75.04/95.12 Algoritmos y Programación II

Trabajo Práctico 1: objetos y algoritmos

Universidad de Buenos Aires - FIUBA

Primer Cuatrimestre de 2018

1. Objetivos

Implementar un algoritmo para la transformada rápida de Fourier (FFT, por sus siglas en inglés), utilizando un patrón de diseño especificado.

2. Alcance

Este trabajo práctico es de elaboración grupal, evaluación individual, y de carácter obligatorio para todos alumnos del curso.

3. Requisitos

El trabajo deberá ser entregado personalmente, en la fecha estipulada, con una carátula que contenga los datos completos de todos los integrantes, un informe impreso de acuerdo con lo que mencionaremos en la sección 5, y con una copia digital de los archivos fuente necesarios para compilar el trabajo.

4. Descripción

En este trabajo extenderemos y mejoraremos nuestras implementaciones previas de la transformada de Fourier. En particular, nos interesa entender e implementar la transformada rápida de Fourier (sección 4.1), que ofrece mejoras teóricas de performance en relación al algoritmo que fue implementado en el trabajo anterior.

Además de esto, y a modo de integrar los conceptos estudiados, se requerirá también que los programas incorporen los algoritmos previamente implementados. Esta incorporación deberá permitir que el usuario pueda elegir al momento de invocar el programa qué versión decide utilizar para computar las transformadas. Para ello, extenderemos el conjunto de valores de la opción `-m` de la línea de comando (sección 4.4).

En cuanto a la implementación propiamente dicha, nos interesa llevar a cabo esta tarea mediante técnicas de diseño orientado a objetos: definiremos un objeto por cada uno de nuestros algoritmos y aplicaremos el patrón de diseño *Strategy* (sección 4.2).

4.1. La transformada rápida de Fourier (FFT)

A diferencia de la DFT, la FFT permite computar eficientemente la transformada de Fourier a una secuencia de N puntos. Vamos a considerar que la cantidad de puntos es una potencia entera de 2, por lo que toda entrada que no tenga esta cantidad de puntos incluirá ceros al final del vector hasta llegar a la potencia entera de 2 más cercana.

A este algoritmo se llega mediante una estrategia de dividir y conquistar. Dada la propiedad de que $W_N^{kn} = e^{-j(kn2\pi/N)}$ es periódica en k y n, se puede separar en:

$$X[k] = G[k] + W_N^{-kn} H[k] \qquad k = 0, ..., N-1; \qquad W_N = e^{-j(2\pi/N)} \tag{1}$$

Donde $G(k)$ y $H(k)$ son las DFTs de $N/2$ puntos de las secuencias $\{x_0; x_2; ...; x_{N-2}\}$ y $\{x_1; x_3; ...; x_{N-1}\}$, es decir, los puntos con subíndices pares e impares respectivamente, donde se puede acceder a los vectores G y H de forma modular: $G(N/2) = G(0)$, $G(N/2+1) = G(1)$ y así sucesivamente.

De forma análoga, también se puede hacer la antitransformada (IFFT) con el mismo razonamiento explicando más arriba e invirtiendo los W_N y agregando el factor multiplicativo $\frac{1}{N}$.

4.2. El patrón de diseño *Strategy*

Un *patrón de diseño* es una solución reutilizable para un problema que ocurre comúnmente en el proceso de diseño de software. Para el caso de *Strategy*, éste se trata de un patrón que define una familia de algoritmos, cada uno representado con un objeto, y los hace intercambiables en tiempo de ejecución. *Strategy* permite que el algoritmo varíe en forma independiente del cliente que requiera su utilización [1].

Esto se logra definiendo una interfaz básica y homogénea para los algoritmos involucrados. Siguiendo esta interfaz, los clientes podrán interactuar de igual forma con cada uno de los algoritmos, sin tener que hacer distinciones innecesarias.

En C++, definiremos esta interfaz como una clase *abstracta* (esto es, una clase que no poseerá instancias). Allí se deberá definir el prototipo de cada mensaje que nuestros algoritmos deben saber responder. La implementación concreta de cada uno de ellos se hará en la definición de la clase de cada algoritmo. Por supuesto, cada una de ellas será subclase de la interfaz mencionada.

Los detalles restantes para la correcta implementación de este patrón quedará a criterio de cada grupo.

4.3. Formato de entrada y salida

El archivo de entrada será del mismo tipo que en los programas anteriores: un archivo de texto con pares ordenados de complejos `(re, im)`, separados por espacios, con una señal por línea. La salida tendrá el mismo formato, siendo cada línea la transformada o antitransformada de la señal correspondiente, según el orden de entrada (la primera señal en la salida es la transformada de la primera señal de la entrada, lo mismo para la segunda y así sucesivamente).

4.4. Interfaz

La interacción con el programa seguirá con las opciones por línea de comando del trabajo anterior, aunque extendiendo ahora el conjunto de valores posibles para la opción `-m`. Todas las opciones involucradas son las siguientes:

- `-i`, o `--input`, donde se pasará el nombre del archivo con la señal de entrada de acuerdo al formato explicado en la sección 4.3. En caso de ser "-" o no estar especificado se deberá leer desde la entrada estándar.
- `-o`, o `--output`, donde se pasará el nombre del archivo donde deben ser escritas las señales de salida. Otra vez, en caso de ser "-" o no estar especificado se escribirá sobre la salida estándar.
- `-m`, o `--method`, donde se especificará qué transformada se hará a la señal, siendo las posibilidades DFT, IDFT, FFT o IFFT. En caso de no ser especificado, se tomará por defecto la FFT.

4.5. Ejemplos

Se incluye para el trabajo práctico un conjunto de archivos de texto con señales y sus transformadas en el formato requerido para éste.

El ejemplo más simple consiste en una entrada vacía. Observar que la salida es también vacía:

```
$ touch entrada1.txt
$ ./tp1 -i entrada1.txt -o salida1.txt
$ cat salida1.txt
$
```

Los siguientes ejemplos son de transformadas simples. Observar que el primer uso del programa no especifica qué algoritmo usar, de manera que se tomará por defecto la FFT:

```
$ cat entrada2.txt
1 1 1 1
$ ./tp1 < entrada2.txt
(1, 0) (1, 0) (1, 0) (1, 0)
$ ./tp1 -m fft < entrada2.txt
(1, 0) (1, 0) (1, 0) (1, 0)
$ ./tp1 -m dft < entrada2.txt
(1, 0) (1, 0) (1, 0) (1, 0)
```

Similarmente, para anti-transformar:

```
$ cat entrada3.txt
(0, 0) (0, 0) (4, 0) (0, 0)
$ ./tp1 -m ifft < entrada3.txt
(1, 0) (-1, 0) (1, 0) (-1, 0)
$ ./tp1 -m idft -o salida3.txt < entrada3.txt
$ cat salida3.txt
(1, 0) (-1, 0) (1, 0) (-1, 0)
```

El siguiente ejemplo es análogo al anterior, aunque su entrada no tiene una cantidad de puntos potencia de 2. Notar que el programa implícitamente extiende la entrada con ceros hasta alcanzar la potencia de 2 más cercana (que en este caso es 4):

```
$ cat entrada4.txt
(0, 0) (0, 0) (4, 0)
$ ./tp1 -m ifft < entrada4.txt
(1, 0) (-1, 0) (1, 0) (-1, 0)
$ ./tp1 -m idft -o salida4.txt < entrada4.txt
$ cat salida4.txt
(1, 0) (-1, 0) (1, 0) (-1, 0)
```

5. Informe

El informe deberá incluir:

- Documentación relevante al diseño e implementación del programa.
- Documentación relevante a los algoritmos involucrados en la solución del trabajo.
- Documentación relevante al proceso de compilación: cómo obtener el ejecutable a partir de los archivos fuente.
- Las corridas de prueba, con los comentarios pertinentes.
- El código fuente, en lenguaje C++ (en dos formatos, digital e impreso).
- Este enunciado.

6. Fechas

La última fecha de entrega es el jueves 17 de mayo.

Referencias

[1] E. Gamma, R. Helm, R. Johnson J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st Ed., Addison-Wesley, 1994. Págs. 315-325.

[2] Alan V. Oppenheim, Roland W. Schafer. *Discrete-Time Signal Processing*, 2nd Ed., Prentice Hall, 1999. Cáp. 9.3

[3] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*, McGraw-Hill Higher Education, 2nd ed., 2001. Cáp. 30.2