Juan Siverio Rojas

Práctica arquitecturas avanzadas

Filtro fir

Contenido

[Objetivo 2](#_Toc62667076)

[Previo 2](#_Toc62667077)

[Aplicación 2](#_Toc62667078)

# Objetivo

Implementar un filtro tipo FIR(filtro de respuesta finita) para un DSP (procesador de señales digitales) de la marca Texas Instruments, más concretamente el TMS320C6000 y en su posterior comprobación de los ciclos de reloj que tarda diferentes versiones del código, a la que se han de aplicar diferentes optimizaciones.

# Previo

La aplicación se ha programado utilizando C, y se ha utilizado un emulador de compilación y ejecución para tal efecto, el Code Composer Studio en su versión 5.5

* Instalación Code Composer Studio 5.5
* **Coeficientes.csv**: Fichero que contendrá los coeficientes del filtro FIR. *(opcional)*
* **Musica4.csv**: Contendrá los valores de la señal. *(opcional)*
* **Ejemplo\_comienzo.c:** Fichero principal de la aplicación.

# Aplicación

La aplicación obtiene la información de los coeficientes y valores de la señal leyendo dos archivos \*.CSV (archivo de texto separado por comas), **Coeficientes.csv y Musica4.csv**.

* La estructura de los ficheros es muy sencilla, simplemente un valor detrás de otro separado por comas. No hace falta un primer valor indicando el tamaño del fichero, ni el número de elementos, **la aplicación irá leyendo valores hasta llegar al final del texto.**
* En el caso de que alguno de los ficheros no exista, la aplicación usará unos valores por defecto que son:

**Coeficientes:** 0.33, 0.33, 0.33

**Datos:** 1,3,2,5,7,11,4,2,1,3

* Para establecer un límite máximo de datos, existe la constante **#define nvMax**, cuyo valor por defecto es de **3.500**

## Código de carga de ficheros

Se muestra el código utilizado para cargar los ficheros de datos, o en caso de error, utilizar los valores por defecto.

**float** v\_datos\_default [] = {1,3,2,5,7,11,4,2,1,3};

**float** v\_coef\_default [] = {0.33,0.33,0.33};

**float** salida,valorEntrada,borradorFloat;

**int** indiceValor,indiceCoeficiente,tmp,nv,nc;

**char** borradorCaracter;

**float** \* **restrict** v\_coef; ///puntero hacia array con los coeficientes leidos desde fichero csv

**float** \* **restrict** v\_datos; ///puntero hacia array con los datos leidos desde fichero csv

///

///Primero intento abrir fichero de coeficientes

///

FILE \*fich\_num = **fopen**("Coeficientes.csv","r");

**if** (fich\_num == NULL){

///

///si no se encontró o pudo abrir, lo indico y utilizao el array por defecto.

///

**printf**("\nNo se puede abrir fichero: Coeficientes.csv");

**printf**("\nUsando valores por defecto");

nc=3;

v\_coef = v\_coef\_default;

}

**else**

{

nc=0;

**while** (!**feof**(fich\_num))

{

**fscanf**(fich\_num,"%f",&borradorFloat); ///leo el valor en flotante

nc++; ///numero de coeficientes que he leido

**if** (!**feof**(fich\_num))

**fscanf**(fich\_num,"%c",&borradorCaracter); ///leo la coma.

}

v\_coef = (**float** \*) **malloc**(nc \* **sizeof**(**float**));

**fseek**(fich\_num,0,SEEK\_SET); ///me vuelvo a colocar al principio del fichero

nc=0;

**while** (!**feof**(fich\_num))

{

**fscanf**(fich\_num,"%f",&v\_coef[nc]); ///leo el valor en flotante

nc++; ///numero de coeficientes que he leido

**if** (!**feof**(fich\_num))

**fscanf**(fich\_num,"%c",&borradorCaracter); ///leo la coma.

}

**fclose**(fich\_num);

}

///Imprimo los coeficientes leidos.

**for** ( tmp = 0; tmp < nc; tmp++)

**printf**("\nCoeficiente %d: %lf",tmp,v\_coef[tmp]);

///

///Ahora intento lo mismo con fichero de datos

///

fich\_num = **fopen**("musica4.csv","r");

**if** (fich\_num == NULL){

///

///si no se encontró o pudo abrir, lo indico y utilizao el array por defecto.

///

**printf**("\nNo se puede abrir fichero: musica4.csv");

**printf**("\nUsando valores por defecto");

nv=10;

v\_datos = v\_datos\_default;

}

**else**

{

///aqui leo el fichero y voy contando para saber cuantos memoria tengo que asignar realmente.Así

//no dependo de un tamaño prefijado, sino depende del tamaño del fichero.

nv=0;

**while** ((!**feof**(fich\_num)) && (nv < nvMax)) ///limite de valores a leer.

{

**fscanf**(fich\_num,"%f",&borradorFloat); ///leo el valor en flotante

nv++; ///numero de coeficientes que he leido

**if** (!**feof**(fich\_num))

**fscanf**(fich\_num,"%c",&borradorCaracter); ///leo la coma.

}

v\_datos = (**float** \*) **malloc**(nv \* **sizeof**(**float**));

**fseek**(fich\_num,0,SEEK\_SET); ///me vuelvo a colocar al principio del fichero

nv=0;

**while** ((!**feof**(fich\_num)) && (nv < nvMax)) ///establezco un límite de valores en 3500

{

**fscanf**(fich\_num,"%f",&v\_datos[nv]); ///leo el valor en flotante

nv++; ///numero de coeficientes que he leido

**if** (!**feof**(fich\_num))

**fscanf**(fich\_num,"%c",&borradorCaracter); ///leo la coma.

}

**fclose**(fich\_num);

}

///Imprimo los valores leidos

**for** ( tmp = 0; tmp < nv; tmp++)

**printf**("\nValor %d: %lf",tmp,v\_datos[tmp]);

**Nota:** Algo interesante en el código anterior, es la siguiente porción, en la que se recorre los ficheros incrementando un contador por cada nuevo valor, para saber exactamente la cantidad de memoria a reservar después, evitando así depender de una cantidad fija o teniendo que especificar el número de elementos en el fichero.

Como contrapartida, esta operación requiere de la lectura del fichero dos veces, una para calcular el número de elementos y otra para realmente leerlo y cargarlo en memoria.

**///aqui leo el fichero y voy contando para saber cuantos memoria tengo que asignar realmente.Así**

**//no dependo de un tamaño prefijado, sino depende del tamaño del fichero.**

**nv=0;**

**while ((!feof(fich\_num)) && (nv < nvMax)) ///limite de valores a leer.**

**{**

**fscanf(fich\_num,"%f",&borradorFloat); ///leo el valor en flotante**

**nv++; ///numero de coeficientes que he leido**

**if (!feof(fich\_num))**

**fscanf(fich\_num,"%c",&borradorCaracter); ///leo la coma.**

**}**

## Código base de filtro FIR

El código principal del filtro FIR implementado, el encargado del cálculo de valores, está en el fichero **Ejemplo\_comienzo.**c y es el siguiente:

///Una vez rellenados los vectores de coeficientes y de datos, procedo al cáculo de la salida

///bucle que comprobará todas las entradas posibles.

1: **for** (indiceValor = 0; indiceValor < nv; indiceValor++)

{

2: salida=0.0; ///inicialmente salida = 0

//bucle que realiza el cáculo para una entrada dada

3: **for** (indiceCoeficiente=0; indiceCoeficiente < nc;indiceCoeficiente++)

///tengo que sumar si o sí, mientras no recorra todos los coeficientes.

{

4: tmp=indiceValor-indiceCoeficiente;

5: **if** ((tmp >= 0) && (tmp < nv)) ///controlo que el índice de datos esté dentro del rango, si no es así, pongo el valor a cero, para que al multiplicar de cero y no aumente nada su valor en la suma.

///esto puede pasar cuando el valor del índice es más pequeño que el número de coeficientes que hay, por ejemplo.

6: valorEntrada= v\_datos[tmp];

7: **else**

8: valorEntrada = 0.0;

9: salida = salida + (v\_coef[indiceCoeficiente]\*valorEntrada);

}

10: **printf**("\nIteración : %d Valor: %lf",indiceValor,salida);

}

## Consideraciones:

* La línea 10, se elimina de las pruebas de tiempo de ejecución para no influir en los tiempos reales.
* La línea 5, controla que el índice apunte a una posición existente del vector de datos, para evitar acceder a direcciones del vector que no existan o con valores desconocidos. Esto realmente no es lo más óptimo, ya que conlleva una comparación en cada iteración, pero si ayuda a que la función sea más portable y adaptable a situaciones de tamaños de coeficientes o datos cambiantes.
* El código principal usa doble bucle sin ninguna optimización a nivel de Pragmas o intrínsicos

Para 21 Coeficientes y 3500 datos, la versión principal tarda:

19,075,093