TP0 de Algoritmos y Programación:  
Programación C++

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++, implementando un programa que permita computar la transformada discreta de Fourier (DFT), y la transformada discreta de Fourier inversa (iDFT), para un conjunto de señales discretas complejas de variable entera, y su inversa correspondiente.

**Palabras clave:** Programación C++, Constructores, Constructor de copia, Constructor sin argumentos, Destructores, Clase Vector, Sobrecarga de operadores, StdIn, StdOut; Señal, DFT, iDFT, Transformada de Fourier

# Introducción

Se pide dar implementaciones de la DFT (Discrete Fourier Transform) y la iDFT (Inverse Discrete Fourier Transform) [1]

**DFT**Para el caso de la DFT, se pide calcular la transformación (DFT) de una secuencia de N puntos, x0, ..., xN-1, 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**

Para el caso de la iDFT, se pide calcular la transformación -inversa a la anterior- (iDFT) de una secuencia de N puntos, x0, ..., xN-1, 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-1[ F[ (xk) ] ] = (xk) [1], la cual se conoce como "Definición de la inversa".

Se verifican además las siguientes propiedades

F[ F-1[ (xk) ] ] = 2π(xk) [1] (segunda propiedad de la inversa)

F[ F [ (xk) ] ] = (x-k) [1] (principio de dualidad)

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

El objetivo de este TP es proveer una implementación y una interfaz de uso para estas dos operaciones, programadas en C++.

# 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* 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

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

Claramente, son posibles otras particiones en responsabilidades distintas, más finas, o más gruesas, según sea el enfoque del problema.

# 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:

g++ -g main.cpp -o main

lo cual producirá el archivo ejecutable main [3].

# Estructura de la solución y análisis de la complejidad

Consideramos la siguiente estructura de la solución:

**Complejo.cc (clase)**

+ complejo::complejo()

*(método, constructor sin argumentos)*

Construye un número complejo por default (a saber el 0+i0). Provisto que no hay estructuras de iteración en su implementación, y que todas las sentencias dentro del cuerpo son sentencias de complejidad temporal constante (es decir, sentencias O(k) ), luego, su complejidad es también constante, es decir O(k).

+ complejo::complejo(double r)

*(método, constructor con un sólo argumento, de tipo double)*

Construye un número complejo (real) a partir de un sólo argumento, de tipo double, suministrado. Provisto que no hay estructuras de iteración en su implementación, y que todas las sentencias dentro del cuerpo son sentencias de complejidad temporal constante (es decir, sentencias O(k) ), luego, su complejidad es también constante, es decir O(k).

+ complejo::complejo(double r, double i)

*(método, constructor con dos argumentos, de tipo double)*

Construye un número complejo (real) a partir de dos argumentos de tipo double, a saber su parte real y su parte imaginaria. Nuevamente, vale la consideración anterior, con lo cual su complejidad es de tipo constante.

+ complejo const &complejo::operator=(complejo const &c)

*(método, operador de asignación)*

Asigna al objeto this como lvalue, el valor del número complejo pasado por argumento. Nuevamente, vale la consideración anterior, es decir, este método es O(k).

+ complejo const &complejo::operator\*=(complejo const &c)

*(método, operador =)*

Asigna al objeto this como lvalue, el valor que resulta de multiplicar el valor del lvalue por el número pasado por argumento. Nuevamente, vale la consideración anterior, es decir, este método es O(k).

+ bool complejo::zero() const

*(método)*

Devuelve si el número complejo es el número cero. Nuevamente, vale la consideración anterior, es decir, este método es O(k).

+ complejo const & complejo::operator=(complejo const &c)

*(método)*

Nuevamente, vale la consideración anterior, es decir, este método es O(k).

+ complejo const & complejo::operator\*=(complejo const &c)

*(método)*

Nuevamente, vale la consideración anterior, es decir, este método es O(k).

+ complejo const & complejo::operator+=(complejo const &c)

*(método)*

Nuevamente, vale la consideración anterior, es decir, este método es O(k).

+ complejo const & complejo::operator-=(complejo const &c)

*(método)*

Nuevamente, vale la consideración anterior, es decir, este método es O(k).

+ complejo::~complejo()

*(método destructor)*

Nuevamente, vale la consideración anterior, es decir, este método es O(k).

+ complejo::re() const

*(método getter)*

Devuelve la parte real del número complejo. Nuevamente, vale la consideración anterior, es decir, este método es O(k).

+ double complejo::im() const

*(método getter)*

Devuelve la parte imaginaria del número complejo. Nuevamente, vale la consideración anterior, es decir, este método es O(k).

+ double complejo::abs() const

*(método getter)*

Devuelve el valor absoluto del número complejo. Nuevamente, vale la consideración anterior, es decir, este método es O(k).

+ double complejo::abs2() const

*(método getter)*

Devuelve el valor absoluto del número complejo. Nuevamente, vale la consideración anterior, es decir, este método es O(k).

+ double complejo::phase() const

*(método getter)*

Devuelve el argumento del número complejo. Nuevamente, vale la consideración anterior, es decir, este método es O(k).

+ complejo const & complejo::conjugar()

*(método getter)*

Conjuga el número complejo. Nuevamente, vale la consideración anterior, es decir, este método es O(k).

+ complejo const complejo::conjugado() const

*(método getter)*

Devuelve el número complejo conjugado. Nuevamente, vale la consideración anterior, es decir, este método es O(k).

+ complejo const operator+(complejo const &x, complejo const &y)

*(método)*

Devuelve el número complejo suma del número complejo en el objeto this, más el número complejo pasado por argumento. Nuevamente, vale la consideración anterior, es decir, este método es O(k).

+ complejo const operator-(complejo const &x, complejo const &y)

*(método)*

Devuelve el número complejo resta del número complejo en el objeto this, con el número complejo pasado por argumento. Nuevamente, vale la consideración anterior, es decir, este método es O(k).

+ complejo const operator\*(complejo const &x, complejo const &y)

*(método)*

Devuelve el número complejo producto del número complejo en el objeto this, con el número complejo pasado por argumento. Nuevamente, vale la consideración anterior, es decir, este método es O(k).

+ complejo const operator/(complejo const &x, complejo const &y)

*(método)*

Devuelve el número complejo cociente del número complejo en el objeto this, con el número complejo pasado por argumento. Nuevamente, vale la consideración anterior, es decir, este método es O(k).

+ complejo const operator/(complejo const &c, double f)

*(método)*

Devuelve el número complejo cociente del número complejo en el objeto this, con el número complejo pasado por argumento. Nuevamente, vale la consideración anterior, es decir, este método es O(k).

+ complejo const operator^(complejo const &c , int power)

*(método)*

Devuelve el número complejo en el objeto actual (this) elevado a la potencia entera *power*. A diferencia de los métodos anteriores, este método implementa una única estructura de iteración lineal (*for*), y si consideramos n=tamaño del problema=power, luego podemos decir que el algoritmo es O(n), provisto que todas las operaciones dentro de este *for* son de complejidad constante. Vale aclarar también que en el caso de que el exponente sea 0, se devuelve 1 (z0=1 para todo complejo z no nulo), y en el caso que el exponente sea 1, se devuelve z. Para estos dos casos en particular, la complejidad es constante (es decir, O(k) ). De todas maneras, cuando se habla de la complejidad de un algoritmo siempre se supone el peor caso , con lo cual la complejidad global de esta operación es O(n).

+ bool operator==(complejo const &c, double f)

+ bool operator==(complejo const &x, complejo const &y)

+ ostream & operator<<(ostream &os, const complejo &c)

+ istream & operator>>(istream &is, complejo &c)

**Vector.h (clase template)**

Suele ser útil recordar que como Vector es una clase template, debe estar definida enteramente en un .h, dado que es una "clase de especificación incompleta", o colección de clases de especificación incompleta, cuya especificación se realizará al momento de compilación.

+ vector::vector()

*(método, constructor sin argumentos)*

Construye un vector vacío, de tamaño 0. Provisto que no hay estructuras de iteración en su implementación, y que todas las sentencias dentro del cuerpo son sentencias de complejidad temporal constante (es decir, sentencias O(k) ), luego, su complejidad temporal es O(k).

+ vector::vector()

*(método, constructor sin argumentos)*

Construye un vector vacío, de tamaño 0. Provisto que no hay estructuras de iteración en su implementación, y que todas las sentencias dentro del cuerpo son sentencias de complejidad temporal constante (es decir, sentencias O(k) ), luego, su complejidad temporal es O(k).

+ vector(int size\_)

*(método, constructor con argumento entero)*

Construye un vector vacío, de tamaño size\_. Nuevamente, la complejidad es O(k), suponiendo que la operación de reserva de memoria del array interior de la implementación sea una operación de complejidad constante.

+ vector(const vector<T> & cv)

*(método, constructor de copia)*

Construye un vector a partir de otro. Dado que hay un único for dentro de este método, y siendo n=size(cv) el tamaño del problema, tenemos que la complejidad de este método es O(n).

~vector()

*(destructor)*

Nuevamente, suponiendo que la implementación de delete[] sea O(k), tendremos que la complejidad de este método es O(k).

+ void pushFron(T & elem)

*(método)*

Pushea un elemento al frente del vector. Sea n=size(this) el tamaño del vector, luego, al haber una única estructura de iteración que barre justamente sobre esta variable, tenemos que la complejidad es O(n).

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

*(método)*

Operador de asignación. Sea n=size(this) el tamaño del vector, luego, al haber una única estructura de iteración que barre justamente sobre esta variable, tenemos que la complejidad es O(n).

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

*(método)*

Operador de comparación. Sea n=size(this) el tamaño del vector, luego, al haber una única estructura de iteración que barre justamente sobre esta variable, tenemos que la complejidad es O(n).

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

*(método)*

Operador de acceso por índice. Suponiendo que la estructura array (nativa de C++), sobre la cual esta construída esta implementación de la clase Vector tenga acceso a sus elementos en complejidad constante, luego la complejidad de este método es también constante, es decir O(k).

**DFTcalculator (clase)**

- void calculate(const vector<complejo> & data , vector<complejo> & result , string algorithm)

*(método privado)*

Calcula la transformada (antitransformada) discreta de Fourier del vector de complejos pasado por parámetro data, y lo deposita en result. Sea N el tamaño del vector data pasado por referencia. Al haber dos fors anidados, cada uno de los cuales itera sobre el total de los N elementos, y una operación de potencia dentro de los for, la cual tiene complejidad O(n), donde n es la potencia suministrada, luego tenemos que la complejidad de este método es O(N2n). Teniendo en cuenta además que en el peor de los casos n=N, tenemos entonces que este método es O(N3).

+ void calculateDFT(const vector<complejo> & data , vector<complejo> & result)

*(método)*

(Se agrega por simplicidad, solamente interfaz de acceso al método de arriba). Vale la misma consideración sobre su complejidad.

+ void calculateDFT(const vector<complejo> & data , vector<complejo> & result)

(Ibídem)

**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