

## Tarea 4: Procesamiento en el dominio de la frecuencia

En el código base para la tarea 4 en [GitHub Classroom](#), la clase `frec_filter` realiza el procesamiento continuo en el dominio de la frecuencia utilizando la técnica de solapamiento y almacenamiento. En esta tarea usted implementará el método de solapamiento y suma.

Note que el script `ubuntu.sh` instala algunas bibliotecas adicionales a las anteriores, en particular, aquellas asociadas al cálculo de la FFT.

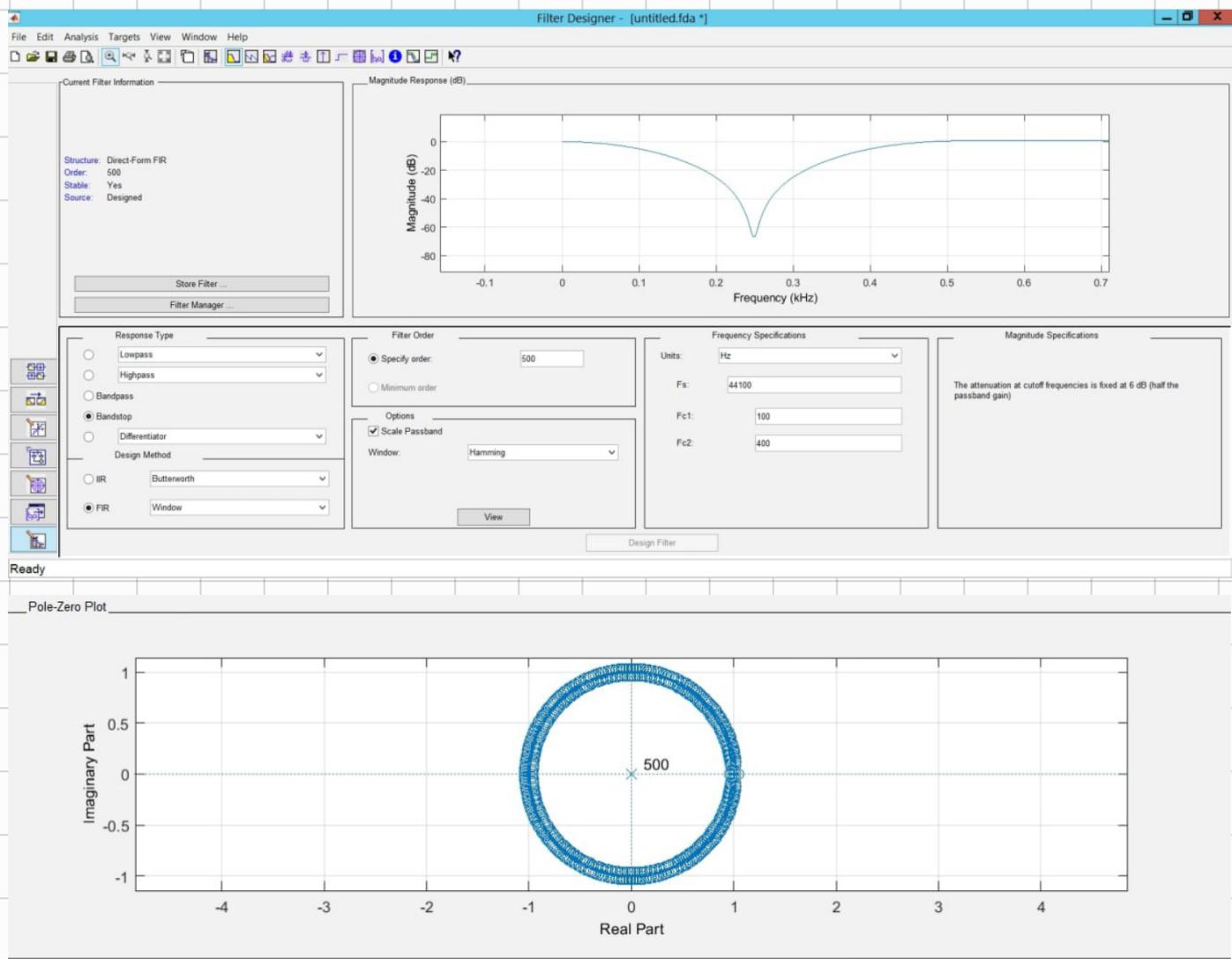
1. El código base utiliza filtrado en la frecuencia para construir el modo “*passthrough*”. Estudie el código para entender cómo se construye el filtro en el dominio de la frecuencia.
2. Programe un filtro que suprima un rango de frecuencias entre 100 Hz y 400 Hz. Debe tener los cuidados que correspondan para que la respuesta en frecuencia de dicho filtro sea real y causal.
3. Estudie la implementación del método `process` de la clase `frec_filter` y contrástela con el planteamiento teórico de la técnica de solapamiento y suma. Identifique cómo se realiza en el código cada paso de la técnica. Preste particular atención al uso de la biblioteca `fftw` para el cálculo de la FFT y la iFFT.
4. Usted puede crear una clase nueva, o simplemente agregar nuevos métodos a `frec_filter` para realizar el filtrado con la técnica de solapamiento y suma.
5. Integre a la funcionalidad ya creada para la tarea 3, la posibilidad de seleccionar tres filtros adicionales, pero que operan en el dominio de la frecuencia: con ‘S’ un filtro que suprime frecuencias entre 100 Hz y 400 Hz, con ‘A’ el filtro pasa-todo que ya fue entregado con el código y con ‘M’ un tercer filtro que realice algún procesamiento interesante de su elección.

Carlos Andrés Mata Calderón

2. Para este punto se usó filter Designer

Se usa un FIR Window de BandStop con una ventana tipo Hanning

Se usa el orden 500 ya que fue el valor mínimo que hizo que el filtro funcione.



Los coeficientes que daba se uso el .h que genera  
y se copio y pego el array que genero y se cambio  
el tipo float.

4). Se creo otra clase llamada

freq\_filter\_adder

que es una copia de freq\_filter  
pero se cambia 2 métodos

1. reset()

```
2 Codeium: Refactor | Explain | Generate Function Comment
3 void freq_filter_adder::reset() {
4     if (_Hw_size > 0) {
5         memset(_xn.get(), 0, sizeof(float) * _Hw_size);
6         std::fill(_yn.get() + _block_size, _yn.get() + _Hw_size, 0.0f); // zero padding
7     }
8 }
```

→ se le agrega un relleno de 0

## 2. process()

→ Se inicia copiando la entrada como viene

```
65     _processing = true;
66
67     // Overlap-add method
68
69     // the add-part first:
70     const std::size_t hn_size1 = (_hn_size - 1);
71
72     // copy the entire input block to _xn
73     std::copy(in, in + _block_size, _xn.get());
74
75     // padding the remaining part with zeros
76     std::fill(_xn.get() + _block_size, _xn.get() + _Hw_size, 0.0f);
77
78         for (std::size_t i = 0; i < _hn_size - 1; ++i)
79     {
```

y se rellena con 0

: se hace exactamente igual el calculo  
de  $y[n]$  nuevo con la FFT y la IFFT

```
291
292     // overlap-add: sum the last _hn_size-1 samples from previous output
293     for (std::size_t i = 0; i < hn_size1; ++i) {
294         _yn.get()[i] += _yn.get()[_block_size + i];
295     }
296
297     // move the data to the output array
298     std::copy(_yn.get(), _yn.get() + _block_size, out);
299
300     _processing = false;
301 }
```

→ Se suman con los ultimos  $M-1$  de  $y[n]$   
anterior con los nuevos  $y[n]$

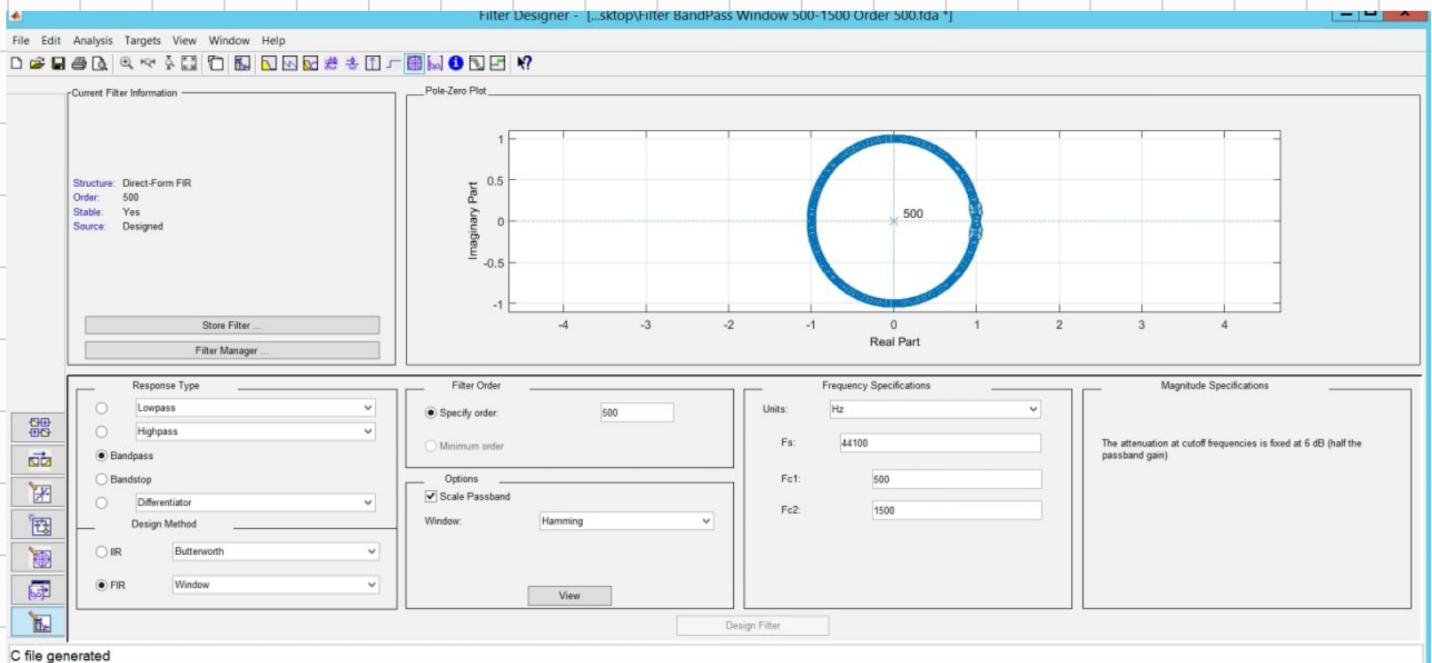
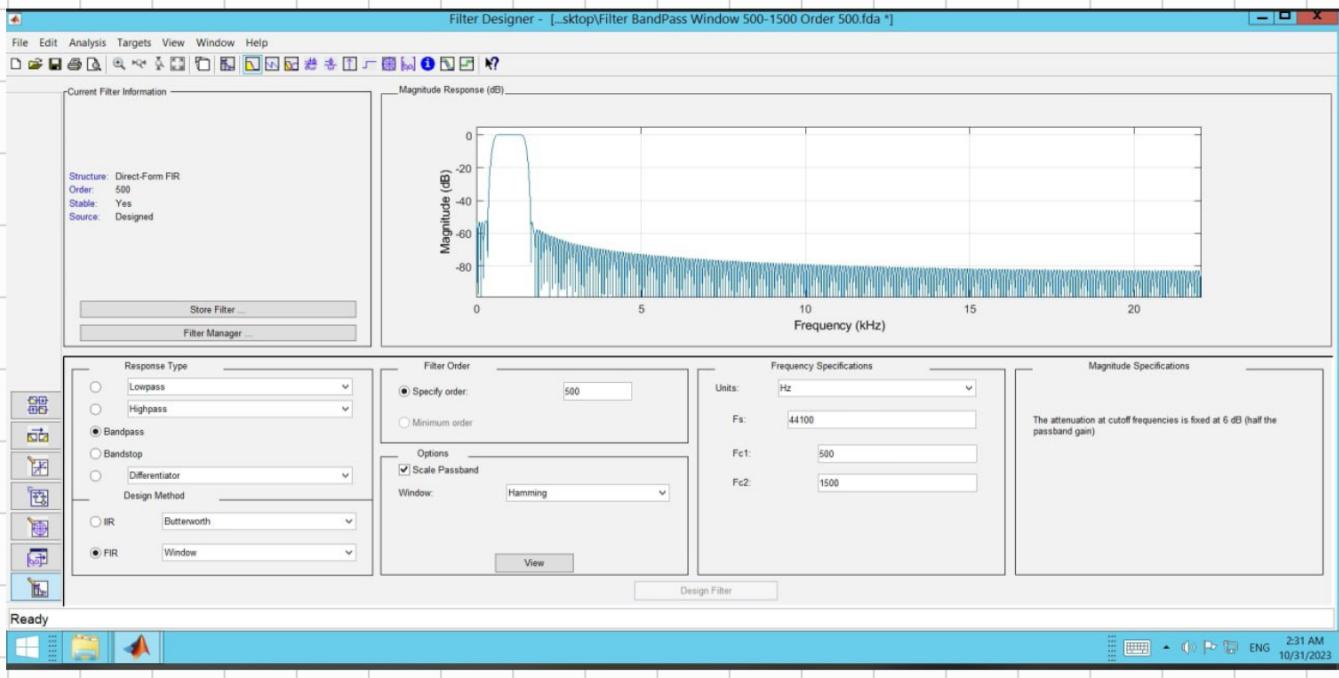
y eso es todo. ~~Es~~ es la diferencia con la

Overlap-Save

## 5. Se crea un filtro que genera una

Voz como Robot se usa un Band Pass de 500 a 1500 Hz

Se usó igual filter Designer



C file generated

Igual se uso 500 de orden con el fin que  
fueran un buen filtro.

Ademas se adjunta los filtros de  
la tarea 3. Revisar el diseño  
de esos filtros hechos a mano y  
sin FFT.