Radio Taller Fourier

Laboratorio 5

Filtros II

1. Introducción

En el laboratorio anterior dimos nuestros primeros pasos en el uso de filtros y verificamos algunas de sus propiedades. ¿Pero de dónde salen los taps que usa GNU Radio para filtrar? ¿Qué compromisos hay a la hora de diseñar un filtro? En este laboratorio exploraremos algunas técnicas clásicas de diseño de filtros, comenzando por filtros con respuesta a impulso infinita (IIR) y siguiendo por los de respuesta finita (FIR). Luego retomaremos el sistema que adapta distintas tasas de muestreo entre la entrada y la salida que comenzamos a ver en el Laboratorio 3 (Muestreo y Multi-Rate Signal Processing II) y trataremos de lograr una implementación realmente eficiente mediante lo que se denomina un banco de filtros (varios filtros en paralelo).

2. Diseño de filtros IIR

Con el surgimiento del procesamiento digital, las técnicas en ese dominio siguieron lo que históricamente se venía usando hasta ese momento en el mundo analógico. Dado que la amplia mayoría de los filtros analógicos son IIR (en particular por la presencia de elementos capacitivos o inductivos), las herramientas clásicas de diseño de filtros digitales generan filtros IIR. El diseño consta básicamente del cálculo de los ceros y polos de la función de transferencia, el cual a su vez parte del diseño de un filtro de tiempo continuo. La metodología consiste en: dadas las especificaciones de diseño del filtro discreto, transformarlas a especificaciones de diseño de filtro de tiempo continuo, diseñar el filtro en tiempo continuo, y finalmente, por medio de alguna transformación (invarianza a respuesta impulso o bilineal), obtener los coeficientes del filtro buscado en tiempo discreto.

En esta parte exploraremos una técnica muy utilizada en la práctica llamada de Butterworth. En tiempo continuo, el mismo tiene la siguiente respuesta en frecuencia:

$$|H(f)|^2 = \frac{1}{1 + (f/f_c)^{2N}}, \tag{1}$$

donde f_c es la frecuencia de corte del filtro y N su orden.

1. Usando su programa de cálculo favorito grafique (en escala logarítmica) la respuesta en frecuencia en función del orden N del filtro para cierto f_c. ¿Cómo es el comportamiento en banda pasante y en banda de rechazo del filtro? ¿Como cambia a medida que aumenta N?

Se quiere diseñar un filtro de Butterworth que en el dominio digital cumpla con las siguientes especificaciones:

$$\begin{cases} -1dB \leq |H(e^{j\theta})|_{dB} \leq 0dB, & 0 \leq |\theta| \leq 0{,}2\pi \\ |H(e^{j\theta})|_{dB} \leq -15dB & 0{,}3\pi \leq |\theta| \leq \pi \end{cases}$$

2. Halle valores para N y f_c en (1) (es decir, en el dominio continuo) que cumplan con las especificaciones anteriores. Para esto, deberá suponer una cierta tasa de muestreo. ¿Es realmente importante este valor?

Como dijimos antes, ahora que tenemos el filtro en el dominio continuo debemos pasar al dominio discreto. Típicamente este proceso se realiza haciendo un mapeo entre la variable s de la transformada de Laplace (tiempo continuo) a la variable z de la tranformada del mismo nombre (en tiempo discreto), y luego calculando la nueva función de transferencia como cociente de polinomios en z. De esta forma podremos implementar el filtro con retardos y realimentaciones. Para este taller, será suficiente usar alguna biblioteca que se encargue de este cálculo, aunque es importante tener claro los fundamentos de estas operaciones. En el EVA podrá encontrar material al respecto.

En este caso nos apoyaremos en una función de la biblioteca SciPy de Python, la que (al ser justamente Python) puede ser usada desde dentro del Companion. Para ello, agregue un bloque `Import`, que en este caso tendrá la siguiente sentencia: `import scipy.signal as sig`. De esta forma podemos usar el módulo signal de SciPy dentro del flowgraph.

3. La función en el módulo signal de la biblioteca SciPy que usaremos se llama `butter`. Lea y entienda los parámetros de la misma. Utilice esta función dentro de un bloque `Variable` para luego asignar los valores en un bloque `IIR filter`. Como siempre, lea con atención la documentación de este bloque. Verifique que el filtro cumple con las especificaciones deseadas.

Una desventaja de los filtros IIR es la inestabilidad numérica. Esto se puede dar cuando algún polo del sistema queda cerca del circulo unidad. Debido a la precisión finita de las variables que se utilizan en los cómputos, al escribir la función de transferencia en función de los polos y ceros puede ocurrir que algún polo quede fuera del circulo unidad, provocando la inestabilidad del sistema.

4. En esta parte trataremos de observar este fenómeno. Cambie el valor de la variable `samp_rate` a 1MHz. Genere un filtro IIR con parámetros $N = 8$ y $f_c = 1$kHz. ¿Qué se observa?

5. Para asegurarnos que lo que está sucediendo es un problema de inestabilidad, verificaremos si los polos del filtro se encuentran dentro del circulo unidad. Para esto, utilizaremos la función de la biblioteca numpy: `np.roots`. Al igual que antes, es necesario importarla mediante la siguiente sentencia: `import numpy as np`. Cree una nueva variable que almacene las raíces del denominador de la función de transferencia del filtro y verifique si se cumple la condición de estabilidad. ¿Qué sucede a medida que sube f_c?

Otra desventaja que presentan los filtros IIR es la no linealidad en la fase. Es decir, al filtrar una señal con distintas componentes en frecuencia, el filtro otorga distintos retardos para distintas frecuencias.

6. Pruebe que si un filtro continuo atrasa un tiempo τ constante a todas las exponenciales complejas, la fase de $H(f)$ es lineal en f. Verifique que este no es el caso para el filtro implementado en la parte 3.

7. Utilice esta misma implementación para filtrar la suma de tres tonos de distinta frecuencia que se encuentren en banda pasante del filtro. Compare la señal de entrada con la señal de salida y verifique que son cualitativamente distintas. Quizá le sea más fácil esta visualización usando un `Trigger` en el `QT GUI Time Sink` y sobre-muestreando la señal antes del Time Sink. Recuerde separar las frecuencias de los tonos para que la diferencia de retardo sea notoria.

3. Diseño de filtros FIR

Tratando de salir del esquema clásico de pasar del mundo continuo al discreto, una diferencia sumamente importante es que en el mundo digital se pueden implementar filtros con respuesta a impulso realmente finita, con algunas ventajas que veremos durante este laboratorio. Aquí nos enfocaremos sobre todo en la técnica de diseño denominada *por enventanado* (que es la que usan los bloques pasa-* de GNU Radio) y usaremos como caso de uso un filtro pasa-bajos.

8. Suponga que tenemos las muestras de una señal continua muestreada a una tasa f_s (y por supuesto con ancho de banda menor a $f_s/2$ para estar dentro de las condiciones del teorema de muestreo). El objetivo es filtrarla de manera que una vez reconstruida (o sea, interpolada a tiempo continua nuevamente) tenga ancho de banda $W < f_s/2$. Pruebe que los taps del filtro a tiempo discreto (ideal) que logra lo anterior tienen la siguiente expresión:

$$h[n] = \frac{2W}{f_s}\text{sinc}\left(\frac{2W}{f_s}n\right). \tag{2}$$

9. Halle al menos dos problemas que tiene implementar este filtro.

La solución más sencilla para resolver el hecho de que el filtro tiene infinitos taps es sencillamente recortando (o *enventanando*) $h[n]$, lo que equivale a usar los $L+1$ taps correspondientes a $-L/2 \leq n \leq L/2$ (suponiendo L par). Vamos a explorar qué sucede a medida que cambiamos L. Para esto vamos a usar el bloque `Interpolating FIR Filter`, en el que se debe asignar la variable `Taps`, correspondiente justamente a los taps del filtro que se quiera evaluar (en forma de lista de Python o array de Numpy). Por ejemplo, si quisiéramos que la salida del filtro fuera un promedio de las últimas 4 muestras, en `Taps` se debe escribir `[0.25, 0.25, 0.25, 0.25]`.

Armar esta lista para el caso del filtro (2) es relativamente sencillo si sabemos usar Python y Numpy.

10. Defina las variables `cutoff` y `ntaps` que se podrán cambiar en tiempo de ejecución (es decir, usando un `QT GUI Range`). Atención porque `ntaps` indicará cuántos taps tendrá el filtro, y por lo tanto deberá ser del tipo entero. La variable `cutoff` será el ancho de banda del filtro (W en la ecuación (2)). Luego agregue un bloque `Import` con la siguiente sentencia: `import numpy as np`. De esta forma podemos usar el módulo numpy dentro del flowgraph. Finalmente, defina una variable que contenga la siguiente sentencia: `[2*cutoff/samp_rate*np.sinc(2*cutoff/samp_rate*n) for n in range(-ntaps//2,ntaps//2+1)]` y asígnela en el bloque `Interpolating FIR Filter`. Trate de entender la sentencia tanto en términos de la ecuación (2) como de Python.

11. Explore el efecto de cambiar `ntaps` y `cutoff`. En particular conteste, ¿cuántos taps se necesitan si se quiere una atenuación de 60 dB fuera de la banda pasante?

En vez de cortar los taps de la expresión (2), se puede usar una función de ventana que suavice los bordes, de la misma forma que se hizo para analizar el espectro en la primera práctica. Numpy cuenta con algunas ventanas (ver `https://numpy.org/doc/stable/reference/routines.window.html`).

12. Cree otra variable con la respuesta impulso enventanada y compare la función de transferencia resultante con la original. Para graficar ambas funciones de transferencia a la vez, puede cambiar el parámetro `Number of Inputs` del `QT GUI Frequency Sink` a 2. Repita la pregunta 11 para algunas ventanas (en particular, explore la ventana por defecto de GNU Radio). ¿Qué compromiso hay al elegir la ventana? En particular, para cierto tamaño de ventana, ¿qué ventana cae más rápido? ¿cuánto atenúa por fuera de la banda?

Finalmente vamos a verificar cómo se comportan los FIR respecto a estabilidad y linealidad de fase.

13. Repita los experimentos de las partes 4 y 7 para un filtro pasa-bajos de GNU Radio. ¿Qué observa?

4. Bancos de filtros

Finalmente retomamos el problema planteado al final del Laboratorio 3.

14. Repase sus respuestas del Laboratorio 3 referidas a qué procesamiento es necesario para interpolar o decimar una señal. En particular, dibuje el diagrama (por ahora en papel) del sistema que permite adaptar cualquier tasa de entrada a cualquier tasa de salida, siempre que el cociente entre éstas sea un número racional.

15. Descargue del EVA del curso un archivo `~24.wav` e implemente en GNU Radio el sistema diseñado de tal manera que pueda escucharlo correctamente en una tarjeta a 16 kHz.

16. Repita el experimento anterior pero utilizando un archivo `~44_1.wav` y una tarjeta configurada a 48 kHz. ¿Porqué cree que el sistema está consumiendo tanto procesador?

Existen al menos dos formas de disminuir el procesamiento. El primero es encontrar la mínima expresión del cociente entre los factores de interpolación y decimado (o sea, dividirlas entre el máximo común múltiplo). Incluso se puede realizar en varias etapas, factorizando el resultado, lo que se conoce como *multi-stage interpolation*. Es decir, por ejemplo si se quisiera decimar por 4, se decima en dos etapas de 2.[1]

17. Implemente un interpolador en tres etapas que adapte el audio de 44.1 kHz a 48 kHz. Verifique el ahorro en procesamiento. ¿Cuál es la ventaja de hacer el re-muestreo en varias etapas?

De todas formas, seguramente el procesamiento del nuevo sistema siga siendo excesivo. Afortunadamente, todavía hay margen para mejorar. Por ejemplo, si la etapa de decimado es de orden M, se descartan $M-1$ salidas del filtro. Si el filtro es FIR (o sea, sin re-alimentación), ¿para qué las calculamos si después las vamos a tirar?

18. La etapa de interpolado es un poco más sutil. ¿Qué sabemos sobre la entrada al filtro en la etapa de interpolado que podamos usar para minimizar el procesamiento?

19. Una manera de aprovechar las dos observaciones anteriores es mediante las denominadas "identidades de Noble" y los filtros polifásicos que resultan de su aplicación. Busque información sobre estas identidades y explique cómo se usan para construir un interpolador y un decimador. Verifique el poder de ahorro de esta arquitectura usando el bloque `Rational Resampler` que los implementa, y compare el uso del CPU contra su implementación.

[1] Esta idea se usa también generalmente en filtros IIR, donde la implementación es en etapas y cada etapa consta de un filtro con dos polos y dos ceros (conocido como *series second-order sections* o simplemente SOS).