75.04/95.12 Algoritmos y Programación II

Trabajo Práctico 2: Diseño, Implementación y Análisis C++

Universidad de Buenos Aires - FIUBA

Primer Cuatrimestre de 2018

1. Objetivos

Diseñar, implementar, y poner a prueba un programa en C++ que permita calcular la transformada rápida de Fourier (FFT, por sus siglas en inglés). Ejercitar conceptos de patrones, programación orientada a objetos, análisis de algoritmos, performance, testing, troubleshooting y portabilidad de programas.

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 continuamos nuestras implementaciones previas de la transformada de Fourier (TP0, TP1), procutando generar un programa con las siguientes características adicionales:

4.1. Diseño OO

El programa deberá adoptar un diseño orientado a objetos utilizando el patrón strategy (como hicimos en el TP 1). Adicionalmente, la implementación de objetos deberá estar alineado con las pautas de diseño explicadas en clase, y con el feedback provisto en los trabajos anteriores.

4.2. Análisis de complejidad de los algoritmos

El informe deberá incluir un análisis de complejidad temporal y espacial de los algoritmos implementados (DFT, FFT).

4.3. Análisis de complejidad del programa completo

De forma similar, cada grupo deber realizar un análisis de complejidad temporal y espacial del programa completo: es decir, incluyendo la complejidad vinculada a los procesos de entrada y salida de información.

4.4. Pruebas de performance

Reporte de las pruebas de performance realizadas, prestando especial atención a las predicciones de complejidad detalladas en los puntos anteriores.

Mediante estas pruebas, buscamos comprender cómo escalan los tiempos de ejecución a medida que se incrementa la longitud de la secuencia de entrada.

El informe deberá incluir además un análisis de la composición (apertura) de los tiempos de ejecución del programa, que permita responder las siguientes preguntas:

- ¿Qué fracción del tiempo se invierte en leer la entrada del programa?
- ¿Qué fracción del tiempo es usada para transformar o antitransformar esa información?
- ¿Qué fracción del tiempo se lleva la generación de la salida por pantalla y archivos?

4.5. Testing

El programa deberá permitir ejecutar casos de regresión automatizados como se explica a continuación.

Usando las opciones `-r` (archivo de regresiones) y `-e` (umbral de error, sección 4.8.2), el programa deberá comparar cada una de las secuencias de salida generadas con las líneas respectivas del archivo de regresión, calculando el módulo del vector de error relativo de la siguiente manera:

$$e_X = \sqrt{\frac{\sum |X[k] - R[k]|^2}{\sum |R[k]|^2}}$$

En donde X representa al vector de salida (calculado por el programa), mientras que R es el vector de regresión (leído del archivo pasado con `-r`), y e_X es el valor del error calculado para esa salida.

Para cada una de las regresiones ejecutadas, el programa deberá imprimir una línea por `std::cout` indicando el resultado de la regresión, comparando cada valor e_X con el umbral de error (opción `-e`). Por ejemplo:

```
$ cat input.txt
1 0 0 0
1 -1 1 -1
$ cat regressions.txt
(4, 0) (0, 0) (0, 0) (0, 0)
(0, 0) (0, 0) (4, 0) (0, 0)
$ tp2 -i input.txt -r regressions.txt
test 1: ok 4 1.3e-12
test 2: ok 4 1.2e-12
```

Por último, en este caso, el programa deberá finalizar con código `0` sólo cuando todos los casos del archivo de regresión han generado un resultado satisfactorio (dentro del umbral de error relativo).

El resto de los detalles de la interfaz de línea de comando y entrada/salida, están deteallados en la sección 4.8.

```
$ tp2 -i input.txt -o /dev/null -r regressions.txt
$ echo $?
0
```

4.6. Troubleshooting

Deberá usarse el módulo `memcheck` de Valgrind para analizar posibles *leaks* de memoria.

Deberá utilizarse alguna herramienta de análisis estático como cppcheck o Coverity C++ para detectar errores en el código fuente del trabajo práctico.

4.7. Portabilidad

El trabajo deberá poder correr en múltiples plataformas, compliadores y sistemas operativos.

4.7.1. Compiladores

Al menos 3 de los siguientes: llvm, gcc, Visual Studio, Intel C++, Digital Mars C++.

4.7.2. Sistemas operativos

El programa deberá soportar Linux y, adicionalmente, al menos 1 de los siguientes: Windows, FreeBSD.

4.8. Interfaz

4.8.1. Formato de los archivos de entrada y salida

En este trabajo adoptaremos el formato de los archivos que usamos durante el TP 1.

4.8.2. Línea de comando

A la interfaz de línea de comando del TP anterior, le agregaremos 2 opciones necesarias para poder ejectuar las regresiones:

- `-r`, o `--regression`, ubicación del archivo con el contenido de las regresiones. Este archivo contiene, en cada línea, la salida precalculada correspondiente con la entrada pasada en `-i`. Tiene el mismo formato que los usados en las opciones `-i` y `-o`. Cuando esta opción está presente, el programa deberá ejecutar las regresiones y generar una salida de acuerdo a lo explicado en la sección 4.5. En caso contrario, cuando la opción no está explícitamente en la línea de comando, el programa deberá comportarse en forma normal, de acuerdo a lo explicado en el TP 1.
- `-e`, o `--error`, para indicar el umbral del error relativo máximo a tolerar durante la ejecución de las regresiones. El valor por defecto de esta parámetro es `1e-3`.

4.8.3. Regresiones

Como vimos antes, la opción `-r` activa la ejecución de las regresiones. En este caso, la salida del programa deberá consistir, exclusivamente, de una línea de texto por cada secuencia de datos procesada (línea del archivo de entrada).

Para cada línea, el prograba deberá imprimir:

1. La palabra `test` seguida del número de secuencia de la prueba (comenzando con `1` para la primera regresión, luego `2`, y así sucesivamente)

   seguido de `:`

2. el resultado de ejecución de la prueba (`ok` o `error`)
3. la longitud del vector transformado (potencia entera de 2)
4. la magnitud del error relativo de acuerdo a la ecuación 4.5, expresada en notación científica.
5. *newline* (`\n`).

Durante la ejecución de las regresiones, el código de salida del programa deberá reflejar el resultado global de la ejecución del conjunto de regresiones: deberá ser `0` cuando todas las regresiones poseen un error menor al valor del umbral, y `1` cuando al menos una de las regresiones tenga un valor de error relativo igual o mayor al umbral controlado por la opción `--error`.

A continuación veremos algunos ejemplos de ejecución de casos, y del formato de salida a utilizar durante la ejecución de las regresiones.

4.9. 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
(4, 0) (0, 0) (0, 0) (0, 0)
```

A continuación, vamos a generar el archivo de regresiones usando la implementación de la DFT, y luego ejecutar las regresiones sobre el algoritmo FFT:

```
$ ./tp1 -m dft -o regressions2.txt < entrada2.txt
$ ./tp1 -m fft -r regressions2.txt < entrada2.txt
test 1: 4 0
$ echo $?
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)
$ cat regressions4.txt
(1, 0) (-1, 0) (1, 0) (-1, 0)
$ ./tp1 -m idft -r regressions4.txt -o salida4.txt < entrada4.txt
$ echo $?
0
$ cat salida4.txt
test1: 4 3.2e-11
```

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 14 de junio.

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