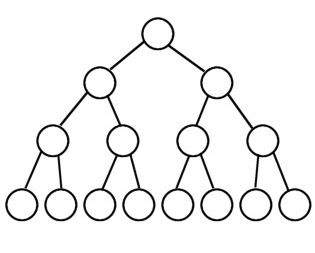
En particular, si definimos N=Tamaño(data) como el tamaño del problema en cuestión, buscamos caracterizar asintóticamente T(N), la función de tiempo de N. Marcamos en el código con O(1) todas las operaciones de complejidad constante (que no depende de N). En particular, vemos que las primeras tres sentencias son operaciones que no dependen del tamaño de la entrada, con lo cual son operaciones O(1).

Observamos que éste algoritmo es de la familia Dividir y Conquistar. La ecuación que caracteriza a T(n) es de la forma:

T(n) = T(n/2) + O(n) (\*3)

Donde el n/2 se deduce dado que se divide la secuencia de entrada en dos subsecuencias de igual número de elementos, y O(n) es la caracterización de la función de complejidad temporal que resulta de hacer la unión de ambas subsecuencias.

En particular, provisto que se asume por la consigna del TP que la entrada data es un vector cuyo cardinal es potencia de 2, esto es N=2M, con M entero, luego el árbol de llamadas será de la forma:



Esto es, un arbol binario completo a M=log2N niveles.

Por el teorema maestro (Teorema de Akra-Bazzi), tenemos en (\*3) las hipótesis del primer caso, y por tanto T(n) es O(n.log n). Además, se puede obtener el mismo resultado observando que (\*3) es la misma ecuación que caracteriza a la complejidad temporal de MergeSort.

Como O(n.log n) < O(n2), la segunda implementación (FFTAlgorithm) es más eficiente en costo temporal que la primera.

# Código fuente

## Archivo complex.h

#ifndef COMPLEX\_H

#define COMPLEX\_H

#include <iostream>

/\*\*

\* Clase representativa de un número complejo. Posee algunas de las operaciones más

\* comunes que se pueden llegar a necesitar para operar con numeros complejos.

\*/

class complex {

//Parte real del número complejo

double re\_;

//Parte imaginaria del número complejo

double im\_;

public:

/\*\*

\* Constructor sin parámetros. Este constructor permite inicializar un

\* complejo en (0,0).

\*/

complex();

/\*\*

\* Constructor. Este constructor permite crear un complejo a partir de

\* un número real. La parte imaginaria queda inicializada en cero.

\*/

complex(double);

/\*\*

\* Constructor. Este constructor permite crear un complejo a partir de

\* dos números reales. El primero corresponde a la parte real y el

\* segundo a la parte imaginaria.

\*/

complex(double,double);

/\*\*

\* Constructor por copia. Construye un complejo a partir de la copiar de otro.

\*/

complex(const complex &);

/\*\*

\* Sobrecarga operador asignación. Asigna parte real a parte real

\* y parte imaginaría a parte imaginaría.

\*/

complex const &operator=(complex const &);

/\*\*

\* Destructor. Destructor para un numero complejo.

\*/

~complex();

/\*\*

\* Parte real de un complejo. Este método devuelve la parte real de

\* un número complejo

\*/

double re() const;

/\*\*

\* Parte imaginaria de un complejo. Este método devuelve la parte

\* imaginaria de un número complejo.

\*/

double im() const;

/\*\*

\* Módulo de un número complejo. Este método devuelve el modulo de

\* un complejo.

\*/

double abs() const;

/\*\*

\* Setea al complejo como un numero al azar con:

\* Re perteneciente al intervalo [from;to]

\* Im perteneciente al intervalo [from;to]

\*/

complex complex::setRandom(double from, double to);

/\*\*

\* Distancia de un complejo a otro.

\*/

double complex::dist(complex const, complex);

/\*\*

\* Fase de un número complejo. Este método devuelve la fase de un número

\* complejo.

\*/

double phase() const;

/\*\*

\* Conversor de complejo de forma polar a forma cartesiana. Este método

\* recibe por parámetro el modulo y la fase de un complejo y retorna

\* un número complejo en su forma cartesiana.

\*/

static complex fromPolarToRectangular(double,double);

/\*\*

\* Sobrecarga operador suma. Este método suma dos números complejos.

\*/

friend complex const operator+(complex const &, complex const &);

/\*\*

\* Sobrecarga operador resta. Este método resta dos números complejos.

\*/

friend complex const operator-(complex const &, complex const &);

/\*\*

\* Sobrecarga operador multiplicación. Este método multiplica dos

\* números complejos.

\*/

friend complex const operator\*(complex const &, complex const &);

/\*\*

\* Sobrecarga operador división. Este método divide un complejo con

\* un número real.

\*/

friend complex const operator/(complex const &, double);

/\*\*

\* Sobrecarga operador potenciación. Este método potencia un número

\* complejo con un numero entero.

\*/

friend complex const operator^(complex const &,int);

/\*\*

\* Sobrecarga operador igual. Este método compara la parte real de un

\* un complejo con un numero real. Además verifica que la parte

\* imaginaria del complejo sea cero.

\*/

friend bool operator==(complex const &, double);

/\*\*

\* Sobrecarga operador igual. Este método compara dos números complejos.

\* Parte real con parte real y parte imaginaria con parte imaginaria.

\*/

friend bool operator==(complex const &, complex const &);

/\*\*

\* Sobrecarga operador escritura. Este operador escribe un complejo

\* en formato (Re,Img) al flujo de salida.

\*/

friend std::ostream &operator<<(std::ostream &, const complex &);

/\*\*

\* Sobrecarga operador lectura. Este operador lee del flujo de entrada

\* un número complejo. Acepta números reales individuales a los cuales

\* se los toma como la parte real del numero complejo que retorna.

\* En el caso de que reciba al numero complejo como par ordenado

\* devuelve un complejo con la parte real y la parte imaginaria

\* obtenida del flujo de entrada.

\*/

friend std::istream &operator>>(std::istream &, complex &);

};

#endif

## Archivo complex.cc

#include "complex.h"

#include <iostream>

#include <cmath>

#include <stdlib.h>

using namespace std;

double fRand(double fMin, double fMax)

{

double f = (double)rand() / RAND\_MAX;

return fMin + f \* (fMax - fMin);

}

complex::complex() : re\_(0), im\_(0) {}

complex::complex(double r) : re\_(r), im\_(0){}

complex::complex(double r, double i) : re\_(r), im\_(i){}

complex::complex(complex const &c) : re\_(c.re\_), im\_(c.im\_){}

complex const & complex::operator=(complex const &c){

re\_ = c.re\_;

im\_ = c.im\_;

return \*this;

}

complex::~complex() {}

double complex::re() const {

return re\_;

}

double complex::im() const {

return im\_;

}

double complex::abs() const {

return std::sqrt(re\_ \* re\_ + im\_ \* im\_);

}

complex complex::setRandom(double from, double to)

{

double re = fRand(from, to);

double im = fRand(from, to);

re\_ = re;

im\_ = im;

return complex(re, im);

}

double complex::dist(complex const x, complex const y)

{

complex r = x-y;

return r.abs();

}

double complex::phase() const {

return atan(this->im\_/this->re\_);

}

complex complex::fromPolarToRectangular(double mod,double phase){

double re = mod\*cos(phase);

double im = mod\*sin(phase);

return complex(re,im);

}

complex const operator+(complex const &x, complex const &y){

complex z(x.re\_ + y.re\_, x.im\_ + y.im\_);

return z;

}

complex const operator-(complex const &x, complex const &y){

complex r(x.re\_ - y.re\_, x.im\_ - y.im\_);

return r;

}

complex const operator\*(complex const &x, complex const &y){

complex r(x.re\_ \* y.re\_ - x.im\_ \* y.im\_,

x.re\_ \* y.im\_ + x.im\_ \* y.re\_);

return r;

}

complex const operator/(complex const &c, double f)

{

return complex(c.re\_ / f, c.im\_ / f);

}

complex const operator^(complex const &c , int power){

if(power == 0){

return complex(1,0);

}

if(power == 1){

return c;

}

double module = c.abs();

double phase = c.phase();

if(power < 0){

module = 1/module;

}

for(int i = 0; i < power - 1 ; i++){

module = module\*module;

}

phase = phase\*power;

return complex::fromPolarToRectangular(module,phase);

}

bool operator== (complex const &c, double f){

bool b = (c.im\_ != 0 || c.re\_ != f) ? false : true;

return b;

}

bool operator== (complex const &x, complex const &y){

bool b = (x.re\_ != y.re\_ || x.im\_ != y.im\_) ? false : true;

return b;

}

ostream & operator<<(ostream &os, const complex &c){

//double reAux = ((int)(c.re\_\*100))/100.0;

//double imAux = ((int)(c.im\_\*100))/100.0;

return os << "(" << c.re\_ << "," << c.im\_ << ")";

}

istream & operator>>(istream &is, complex &c) {

int good = false;

int bad = false;

double re = 0;

double im = 0;

char ch = 0;

if (is >> ch && ch == '(') {

if (is >> re && is >> ch && ch == ',' && is >> im && is >> ch

&& ch == ')')

good = true;

else

bad = true;

} else if (is.good()) {

is.putback(ch);

if (is >> re)

good = true;

else

bad = true;

}

if (good)

c.re\_ = re, c.im\_ = im;

if (bad)

is.clear(ios::badbit);

return is;

}

## Archivo vector.h

#ifndef VECTOR\_H

#define VECTOR\_H

#include <iostream>

using namespace std;

/\*\*

\* Clase vector. Permite almacenar un arreglo de datos. Contiene una serie de

\* de métodos que permiten agregar, quitar y leer los distintos elementos almacenados

\* en el vector. Además la clase esta templetizada, permitiendo así crear vectores

\* de cualquier tipo.

\*/

template <class T>

class vector{

private:

//Tamaño del vector

int size;

//Cantidad de espacio en memoria del vector

int capacity;

//Puntero a la primera posición del vector

T\* pv;

public:

/\*\*

\* Constructor sin parámetros. Inicializa un vector en cero. Además no

\* crea la memoria para el vector.

\*/

vector(){

this->pv = NULL;

this->size = 0;

this->capacity = 0;

}

/\*\*

\* Constructor. Inicializa un vector vacío. A diferencia del constructor

\* sin parámetros, este crea la memoria para el vector de acuerdo al

\* parámetro size\_.

\*/

vector(int size\_){

this->pv = new T[size\_];

this->size = size\_;

this->capacity = size\_;

}

/\*\*

\* Constructor copia. Inicializa un vector a partir de otro pasado

\* por parámetro.

\*/

vector(const vector<T> & cv){

this->size = cv.size ;

this->capacity = cv.capacity;

T\* cp = new T[ size ];

for ( int i = 0; i < this->size; i++ ){

cp[ i ] = cv.pv[ i ];

}

if(this->pv){

delete[] this->pv;

}

this->pv = cp;

}

/\*\*

\* Destructor de vector. borra la memoria creada para el vector.

\*/

~vector(){

if(this->pv != NULL){

delete[] pv;

}

}

/\*\*

\* Largo del vector. Este método devuelve el largo del vector.

\*/

int length() const{

return this->size;

}

/\*\*

\* Inserta un elemento al vector. Este método inserta al final del vector

\* un elemento. Para agregar el elemento primero verifica que tenga

\* memoria, en caso de no tener crea memoria con capacidad igual al

\* doble de la que tenía.

\*/

void pushBack(T & elem){

if(this->size == 0){

this->pv = new T[2];

this->capacity = 2;

}

else{

if(this->capacity == this->size){

T\* aux = this->pv;

this->pv = new T[this->capacity\*2];

this->capacity = this->capacity\*2;

for(int i = 0 ; i < this->size ; i++){

this->pv[i] = aux[i];

}

delete[] aux;

}

}

this->pv[this->size] = elem;

this->size++;

}

/\*

\* Este método remueve del arreglo el último elemento agregado.

\*/

T & popBack(){

this->size--;

return this->pv[this->size - 1];

}

/\*

\* Este método permite obtener una determinada posición del arreglo, si

\* la posición es mayor al largo del arreglo, vuelve a empezar a contar

\* desde la posición inicial.

\*/

T & getCircular(int index){

if(index >= this->size){

return this->pv[index - this->size];

}

else{

return this->pv[index];

}

}

/\*\*

\* Operador asignación. Este operador copia los elementos del vector

\* pasado por parámetro al objeto sobre el cual se ejecuto.

\*/

vector<T> & operator=(const vector<T> & rigth) {

if (&rigth != this)

{

if (this->size != rigth.size) {

T \* aux;

aux = new T[ rigth.size ];

delete [] this->pv;

this->size = rigth.size;

this->pv = aux;

for (int i = 0; i < size; i++){

this->pv[i] = rigth.pv[i];

}

return \*this;

}

else

{

for (int i = 0; i < this->size; i++){

this->pv[i] = rigth.pv[i];

}

return \*this;

}

}

return \*this;

}

/\*\*

\* Operador comparación. Compara el contenido del vector pasado por

\* parámetro con el vector sobre el cual se ejecuto el operador. En el

\* caso que todos los elementos coincidan devuelve true.

\*/

bool operator==(const vector<T> & rigth) const {

if (this->size != rigth.size)

return false; // Vectores de diferentes tamaños

else

for (int i = 0; i < this->size; i++)

if (this->pv[ i ] != rigth.pv[ i ])

return false;

return true;

}

/\*\*

\* Operador indexación constante. Permite obtener el objeto almacenado en una

\* determinada posición del vector.

\*/

const T & operator[](int index) const {

if(index >= this->size){

cerr << "Indice incorrecto en const operator[]" << endl;

abort();

}

return this->pv[index];

}

/\*\*

\* Operador indexación. Permite asignarle un valor a una determinada

\* posición del vector.

\*/

T & operator[](int index) {

if(index >= this->size){

cout << "Indice incorrecto en operator[]" << endl;

abort();

}

return this->pv[index];

}

friend vector<T> & operator/(vector<T> & vector, double divider){

for(int i = 0; i < vector.length();i++){

vector.pv[i] = (vector.pv[i])/divider;

}

return vector;

}

/\*\*

\* Operador lectura. Permite leer de un stream de entrada un conjunto

\* de objetos del tipo T, almacenándolos en un vector.

\*/

friend std::istream &operator>>(std::istream & is,vector<T> & vector){

T aux;

for(int i = 0 ; is >> aux ; i++){

vector.pushBack(aux);

}

return is;

}

/\*\*

\* Operador escritura. Permite escribir en un stream de salida el vector

\* del cual se llamo.

\*/

friend std::ostream &operator<<(std::ostream & os, const vector<T> & vector){

for(int i = 0 ; i < vector.size ; i++){

os << vector[i] << " ";

}

os << endl;

return os;

}

};

#endif /\* VECTOR\_H \*/

## Archivo DFTcalculator.h

/\*

\* File: DFTcalculator.cc

\* Author: Diego / Marcelo

\*

\* Created on March 21, 2016, 8:09 PM

\*/

#include <cstdlib>

#include "vector.h"

#include "complex.h"

#include <fstream>

#include <iostream>

#include <cmath>

#include <string>

#include <stack>

#include <math.h>

#ifndef M\_PI

#define M\_PI 3.14159265358979323846

#endif

/\*\*

\* Clase DFTcalculator. Esta clase contiene una serie de metodos que permiten

\* calcular la transformada discreta de fourier y la transformada inversa

\* de fourier.

\*/

class DFTcalculator

{

private:

/\*\*

\* Método privado estatico calculate. Este método permite calcular tanto la

\* transformada como la anti transformada de fourier. Es un método privado que

\* utiliza la clase.

\*/

static void bruteForceAlgorithm(const vector<complex> & data , vector<complex> & result , string algorithm){

int N = data.length();

int sign = 1;

if(algorithm == "idft"){

sign = -1;

}

for(int i = 0 ; i < N ; i++){

complex sum = 0;

for(int j = 0 ; j < N ; j++){

double re = cos(2\*M\_PI\*j\*i\*sign/N);

double im = sin(2\*M\_PI\*j\*i\*sign/N);

complex W = complex(re,im);

sum = sum + (data[j]\*W);

}

if(algorithm == "idft"){

sum = sum / N;

}

result.pushBack(sum);

}

}

/\*

\* Implementación del algoritmo de la FFT

\*/

static void FFTAlgorithm(const vector<complex> & data , vector<complex> & result , vector<int> & indexes,string algorithm){

int sign = 1;

if(algorithm == "ifft"){

sign = -1;

}

// Si todavía no llegué al caso base llamo recursivamente

if(indexes.length() != 1){

//Creo dos vectores de índices, estos vectores me sirven para no tener que

//crear vectores de data nuevos para las llamadas recursivas

//sino que utilizo el mismo vector data y el vector de índice

//para obtener el dato que necesito en el caso base.

//Vector de índices pares

vector<int> evenIndexes = vector<int>();

//Vector de índices impares

vector<int> oddIndexes = vector<int>();

//Guardo los índices pares e impares en los respectivos vectores

for(int i = 0 ; i < indexes.length() ; i = i + 2){

evenIndexes.pushBack(indexes[i]);

oddIndexes.pushBack(indexes[i+1]);

}

//Creo dos vectores que almacenarán los resultados de las transformadas

//de los índices pares e impares.

vector<complex> evenResult = vector<complex>();

vector<complex> oddResult = vector<complex>();

//Llamo recursivamente para calcular la DFT sobre la sub secuencia

FFTAlgorithm(data,evenResult,evenIndexes,algorithm);

FFTAlgorithm(data,oddResult,oddIndexes,algorithm);

//Uno la DFT de ambas sub secuencias

int N = evenResult.length() + oddResult.length();

for(int i = 0 ; i < N ; i++){

double re = cos(sign\*2\*M\_PI\*i/N);

double im = sin(sign\*2\*M\_PI\*i/N);

complex W(re,im);

complex Xk = W\*oddResult.getCircular(i) + evenResult.getCircular(i);

result.pushBack(Xk);

}

}

else{

//Obtengo el dato que necesito a partir del vector de índices

int dataIndex = indexes[0];

complex dataComplex = data[dataIndex];

result.pushBack(dataComplex);

}

//CUando terminan todas las llamadas recursivas, si era idft divido por N

if(algorithm == "ifft" && result.length() == data.length()){

result = result/result.length();

}

}

/\*

\* Este método permite llenar un arreglo con ceros hasta llegar a la mínima

\* potencia de 2.

\*/

static void fillMinPower2(vector<complex> & data){

//Si el largo no es potencia de 2 completo con ceros

int dataLength = data.length();

double l1 = log((double) dataLength);

double base = 2;

double l2 = log(base);

if(dataLength & (dataLength - 1)){

int nearest = pow(2, ceil( l1 / l2 ) );

for(int i = dataLength ; i < nearest ; i++){

complex complexZero = complex();

data.pushBack(complexZero);

}

}

}

public:

/\*\*

\* Método el cual permite calcular la transformada discreta de fourier. Recibe

\* por parámetro dos vectores uno con la información y otro donde escribirá

\* el resultado.

\*/

static void calculateDFT(const vector<complex>& data , vector<complex>& result)

{

bruteForceAlgorithm(data , result , "dft");

}

/\*\*

\* Método el cual permite calcular la anti-transformada discreta de fourier.

\* Recibe dos parámetros uno con la información y otro donde la escribirá.

\*/

static void calculateiDFT(const vector<complex>& data , vector<complex>& result)

{

bruteForceAlgorithm(data , result , "idft");

}

/\*\*

\* Este método permite calcular la transformada de fourier utilizando el algoritmo

\* de la fft

\*/

static void calculateFFT(vector<complex> & data , vector<complex> & result){

DFTcalculator::fillMinPower2(data);

vector<int> initIndexes = vector<int>();

for(int i = 0 ; i < data.length() ; i++){

initIndexes.pushBack(i);

}

FFTAlgorithm(data,result,initIndexes,"fft");

}

/\*

\* Este método permite calcular la inversa de la transformada de fourier utilizando

\* el algoritmo de la fft.

\*/

static void calculateiFFT(vector<complex> & data , vector<complex> & result){

DFTcalculator::fillMinPower2(data);

vector<int> initIndexes = vector<int>();

for(int i = 0 ; i < data.length() ; i++){

initIndexes.pushBack(i);

}

FFTAlgorithm(data,result,initIndexes,"ifft");

}

/\*

\* Otro método posible para los mismos efectos (otra interfaz distinta)

\*/

static void calculateFT(vector<complex> & data , vector<complex> & result, string method="fft"){

DFTcalculator::fillMinPower2(data);

vector<int> initIndexes = vector<int>();

for(int i = 0 ; i < data.length() ; i++){

initIndexes.pushBack(i);

}

if ( method=="fft" || method=="ifft" )

{

FFTAlgorithm(data,result,initIndexes, method);

}

else if ( method=="dft" || method=="idft" )

{

bruteForceAlgorithm(data , result , method);

}

}

};

## Archivo main.cpp

/\*

\* File: main.cpp

\* Author: Diego / Marcelo

\*

\* Created on March 21, 2016, 8:09 PM

\*/

#include <cstdlib>

#include "vector.h"

#include "complex.cc"

#include <fstream>

#include <iostream>

#include <iomanip>

#include <sstream>

#include <cmath>

#include "cmdline.cc"

#include "DFTcalculator.h"

#include <ctime>

//#include "libs/plog/Log.h"

using namespace std;

#define OPT\_DEFAULT 0

#define OPT\_SEEN 1

#define OPT\_MANDATORY 2

static string method;

static istream \*is = NULL;

static ostream \*os = NULL;

static fstream ifs;

static fstream ofs;

static void opt\_input(string const &arg){

if (arg == "-") {

is = &cin;

}

else {

ifs.open(arg.c\_str(), ios::in);

is = &ifs;

}

if (!is->good()) {

cerr << "cannot open " << arg << "." << endl;

abort();

}

}

static void opt\_output(string const &arg){

if (arg == "-") {

os = &cout;

}

else {

ofs.open(arg.c\_str(), ios::out);

os = &ofs;

}

if (!os->good()) {

cerr << "cannot open " << arg << "." << endl;

abort();

}

}

static void opt\_help(string const &arg){

cout << endl;

cout << "cmdline [-m \"dft\" \"idft\" \"fft\" \"ifft\"] [-i file] [-o file]" << endl;

cout << endl;

cout << "The default input output are the standard IO.";

cout << "The default method is the discrete fast fourier transform (fft).";

cout << endl;

exit(0);

}

static void opt\_method(string const &arg){

method = arg;

}

static option\_t options[] = {

{1, "i", "input", "-", opt\_input, OPT\_DEFAULT},

{1, "o", "output", "-", opt\_output, OPT\_DEFAULT},

{1, "m", "method", "fft", opt\_method, OPT\_DEFAULT},

{0, "h", "help", NULL, opt\_help, OPT\_DEFAULT},

{0, },

};

/\*\*

\* Función main. El programa espera por comando que le configuren el stream de entrada

\* de donde tomará los datos para calcular la transformada discreta de fourier.

\* Además necesita que le configuren el stream de salida a donde escribirá los

\* datos obtenidos.

\*/

int main(int argc, char\*\* argv) {

//Leo las opciones con las que me ejecutan el programa

cmdline cmdl(options);

cmdl.parse(argc, argv);

//Leo la información del stream de entrada linea por linea

string line;

while(getline(\*is,line)){

//Creo un vector que almacenará la información leída

vector<complex> data = vector<complex>();

//Creo un vector que almacenará el resultado

vector<complex> result = vector<complex>();

istringstream iss(line);

iss >> data;

\*os << std::setprecision(2);

\*os << std::fixed;

//Dependiendo que ingreso el usuario hago una cosa u otra

if(method == "dft"){

//Calculo la DFT

DFTcalculator::calculateDFT(data,result);

}

else if(method == "idft"){

//Calculo la DFT

DFTcalculator::calculateiDFT(data,result);

}

else if(method == "fft"){

//Calculo la FFT

DFTcalculator::calculateFFT(data,result);

}

else if(method == "ifft"){

//Calculo la iFFT

DFTcalculator::calculateiFFT(data,result);

}

//Imprimo el resultado

\*os << result;

std::ostringstream stream;

stream << result;

std::string str = stream.str();

const char\* chr = str.c\_str();

cout << "Ouput:" << str;

//imprimo data y result.

cout << "Longitud vector data: " << data.length() << endl;

cout << "Longitud vector result: " << result.length() << endl;

cout << "data: " << data << endl;

cout << "result: " << result << endl;

}

return 0;

}

## Archivo UnitTests.cpp

/\*

\* File: UnitTests.cpp

\* Author: Diego / Marcelo

\*

\*/

#include <cstdlib>

#include "vector.h"

#include "complex.cc"

#include <fstream>

#include <iostream>

#include <iomanip>

#include <sstream>

#include <cmath>

#include "cmdline.cc"

#include "DFTcalculator.h"

#include <ctime>

#include <assert.h>

#include <algorithm>

#include <functional>

#include <cctype>

#include <locale>

using namespace std;

#define OPT\_DEFAULT 0

#define OPT\_SEEN 1

#define OPT\_MANDATORY 2

#define EPS 0.001

#define NMAX 10

static string method;

static istream \*is = NULL;

static ostream \*os = NULL;

static fstream ifs;

static fstream ofs;

// funci�n auxiliar ltrim

static inline void ltrim(std::string &s) {

s.erase(s.begin(), std::find\_if(s.begin(), s.end(), std::not1(std::ptr\_fun<int, int>(std::isspace))));

}

// funci�n auxiliar rtrim

static inline void rtrim(std::string &s) {

s.erase(std::find\_if(s.rbegin(), s.rend(), std::not1(std::ptr\_fun<int, int>(std::isspace))).base(), s.end());

}

// funci�n auxiliar trim

static inline void trim(std::string &s) {

ltrim(s);

rtrim(s);

}

static inline std::string ltrimmed(std::string s) {

ltrim(s);

return s;

}

static inline std::string rtrimmed(std::string s) {

rtrim(s);

return s;

}

static inline std::string trimmed(std::string s) {

trim(s);

return s;

}

class UnitTests

{

private:

double L2Distance(vector<complex> v1, vector<complex> v2)

{

double s = 0;

for (int i=0; i < v1.length(); i++)

{

complex a = (v1[i] - v2[i])\*(v1[i] - v2[i]);

s = s + a.abs();

}

return sqrt(s);

}

public:

bool TestFTOfZeroesIsZeroForAllTransforms()

{

for (int n=0; n<NMAX; n++)

{

int m = pow((long double) 2, (int) n);

cout << m;

TestFTOfZeroesIsZero("fft", m);

TestFTOfZeroesIsZero("dft", m);

TestFTOfZeroesIsZero("ifft", m);

TestFTOfZeroesIsZero("idft", m);

}

return true;

}

bool TestFTOfZeroesIsZero(string method="fft", int n=2)

{

//para un string de entrada de la forma: 0 0 0 0, el resultado de aplicar la TF (bien sea FFT, iFFT, DFT, iDFT)

//debe ser un string de la forma (0.00,0.00) (0.00,0.00) (0.00,0.00) (0.00,0.00)

string input = "";

for (int i=0; i<n; i++)

{

input = input + "0 ";

}

trim(input);

string expectedoutput = "";

for (int i=0; i<n; i++)

{

expectedoutput = expectedoutput + "(0.00,0.00) ";

}

trim(expectedoutput);

string output;

Calculate(input, output, method);

trim(output);

//imprimo data y result.

cout << "input: " << input << endl;

cout << "output: " << output << endl;

assert (output==expectedoutput);

return true;

}

bool TestFTOf11(string method="fft")

{

string input ="1 1";

string expectedoutput = "(2.00,0.00) (0.00,0.00)";

string output;

Calculate(input, output, method);

trim(output);

//imprimo data y result.

cout << "input: " << input << endl;

cout << "output: " << output << endl;

assert (output==expectedoutput);

return true;

}

bool TestFTOfii(string method="fft")

{

string input ="(0,1) (0,1)";

string expectedoutput = "(0.00,2.00) (-0.00,0.00)";

string output;

Calculate(input, output, method);

trim(output);

//imprimo data y result.

cout << "input: " << input << endl;

cout << "output: " << output << endl;

assert (output==expectedoutput);

return true;

}

bool TestIFTOf20(string method="ifft")

{

string input ="2 0";

string expectedoutput = "(1.00,0.00) (1.00,0.00)";

string output;

Calculate(input, output, method);

trim(output);

//imprimo data y result.

cout << "input: " << input << endl;

cout << "output: " << output << endl;

assert (output==expectedoutput);

return true;

}

bool TestIFTOf2i0(string method="ifft")

{

string input ="(0.00,2.00) (0.00,0.00)";

string expectedoutput = "(0.00,1.00) (0.00,1.00)";

string output;

Calculate(input, output, method);

trim(output);

//imprimo data y result.

cout << "input: " << input << endl;

cout << "output: " << output << endl;

assert (output==expectedoutput);

return true;

}

bool TestFTAditivity(string method="fft", int n=4)

{

//al ser FFT, DFT, iDFT, iFFT operadores lineales, todos ellos verifican F(a+b)=F(a)+F(b) (aditividad)

vector<complex> input1 = vector<complex>();

vector<complex> input2 = vector<complex>();

vector<complex> inputsum = vector<complex>();

vector<complex> output1 = vector<complex>();

vector<complex> output2 = vector<complex>();

vector<complex> outputsum = vector<complex>();

//lleno los dos vectores input1, input2 con numeros complejos al azar dentro del intervalo [1;10]

for (int i=0; i<n; i++)

{

complex\* a = new complex();

complex\* b = new complex();

complex\* sum = new complex();

a->setRandom(1,10);

b->setRandom(1,10);

\*sum = \*a + \*b;

input1.pushBack(\*a);

input2.pushBack(\*b);

inputsum.pushBack(\*sum);

}

DFTcalculator::calculateFT(input1, output1, method);

DFTcalculator::calculateFT(input2, output2, method);

DFTcalculator::calculateFT(inputsum, outputsum, method);

cout << "input1: " << input1 << endl;

cout << "input2: " << input2 << endl;

cout << "inputsum: " << inputsum << endl;

cout << "output1: " << output1 << endl;

cout << "output2: " << output2 << endl;

cout << "outputsum: " << outputsum << endl;

//verificamos ahora que F(a+b) = F(a) + F(b), m�s all� de un error de truncamiento/representaci�n acotado.

for (int i=0; i<n; i++)

{

complex\* d = new complex();

\*d = (output1[i] + output2[i]) - outputsum[i];

assert (d->abs() < EPS);

}

return true;

}

bool TestFTHomogeneity(string method="fft", int n=4)

{

//al ser FFT, DFT, iDFT, iFFT operadores lineales, todos ellos verifican F(ka)=kF(a) (homogeneidad)

vector<complex> input1 = vector<complex>();

vector<complex> input2 = vector<complex>();

vector<complex> inputscalarmult = vector<complex>();

vector<complex> output1 = vector<complex>();

vector<complex> output2 = vector<complex>();

vector<complex> outputscalarmult = vector<complex>();

//lleno los dos vectores input1, input2 con numeros complejos al azar dentro del intervalo [1;10]

complex\* k = new complex();

k->setRandom(1,10);

for (int i=0; i<n; i++)

{

complex\* a = new complex();

complex\* mult = new complex();

a->setRandom(1,10);

\*mult = \*k \* \*a;

input1.pushBack(\*a);

inputscalarmult.pushBack(\*mult);

}

DFTcalculator::calculateFT(input1, output1, method);

DFTcalculator::calculateFT(inputscalarmult, outputscalarmult, method);

cout << "input1: " << input1 << endl;

cout << "input2: " << input2 << endl;

cout << "inputsclaramult: " << inputscalarmult << endl;

cout << "output1: " << output1 << endl;

cout << "output2: " << output2 << endl;

cout << "outputscalarmult: " << outputscalarmult << endl;

//verificamos ahora que F(ka) = kF(a), m�s all� de un error de truncamiento/representaci�n acotado.

for (int i=0; i<n; i++)

{

complex\* d = new complex();

\*d = (\*k \* output1[i]) - outputscalarmult[i];

assert (d->abs() < EPS);

}

return true;

}

bool TestFTOfSingleton(string method="fft")

{

//la transformada de un vector de un solo �ndice (v.g. "(5)") tiene que ser el mismo n�mero.

vector<complex> input1 = vector<complex>();

vector<complex> output1 = vector<complex>();

complex\* a = new complex();

a->setRandom(1,10);

input1.pushBack(\*a);

DFTcalculator::calculateFT(input1, output1, method);

cout << "input1: " << input1 << endl;

cout << "output1: " << output1 << endl;

//verificamos ahora que a = F(a), m�s all� de un error de truncamiento/representaci�n acotado.

complex\* d = new complex();

\*d = input1[0] - output1[0];

assert (d->abs() < EPS);

return true;

}

bool TestFastEqualsDiscrete(string method="fft", int n=4)

{

string methodfast;

string methoddiscrete;

if (method=="fft")

{

methodfast="fft";

methoddiscrete= "dft";

}

else if (method=="ifft")

{

methodfast="ifft";

methoddiscrete= "idft";

}

else if (method=="dft")

{

methodfast="fft";

methoddiscrete= "dft";

}

else if (method=="idft")

{

methodfast="ifft";

methoddiscrete= "idft";

}

//testeamos -tanto para la transformada, como para la transformada inversa- que el m�todo via la definici�n ("discrete")

//sea igual al resultado del m�todo mediante el algoritmo r�pido ("fast").

vector<complex> input1 = vector<complex>();

vector<complex> outputfast = vector<complex>();

vector<complex> outputdiscrete = vector<complex>();

//lleno los dos vectores input1, input2 con numeros complejos al azar dentro del intervalo [1;10]

for (int i=0; i<n; i++)

{

complex\* a = new complex();

a->setRandom(1,10);

input1.pushBack(\*a);

}

DFTcalculator::calculateFT(input1, outputfast, methodfast);

DFTcalculator::calculateFT(input1, outputdiscrete, methoddiscrete);

cout << "outputfast:" << outputfast << endl;

cout << "outputdiscrete:" << outputdiscrete << endl;

cout << endl;

//verificamos ahora que outputfast=outputdiscrete, m�s all� de un error de truncamiento/representaci�n acotado.

for (int i=0; i<n; i++)

{

complex\* d = new complex();

\*d = (outputfast[i]) - outputdiscrete[i];

//assert (d->abs() < EPS);

}

return true;

}

bool TestFTofIFT(string method="fft", int n=4)

{

//testeamos -tanto para la transformada como para la antitransformada, tanto en version discreta como r�pida- que valgan

//las siguientes identidades funcionales: F(F-1(a))=F-1(F(a))=a

string func;

string inversefunc;

if (method=="fft")

{

func="fft";

inversefunc = "ifft";

}

else if (method=="ifft")

{

func="ifft";

inversefunc = "fft";

}

else if (method=="dft")

{

func="dft";

inversefunc = "idft";

}

else if (method=="idft")

{

func="idft";

inversefunc = "dft";

}

// F( F'1(a) ) = a (inversa a derecha)

vector<complex> a = vector<complex>();

vector<complex> b = vector<complex>();

vector<complex> c = vector<complex>();

//lleno el vector input1 con numeros al azar dentro del intervalo [1;10]

for (int i=0; i<n; i++)

{

complex\* ca = new complex();

ca->setRandom(1,10);

a.pushBack(\*ca);

}

DFTcalculator::calculateFT(a, b, func);

DFTcalculator::calculateFT(b, c, inversefunc);

cout << "a: " << a << endl;

cout << "b = F(a): " << b << endl;

cout << "c = F-1(F(a)): " << c << endl;

for (int i=0; i<n; i++)

{

complex\* d = new complex();

\*d = a[i] - c[i];

cout << \*d << " ";

assert (d->abs() < EPS);

}

return true;

}

bool TestParseval(string method="fft", int n=4)

{

//si es a tal que ||a||2=L, luego vale: ||F(a)||2=nL (igualdad de Parseval), donde n es la longitud de a

//esto es, la transformada de Fourier preserva la norma2.

double multiplier=n;

if (method=="fft" || method=="dft")

{

multiplier = n;

}

else if (method=="ifft" || method=="idft")

{

multiplier = 1/((double) n);

}

vector<complex> input = vector<complex>();

vector<complex> output = vector<complex>();

//lleno el vector input con numeros al azar dentro del intervalo [1;10]

for (int i=0; i<n; i++)

{

complex\* z = new complex();

z->setRandom(1,10);

input.pushBack(\*z);

}

//calculamos la norma2 de input:

double sum=0;

for (int i=0; i<n; i++)

{

complex\* z2 = new complex();

\*z2 = input[i] \* input[i];

sum = sum + z2->abs();

}

double inputnorm = sum;

//transformamos...

DFTcalculator::calculateFT(input, output, method);

//calculamos la norma2 de output:

sum=0;

for (int i=0; i<n; i++)

{

complex\* z2 = new complex();

\*z2 = output[i] \* output[i];

sum = sum + z2->abs();

}

double outputnorm = sum;

cout << "input norm: " << inputnorm << endl;

cout << "output norm/" << multiplier << ": " << outputnorm/multiplier << endl;

assert( abs(inputnorm - outputnorm/multiplier) < EPS );

return true;

}

void Calculate(string input, string& output, string method="fft")

{

string line = input;

os = &cout;

//Creo un vector que almacenar� la informaci�n le�da

vector<complex> data = vector<complex>();

//Creo un vector que almacenar� el resultado

vector<complex> result = vector<complex>();

istringstream iss(line);

iss >> data;

\*os << std::setprecision(2);

\*os << std::fixed;

//Dependiendo que ingreso el usuario hago una cosa u otra

if(method == "dft"){

//Calculo la DFT

DFTcalculator::calculateDFT(data,result);

}

else if(method == "idft"){

//Calculo la DFT

DFTcalculator::calculateiDFT(data,result);

}

else if(method == "fft"){

//Calculo la FFT

DFTcalculator::calculateFFT(data,result);

}

else if(method == "ifft"){

//Calculo la iFFT

DFTcalculator::calculateiFFT(data,result);

}

//Imprimo el resultado

std::ostringstream stream;

stream << setprecision(2);

stream << fixed;

stream << result;

std::string str = stream.str();

const char\* chr = str.c\_str();

//cout << "Ouput:" << chr;

output = chr;

//imprimo data y result.

cout << "Longitud vector data: " << data.length() << endl;

cout << "Longitud vector result: " << result.length() << endl;

cout << "data: " << data << endl;

cout << "result: " << result << endl;

}

};

int main(int argc, char\*\* argv)

{

//chequeamos a continuaci�n con esta clase de tests que se cumplan una serie de propiedades fundamentales

//que caracterizan a la DFT y a la iDFT (tanto en sus versiones discretas, como en sus versiones r�pidas: FFT, iFFT)

//Para cubrir m�s casos con menos c�digo, parametrizamos las pruebas

UnitTests\* tests = new UnitTests();

tests->TestFTOfZeroesIsZero();

//tests->TestFTOfZeroesIsZeroForAllTransforms();

tests->TestFTOf11("fft");

tests->TestFTOf11("dft");

tests->TestFTOfii("fft");

tests->TestFTOfii("dft");

tests->TestIFTOf20("ifft");

tests->TestIFTOf20("idft");

tests->TestIFTOf2i0("ifft");

tests->TestIFTOf2i0("idft");

string fts[] = {"fft", "ifft", "dft", "idft"};

for (int i=0; i<4; i++)

{

tests->TestFTOfSingleton(fts[i]);

for (int j=0; j<NMAX; j++)

{

int m = pow((long double) 2, (int) j);

tests->TestFTAditivity(fts[i], m);

tests->TestFTHomogeneity(fts[i], m);

tests->TestFastEqualsDiscrete(fts[i], m);

tests->TestFTofIFT(fts[i], m);

tests->TestParseval(fts[i], m);

tests->TestParseval(fts[i], m);

}

}

return 0;

}

## Corridas de ejecución del programa

# Si bien la clase de prueba UnitTests constituye una amplia fuente de corridas de ejemplo del algoritmo FFTAlgorithm y bruteForceAlgorithm, damos a continuación algunas corridas de ejemplo del programa, para verificar su correcto funcionamiento.

# main -i input\_1.csv -o ouput\_1.csv con input\_1.csv: 0 0 0 0 Luego, output\_1.csv contendrá: 0 0 0 0 (OK)

# main -i input\_2.csv -o ouput\_2.csv con input\_2.csv: 0 0 0 0 0 0 0 0 0 0 Luego, output\_2.csv contendrá: (0.00,0.00) (0.00,0.00) (0.00,0.00) (0.00,0.00) (0.00,0.00) (0.00,0.00) (0.00,0.00) (0.00,0.00) (0.00,0.00) (0.00,0.00) (OK)

# main -i input\_3.csv -o ouput\_1.csv con input\_4.csv: 1 1 Luego, output\_3.csv contendrá: (2.00,0.00) (0.00,0.00) (OK, lo cual se puede verificar por definición)

# main -i input\_4.csv -o ouput\_4.csv -m ifft con input\_4.csv: (2.00,0.00) (0.00,0.00) Luego, output\_3.csv contendrá: (1.00,0.00) (1.00,0.00) (OK, de acuerdo con la definición de inversa)

**Bibliografía**

[1] Oppenheim, Schaffer, Discrete-time signal processing, second edition, Ed. Prentice Hall.

[2] Walkthrough: Compiling a Native C++ Program on the Command Line (C++), Microsoft Developer Network, disponible desde https://msdn.microsoft.com/en-us/library/ms235639(v=vs.100).aspx

[3] Gcc Tutorial, B. Chung, accesible desde http://pages.cs.wisc.edu/~beechung/ref/gcc-intro.html

[4] Apuntes de Cátedra - curso Algoritmos y Programación II - Calvo - Santi - Santi, Axiomas de la Clase Vector

[5] Wikipedia, Big O Notation, disponible en https://en.wikipedia.org/wiki/Big\_O\_notation

[6] Introduction to Analysis of Algorithms, Asymptotic Notation, D. Helmbold, disponible desde https://classes.soe.ucsc.edu/cmps102/Spring04/TantaloAsymp.pdf

[7] Wikipedia, Akra Bassi Method, disponible en https://en.wikipedia.org/wiki/Akra%E2%80%93Bazzi\_method

1. cfr. Consigna TP1 - Algoritmos y Programación II - Recursividad [↑](#footnote-ref-2)
2. En efecto, la complejidad amortizada de la operación pushBack para el contenedor vector es O(1), lo cual fue demostrado en clase. [↑](#footnote-ref-3)