Instituto Tecnológico de Costa Rica
Escuela de Ingeniería Electrónica
Profesor: Dr. Pablo Alvarado Moya
EL 5805 Procesamiento Digital de Señales
II Semestre 2024

Entrega: **1 de octubre, 2024**

Versión: 17 de septiembre de 2024

Tarea 3: Filtrado en el tiempo

En esta tarea tiene varios objetivos:

- Diseñar filtros digitales utilizando GNU/Octave y la funciones en el paquete `signal`.
- Graficar respuestas en frecuencia de filtros digitales.
- Implementar varias estrategias de cascada de filtros bicuadráticos en C++, para el procesamiento en tiempo real.
- Utilizar perfiladores para determinar la estrategia más eficiente de implementación de la cascada de filtros digitales.

Debe utilizar la siguiente invitación al Github Classroom para iniciar su repositorio. Note que allí está una base de código muy similar a la utilizada en el proyecto 1, con la diferencia de que permite repetir la ejecución de los archivos `.wav`.

El código construye un ejecutable llamado `tarea3`, que puede utilizar con

```
> tarea3 -f audio1.wav audio2.wav audio3.wav
```

Una vez que se han reproducido todos los archivos, se reactiva la captura desde el micrófono. Con la tecla '`r`' (de *replay*) vuelven a reproducir los archivos indicados en la línea de comando.

Usted debe agregar la opción pasatodo con la tecla '`p`' (*passthrough*), para poder desconectar los filtros y escuchar la función original. También puede copiar la opción de volumen del proyecto 1, porque en ocasiones usted deseará poder escuchar detalles que en el volumen original serán difíciles de percibir.

I. Diseño de filtros digitales

La primera consideración en el diseño de filtros digitales es la selección correcta de la frecuencia de muestreo F_s, pues de ella dependen enteramente los valores de frecuencia normalizada a utilizar. Usted debe decidir cuál frecuencia va a utilizar en Jack, y configurar todos los diseños acorde con esa decisión. En otras palabras, lo que usted use como frecuencia de muestreo en el diseño de sus filtros, debe ser lo que configure en el Jack también. Por ello, en sus scripts de GNU/Octave debe permitir seleccionar la frecuencia de muestreo como una variable en un único lugar, que luego emplea en todos los cálculos, por si debe cambiar su configuración, que le sea fácil recalcular todos los filtros (por ejemplo, no le da tiempo de procesar y necesita bajar esa frecuencia).

Se recomienda crear dos archivos `.wav` de prueba: ruido blanco, y un barrido de frecuencia, no muy extensos como para que cada prueba sea una tortura por su larga duración, pero no tan cortos

que cueste escuchar lo que ocurre. Con estos archivos usted tiene dos ayudas perceptuales, para diagnosticar si sus filtros operan correctamente.

Usted utilizará las funciones `ellip`, `butter`, `cheby1` y `cheby2` para diseñar sus filtros digitales. Aquí una **advertencia** importante: por razones históricas, poco claras, Matlab, y por ende GNU/Octave, usan como rango de frecuencias de diseño el intervalo $[0; 1]$, en donde 1 corresponde a la frecuencia de Nyquist, es decir $F_s/2$. En otras palabras, lo esperado por estas funciones es el doble de la frecuencia normalizada tradicional $f = F/F_s$, y entonces las variables `Wp`, `Wl` y `Wh` se indican con $2f = F/(F_s/2)$.

1. Usted deberá realizar un script `design_filters.m` para diseñar filtros

   - de tercer orden
   - con un rizado en banda de paso de 1 dB
   - con atenuación en banda de rechazo de 50 dB

   Con los cuatro métodos (`ellip`, `butter`, `cheby1` y `cheby2`), los filtros a diseñar son:

   - Filtros paso bajos con frecuencia de corte 440 Hz
   - Filtros paso altos con frecuencia de corte 600 Hz
   - Filtros paso bandas con frecuencias de corte inferior y superior de 220 Hz y 1000 Hz.
   - Filtros supresores de banda con frecuencias de corte inferior y superior de 220 Hz y 1000 Hz.

   Cada filtro se deberá implementar como una cascada de filtros de segundo orden, conversión que realizará con la función `tf2sos`. Esta función le retorna una matriz, con los coeficientes de cada filtro de segundo orden en una fila, y un número de filas igual al número de etapas del filtro completo. En cada fila, las primeras tres columnas corresponden a los coeficientes de los numeradores y las últimas tres columnas corresponden a los coeficientes de los denominadores:

   $$\begin{bmatrix} b_0^{(1)} & b_1^{(1)} & b_2^{(1)} & 1 & a_1^{(1)} & a_2^{(1)} \\ b_0^{(2)} & b_1^{(2)} & b_2^{(2)} & 1 & a_1^{(2)} & a_2^{(2)} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_0^{(N)} & b_1^{(N)} & b_2^{(N)} & 1 & a_1^{(N)} & a_2^{(N)} \end{bmatrix} \tag{1}$$

   donde la i-ésima etapa de segundo orden es:

   $$H^{(i)}(z) = \frac{b_0^{(i)} + b_1^{(i)} z^{-1} + b_2^{(i)} z^{-2}}{a_0^{(i)} + a_1^{(i)} z^{-1} + a_2^{(i)} z^{-2}}$$

   Usted entonces debe almacenar con

   ```
   save("método_tipo.mat", "SOS")
   ```

   cada uno de los 16 filtros, donde `método` debe ser `ellip`, `butter`, `cheby1` o `cheby2`, y `tipo` debe ser `lowpass`, `highpass`, `bandpass`, `stopband`. `SOS` es el nombre de la variable a almacenar, que será la matrix que produce `tf2sos`, esto es, la ganancia ya se espera esté integrada en alguna de las etapas.

2. Implementar una función `viewfreqresp` que recibe el nombre del archivo con la la configuración de un filtro y la frecuencia de muestreo (que debe tener algún valor por defecto), y muestra el diagrama de polos y ceros y la respuesta en frecuencia.

   Usted puede recuperar la matriz `SOS` con

   ```
   Data=load("método_tipo.mat", "SOS")
   ```

   pero `Data` será una estructura, y para acceder a `SOS` deberá usar `Data.SOS`.

   La funcion `sos2fs` le permite recuperar el filtro completo con todas las etapas.

   Para la respuesta en magnitud deberá construir su propio código que le permita controlar exactamente cómo mostrar la información: se desea la frecuencia en hertz, el ángulo en grados, la respuesta en magnitud en decibeles, tal y como se muestra en la figura 1.

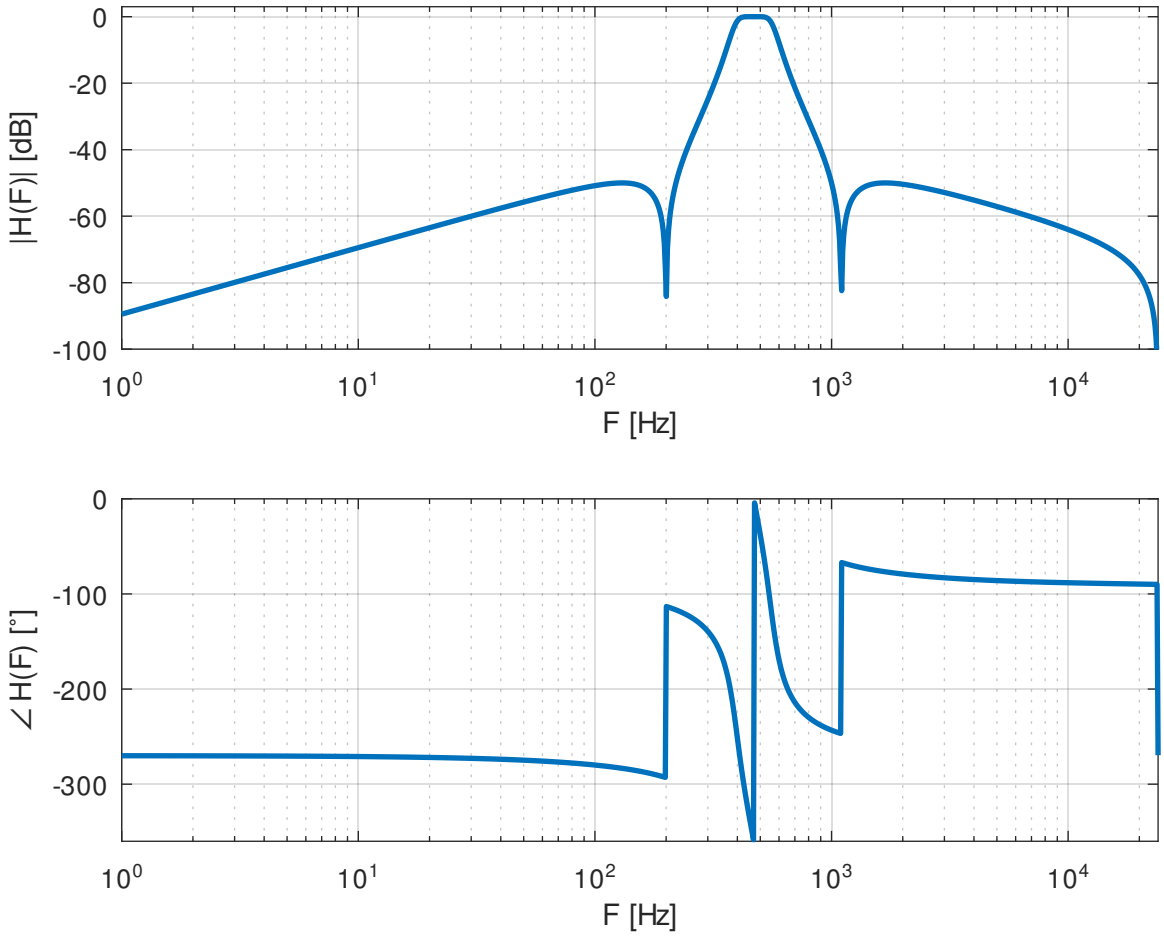

Figura 1: Respuesta en frecuencia con las unidades y rangos deseados.

La función `bode(filt(B,A))` ya construye *casi* lo que se busca, pero brinda poco control sobre las unidades y rangos mostrados, lo que se presta a confusión, y por eso no debe emplearse.

Para hacer el plot de fase y magnitud, se recomienda reconstruir $H(z)$ utilizando `polyval` para numerador y denominador, y luego evaluar dicha función como corresponda para encontrar la respuesta en frecuencia.

Deberá además utilizar los ejes logarítmicos o lineales, según corresponda, para lograr que el plot esté dado en decibeles [dB], y los ángulos en grados, y la frecuencia de 1 Hz a $F_s/2$. Las funciones de GNU/Octave `logspace`, `semilogx` le serán de utilidad.

Usted va a tener que manipular la fase, para que aparezca en rangos sencillos de interpretar (usualmente de $-360°$ a $0°$).

Deberá utilizar `axis` para que los rangos mostrados no se salgan de control, etiquetar los ejes con `xlabel` y `ylabel` y mostrar la rejilla.

Para el diagrama de polos y ceros basta emplear `zplane`.

3. Construya un script `listen_filter` que también recibe el nombre del archivo del filtro, el nombre de un archivo `.wav` y la frecuencia de muestreo (con un valor por defecto), que permita escuchar el archivo, al que le es aplicado el filtro con la función `filter`.

   Debe determinar si el archivo `wav` brindado utiliza una frecuencia de muestreo distinta a la indicada por usted en la función, y de ser así, utilice la función `resample` para asegurar que las muestras del archivo utilizan la frecuencia de muestreo por usted indicada en la función (con la que fueron diseñados los filtros).

   El objetivo de esta parte es tener alguna referencia para escuchar luego si la implementación en C++ suena igual.

II. Implementación de filtros digitales

El código brindado como referencia es el mismo código pasa todo del proyecto. La idea es que usted cree su propio cliente, llamado en esta tarea `filter_client`, que tenga la funcionalidad de filtrado.

Ese código tiene además un ejemplo para cargar los coeficientes de los archivos generados en GNU/Octave, con la opción `-c`. Se provee una función plantilla declarada en `parse_filter.h` que retorna un vector de "etapas" de segundo orden, donde los coeficientes de cada etapa se almacenan de forma cruda en un `std::vector` en el orden indicado en (1). Usted puede modificar todo este código de lectura del archivo a lo que usted requiera en la función `main.cpp` y su cliente para filtrado.

1. Cree una clase `biquad` que implemente el filtrado con un filtro de segundo orden IIR. Usted debe definir métodos para inicializar el filtro, tanto sus coeficientes como su estado, de modo que sea fácil integrarlo a su cliente de filtrado.

   La clase `biquad` tendrá varios métodos de procesamiento. Para este punto, basta uno para procesar todo el bloque de entrada, tal cual lo recibe `process` del cliente de Jack:

```
void biquad::process(jack_nframes_t nframes,
                     const sample_t *const in,
                     sample_t *const out) {
  // ...
}
```

Procure que su código sea lo más rápido posible.

2. Cree algún filtro de segundo grado a su gusto, que se active cuando el usuario presiona la tecla '`p`' (de *prueba*), y que utiliza la clase anterior para operar.

3. Ahora cree una clase `cascade` que contenga cuantos bloques `biquad` sean necesarios, para poder realizar los filtros diseñados por usted en GNU/Octave en la primera parte de la tarea.

   Dicha clase también debe tener un método `process` que opere todo el bloque, pero encadenando de alguna manera todos los `biquad` en cascada.

   Para esto hay varias opciones: cada `biquad` opera el búfer completo, uno detrás de otro. Quizá esta sea la forma más fácil de implementar, pero esto producirá muchas fallas de caché y afectará la velocidad, porque debe recorrer el búfer completo varias veces. Otra forma es encadenar el procesamiento muestra por muestra, y aquí hay muchas formas de aprovechar las características avanzadas de C++ para asegurar que el código generado es óptimo (no se hacen llamadas a funciones, se realiza *loop unrolling*, etc. Varias técnicas aprovechan algunos cálculos o decisiones en tiempo de compilación, que ahorran entonces cálculos en tiempo de ejecución.

   La idea es que usted tenga varias estrategias que evaluar, para ofrecer la versión más rápida ¡Hay un 15 %, 10 % y 5 % de puntos extra para los primer, segundo y tercer lugares del código más rápido en el grupo!

4. Integre su clase `cascade` de modo que el filtro especificado en línea de comandos se aplique a la señal de entrada, cuando el usuario presiona la tecla '`c`'.

III. Evaluación

Para evaluar sus opciones, utilice algún perfilador (por ejemplo VTune de Intel o gperftools de google), para evaluar qué tan rápidas son sus funciones de filtrado. También existe el perfilador integrado en `g++`, pero es mucho más lento de evaluar, y requiere una compilación particular para insertar código de instrumentación en el código, con la opción `-p`.

Note que usted eventualmente deberá alterar `meson.build` para especificar las banderas necesarias en el código generado, que usualmente es integrar los símbolos de fuente (con `-g`) y activar las banderas de optimización (con `-O3`).

IV. Otras indicaciones generales

Esta tarea se realiza en los mismos grupos de trabajo del proyecto 1.

Entregables:

1. Código GNU/Octave.
2. Código C++ con la implementación de los filtros.
3. Los archivos `.mat` con los 16 filtros.
4. Archivo PDF con las gráficas de respuesta en frecuencia de los filtros diseñados, y con los resultados del perfilado de las técnicas evaluadas por el grupo.

5. El desarrollo debe quedar documentado paso a paso en Git.

6. El código y PDF deben ser subidos al tecDigital.

Toda tarea debe ser resultado del trabajo intelectual propio de la persona o personas que la entregan. Además de la literatura de referencia, solo puede utilizarse el material expresamente así indicado en la tarea, lo que incluye código brindado por el profesor o indicado en los enunciados a ser utilizado como base de la tarea. Expresamente quedan excluidos como material de referencia los trabajos entregados por otros estudiantes en el mismo semestre o semestres anteriores.

Nótese que esto no elimina la posibilidad de discutir estrategias de solución o ideas entre personas y grupos, lo cual es incluso recomendado, pero la generación concreta de cada solución, derivación o programa debe hacerse para cada entrega de forma independiente.

Para toda referencia de código o bibliografía externa deben respetarse los derechos de autor, indicando expresamente de dónde se tomó código, derivaciones, etc. Obsérvese que el código entregado por el profesor usualmente ya incluye encabezados con la autoría correspondiente. Si un estudiante agrega código a un archivo, debe agregar su nombre a los encabezados si la modificación es de más del 50 % del archivo, o indicar expresamente en el código, con comentarios claros, la autoría de las nuevas líneas de código, pues a la autora o al autor del archivo original no se le debe atribuir código que no es suyo.

Si se detecta código o deducciones teóricas iguales o muy cercanas a trabajos de otros estudiantes del mismo semestre o de semestres anteriores, se aplicará lo establecido por la reglamentación vigente, en particular el Artículo 75 del Reglamento de Régimen de Enseñanza y Aprendizaje.

Modificaciones de comentarios, cadenas alfanuméricas, nombres de variables, orden de estructuras independientes, y otras modificaciones menores de código se siguen considerando como clones de código, y las herramientas automatizadas de detección reportarán la similitud correspondiente.

Los estudiantes que provean a otros estudiantes del mismo o futuros semestres soluciones de sus tareas, también son sujetos a las sanciones especificadas en la reglamentación institucional. Por lo tanto, se advierte no poner a disposición soluciones de las tareas a otros estudiantes.