75.04/95.12 Algoritmos y Programación II
Trabajo práctico 1: Recursividad

Universidad de Buenos Aires - FIUBA
Segundo cuatrimestre de 2016
`$Date: 2016/10/27 17:28:40 $`

1. Objetivos

Ejercitar técnicas de diseño, análisis, e implementación de algoritmos recursivos.

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 será 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 7, y con una copia digital de los archivos fuente necesarios para compilar el trabajo.

4. Introducción

En este trabajo implementaremos una herramienta para procesar señales acuerdo con el flujo de cómputo que figura a continuación.

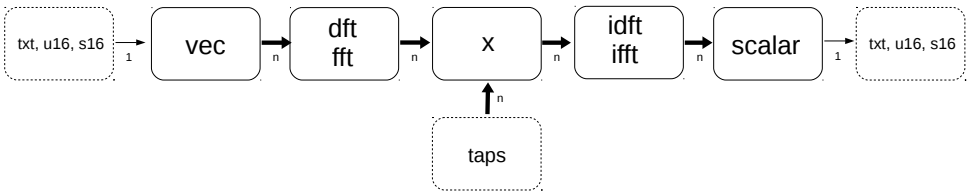


Figura 1: diagrama de flujo de procesamiento de señales

El programa recibirá un *stream* de números reales que ingresarán al flujo de procesamiento de señales de la figura 1, en donde se realizan tres funciones esenciales:

- **Transformación**. Las muestras de la señal son agrupadas y transformadas al dominio de la frecuencia en bloques de `n` muestras. Para esto, el programa deberá implentar los algoritmos de transformación DFT y FFT, teniendo en cuenta que tanto la selección de los algoritmos, como el tamaño de bloque `n` deben poder definirse mediante parámetros de configuración del programa.
- **Ecualización**. En esta etapa, el programa debe tomar el vector de muestras generado en el paso anterior, y multiplicarlo componente a componente por los coeficientes de ecualización, los cuales

pueden ser especifciados en la línea de comandos y por defecto (cuando no han sido definidos explícitamente) valen `1`: es decir, que la salida del programa deberá ser equivalente a la entrada cuando no se especifiquen los *taps* de ecualización en la línea de comandos.

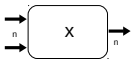
- **Anti-transformación**. Aquí se toman las muestras transformadas (en el dominio de la frecuencia), y se convierten al dominio temporal mediante las operaciones iDFT e iFFT.

4.1. Componentes

A continuación se detallan los bloques funcionales del flujo de procesamiento de señales presentado en la introducción.

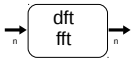
$\mathbf{y[n]} = \{x[1], \cdots, x[n]\}$

Vector. Construye un vector de $n = 2^s$ muestras a partir de un stream de escalares (números complejos), en donde s es un parámetro del programa (ver sección 5.1). Si el *stream* de entrada llegase a su fin (y no es posible leer n elementos del mismo), este componente completa el vector de salida con valores nulo, de tal forma que la cantidad de elementos siempre sea n.



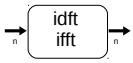
$\mathbf{z[n]} = \mathbf{x[n]} \cdot \mathbf{y[n]}$

Producto. Multiplica dos vectores complejos componente a componente.



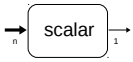
$\mathbf{y[n]} = \mathbf{fft}(\mathbf{x[n]})$
$\mathbf{y[n]} = \mathbf{dft}(\mathbf{x[n]})$

Transformada. Calcula la DFT o FFT del vector de entrada (vector de números complejos), en sentido directo.



$\mathbf{y[n]} = \mathbf{ifft}(\mathbf{x[n]})$
$\mathbf{y[n]} = \mathbf{idft}(\mathbf{x[n]})$

Anti-transformada. Calcula la transformación inversa del vector de entrada.



$y[n] = \mathbf{x[n]}$

Conversión a esaclar. Inserta los elementos del vector de entrada en el *stream* de escalares de salida.

5. Programa

Para configurar el funcionamiento del programa, usaremos parámetros pasados por la línea de comando, como hicimos en el TP anterior. Por defecto, el programa se limitará a leer, con formato, la secuencia de números complejos por el stream de texto de entrada `std::cin`, e imprimirá (también en forma de texto con formato) el resultado de la FFT por `std::cout`.

El programa deberá implementar el algoritmo FFT como también la transformada por definición (DFT). Dado que ambas operaciones son conceptualmente equivalentes, sea cual fuere el algoritmo de transformación seleccionado la salida del programa deberá permanecer invariante ante diferentes selecciones de los algoritmos de transformación.

Asimismo, para facilitar las tareas de *debug*, deberá permitir seleccionar las operaciones en sentido directo e inverso en forma independiente. Estos algoritmos están detallados en las secciones 6.3 y 6.4.

Si bien existen variantes de la FFT que permiten calculas transformaciones de secuencias de longitud arbitraria, en este TP nos limitaremos a realizar transformaciones de secuencias cuya longitud es potencia entera de 2. Para ello, el programa deberá completar con ceros las entradas que no cumplan con este requisito, llevando así su longitud a la potencia entera de 2 que resulte más cercana.

5.1. Interfaz

Como en el trabajo anterior, la interacción con el programa se dará a través de la línea de comando teniendo en cuenta las características descriptas a continuación.

Entrada. Los datos de entrada ingresan a nuestro programa en forma de números reales o enteros.
A lo largo de este TP, supondremos que la entrada/salida es realizada en formato texto: es decir, la entrada está conformada por una secuencia de números separados por espacios.

Salida. El *stream* de salida producido por este programa estará conformado por valores escalares separados por espacios. Las opciones de línea de comando a implementar en este TP son:

- `-o`, o `--output`, permite colocar la secuencia de números de salida en el archivo pasado como argumento; o por salida estándar -`std::cout`- si el argumento es "`-`".
- `-i`, o `--input`, para controlar el stream de entrada, de forma similar a la opción anterior; es decir, el programa leerá de `std::cin` cuando el valor pasado sea "`-`"..
- `-f`, o `--forward-op` permite especificar el algoritmo de cómputo: `dft` (sección 6.1), `idft` (6.2), `fft` (6.3), `ifft` (6.4), `fft-iter` e `ifft-iter` (10). En este TP, el valor por defecto a utilizar es `fft`.
- `-r`, o `--reverse-op` permite especificar el algoritmo de anti-transformación: `idft` (sección 6.1), `ifft` (6.3). El valor por defecto a utilizar es `ifft`.
- `-b`, o `--block-shift`. Controla el tamaño de bloque, es decir, el tamaño del vector de elementos del *stream* de entrada que se agrupan y transforman, de tal forma que si n es el tamaño de que, y s el argumento de este parámetro, entonces $n = 2^s$ (siempre sebe ser potencia de 2). Asimismo también controla la cantidad de coeficientes para multiplicar en el dominio transformado (ver opción `taps`). El valor por defecto a usar es `0`.
- `-t`, o `--taps`. Este parámetro apunta a un archivo conteniendo los 2^s coeficientes complejos que deben usarse para calcular la multiplicación en el dominio transformado. El valor por defecto a usar para los mismos es 1; es decir, cuando esta opción no se especifica en forma explícita, el programa utilizará un vector de 2^s coeficientes de valor unitario.

5.2. Ejemplos

Comencemos invocando al programa con las opciones por defecto, es decir, transformando con FFT los datos de `std::cin` y enviando la salida a `std::cout`:

```
$ cat entrada.txt
1 1 1 1
$ tp1 < entrada.txt
(1, 0) (1, 0) (1, 0) (1, 0)
$ tp1 -f fft < entrada.txt
(1, 0) (1, 0) (1, 0) (1, 0)
$ tp1 -f dft < entrada.txt
(1, 0) (1, 0) (1, 0) (1, 0)
```

Observamos que la entrada también puede ser compleja:

```
$ cat entrada2.txt
(1, 0) (0, 0) 0 (0, 0) 0 0 (0, 0) (0, 0)
$ tp1 < entrada2.txt
(1, 0) (0, 0) (0, 0) (0, 0) (0, 0) (0, 0) (0, 0) (0, 0)
```

Además, podemos invocar al programa para que realice los cálculos con DFT, y así validar los resultados anteriores:

```
$ tp1 -f dft < entrada2.txt
(1, 0) (0, 0) (0, 0) (0, 0) (0, 0) (0, 0) (0, 0) (0, 0)
$ tp1 -f dft -r idft < entrada2.txt
(1, 0) (0, 0) (0, 0) (0, 0) (0, 0) (0, 0) (0, 0) (0, 0)
$ tp1 -f fft -r idft < entrada2.txt
(1, 0) (0, 0) (0, 0) (0, 0) (0, 0) (0, 0) (0, 0) (0, 0)
$ tp1 -f dft -r ifft < entrada2.txt
(1, 0) (0, 0) (0, 0) (0, 0) (0, 0) (0, 0) (0, 0) (0, 0)
```

Generamos un vector de coeficientes nulo, de tal forma que la salida también lo sea.

```
$ echo 0 >coeficientes1.txt
$ tp1 -t coeficientes1.txt < entrada2.txt
(0, 0) (0, 0) (0, 0) (0, 0) (0, 0) (0, 0) (0, 0) (0, 0)
$ echo 0 0 >coeficientes2.txt
$ tp1 -t coeficientes2.txt -b 1 < entrada2.txt
(0, 0) (0, 0) (0, 0) (0, 0) (0, 0) (0, 0) (0, 0) (0, 0)
$ echo 1 1 >coeficientes3.txt
$ tp1 -t coeficientes3.txt -b 1 <entrada2.txt
(1, 0) (0, 0) (0, 0) (0, 0) (0, 0) (0, 0) (0, 0) (0, 0)
```

Por último, creamos un vector de coeficientes que permita amplificar los valores del *stream* de entrada.

```
$ echo 2 2 2 2 >coeficientes4.txt
$ cat entrada3.txt
(1, 0) (2, 0) 3 (0, 4) 5 6 (7, 0) (0, 0)
$ tp1 -r idft -t coeficientes4.txt -b 2 -i entrada3.txt
(2, 0) (4, 0) (6, 0) (0, 8) (10, 0) (12, 0) (14, 0) (0, 0)
```

5.3. Portabilidad

Es deseable que la implementación desarrollada provea un grado mínimo de portabilidad. Sugerimos verificar nuestros programas en alguna versión reciente de UNIX: BSD o Linux.

6. Algoritmos

6.1. DFT: discrete fourier transform (ida)

Este algoritmo permite calcular la transformación de una secuencia de N puntos $x(0), \ldots, x(N-1)$ dada por la fórmula:

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk}; k = 0, \ldots, N-1 \quad (1)$$
$$; W_N = e^{-j\frac{2\pi}{N}}$$

Es decir, ante una entrada de N valores complejos, producirá una secuencia de salida de igual tipo y longitud.

6.2. iDFT: inverse discrete fourier transform

En este caso queremos hacer la operación inversa: obtener el vector original, x, a partir del vector transformado X,

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) W_N^{-nk}; n = 0, \ldots, N-1 \quad (2)$$

Podemos ver entonces, que se trata de un cálculo muy similar al descripto en la sección anterior: las únicas diferencias están en un factor $1/N$, y en el signo negativo en el exponente de W.

6.3. FFT: fast fourier transform

Se trata de una variante de la DFT que permite obtener *eficientemente* una secuencia de N números X que verifique la fórmula 1.

Aplicando la técnica recursiva *dividir y conquistar*, y separando el vector de entrada x en dos subvectores p y q, que contienen las componentes de índices pares e impares respectivamente, se puede verificar lo siguiente:

$$X(k) = P(k) + W_N^k Q(k); k = 0, \ldots, N-1$$

En donde P y Q son las transformaciones p y q, y el operador de acceso a estos arreglos trabaja en forma modular (es decir, para cualquier par de enterios k y l, tendremos que $P(k) = P(k + l|P|)$). Si N es potencia entera de 2, podemos aplicar dividir y conquistar una y otra vez, y obtener así el siguiente algoritmo:

```
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;
}
```

6.4. iFFT: inverse FFT

En este caso, queremos hacer la operación inversa: obtener el vector original, x, a partir del vector transformado X, de manera más eficiente que la sugerida en la ecuación 2.

Por lo visto antes, podemos ver que se trata de un cálculo muy similar al de la FFT: como ya vimos en la sección 6.2, las únicas diferencias están en un factor $1/N$, y en el signo negativo en el exponente de W. Por ende, el proceso de derivar el algoritmo recursivo, que calcula eficientemente la fórmula 2, es análogo al descripto en la sección 6.3.

7. Informe

El informe deberá incluir:

- Documentación relevante al diseño e implementación del programa.
- Documentación relevante a los algoritmos involucrados en la solución del trabajo, incluyendo los detalles omitidos en la descripción del algoritmo de la sección 6.4.
- El análisis de complejidad espacial y temporal de todos los algoritmos involucrados, incluyendo los presentados en la sección **??**.
- 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.

8. Fechas

La última fecha de entrega es el jueves 17/11.

9. Optativo: cálculo *in-place*

Esta técnica permite eliminar el uso de los vectores p, q, P, y Q, presentados en el algoritmo de la sección 6.3. Gracias a esto, es posible ahorrar memoria.

Se trata entonces de cuantificar el ahorro, realizando análisis de complejidad espacial; e implementar esta variante del algoritmo en el programa. Para más información ver [1].

10. Optativo: cálculo iterativo

Es posible eliminar las llamadas recursivas en los algoritmos presentados en las secciones 6.3 y 6.4[1]: buscamos obtener un programa adicional que implemente esta variante iterativa, e incluir en el informe los análisis de complejidades, mediciones, y otros datos cuantitativos que permitan comparar ambas implementaciones objetivamente.

11. Optativo: formatos alternativos de entrada/salida

Idea: leer muestras de 16 bits con/sin signo tal como vienen del arecord, convertirlas a complejo, procesarlas, y reconvertir a 16 bits para enviar al aplay (se puede hacer en tiempo real usando pipes, y lograr una suerte de ecualizador de audio).

Referencias

[1] Alan V. Oppenheim, Roland W. Schafer. Discrete-time signal processing, second edition. Sección 9.3.1: In-Place Computations. Ed. Prentice Hall.

[1] Ver, por ejemplo, los gráficos de flujo se señal presentados en [1].