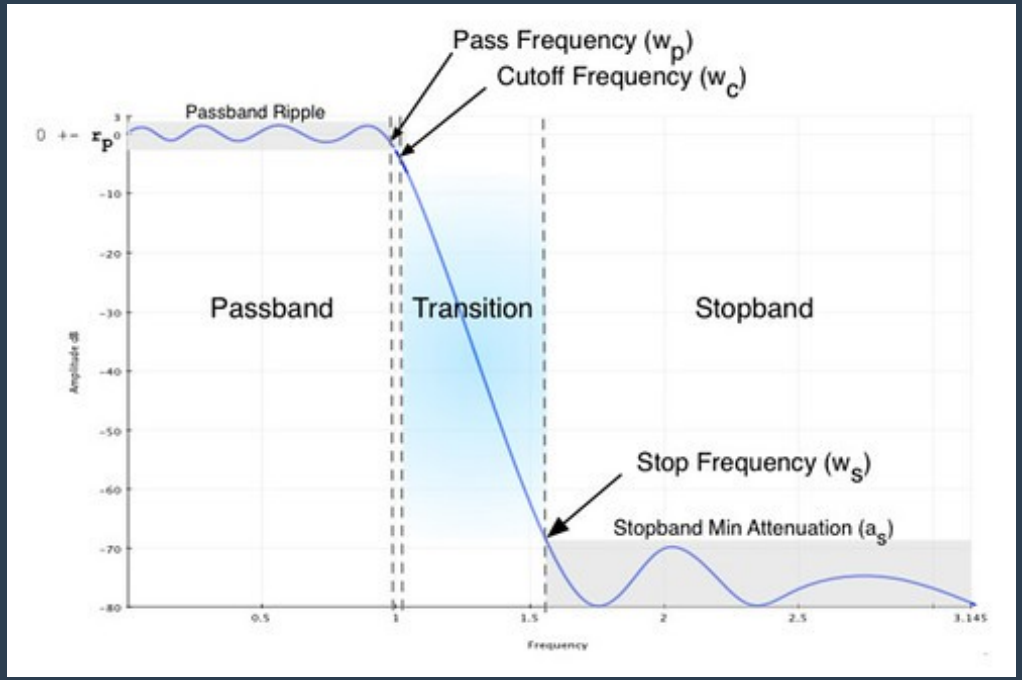
Procesamiento de señales. Fundamentos

Clase 6 – Filtrado | FIR

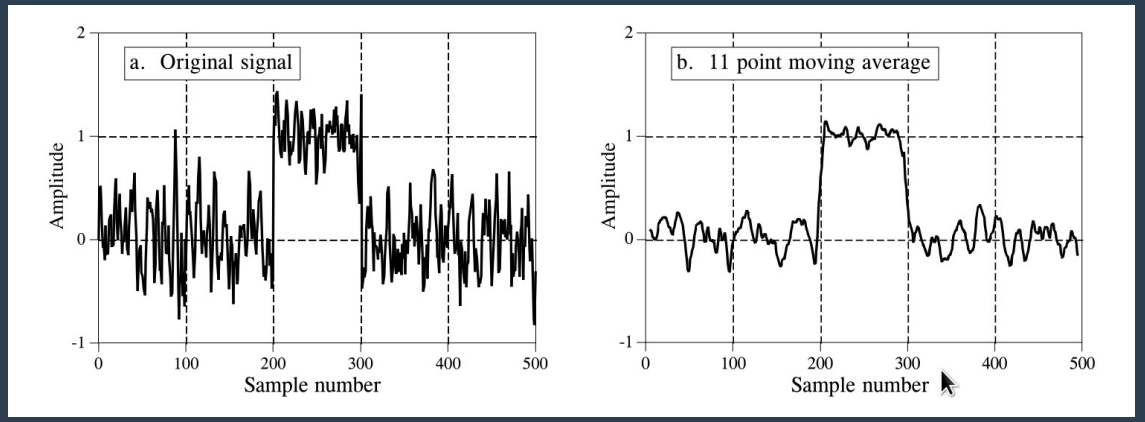
- Filtrado
  - Clasificación
  - Objetivos
  - Moving Average
  - Sinc
  - Ventanas
- Tecnica de Overlapp and Add
- Promedio en espectro
- Pyfdax
- Filtrado en CIAA
  - Por convolución
  - Por multiplicación en f
- Pyfdax-> CIAA header

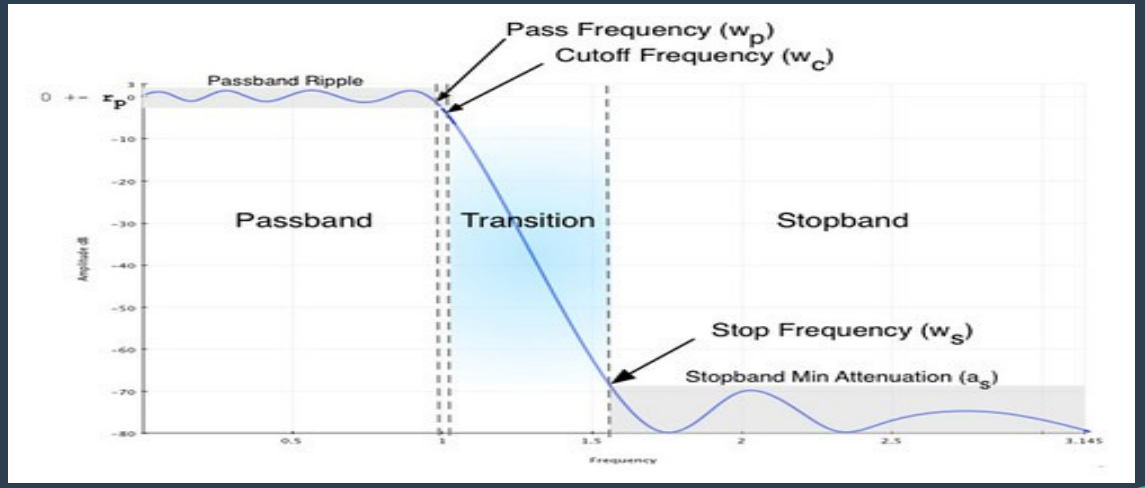


Filtrado

Filtrado | Clasificación | Objetivos

- Se puede clasificar el filtrado en dos grandes grupos:
  - Filtrar en frecuencia para separar
  - Filtrar en tiempo para restaurar
- Los filtros se pueden representar con la respuesta al impulso, al escalon y/o respuesta en frecuencia. (los tres dependientes uno de otro)
- Cuando se usa la convolucion y la respuesta al impulso del filtro para filtrar en tiempo se habla del kernel del filtro
- Los filtros digitales logran una performance cientos de veces superior a los analogicos
- Los filtros FIR son sistemas cuya respuesta al impulso tiene una cantidad de puntos finita.
- Los filtros FIR calculan la salida del sistema utilizando la convolución en tiempo

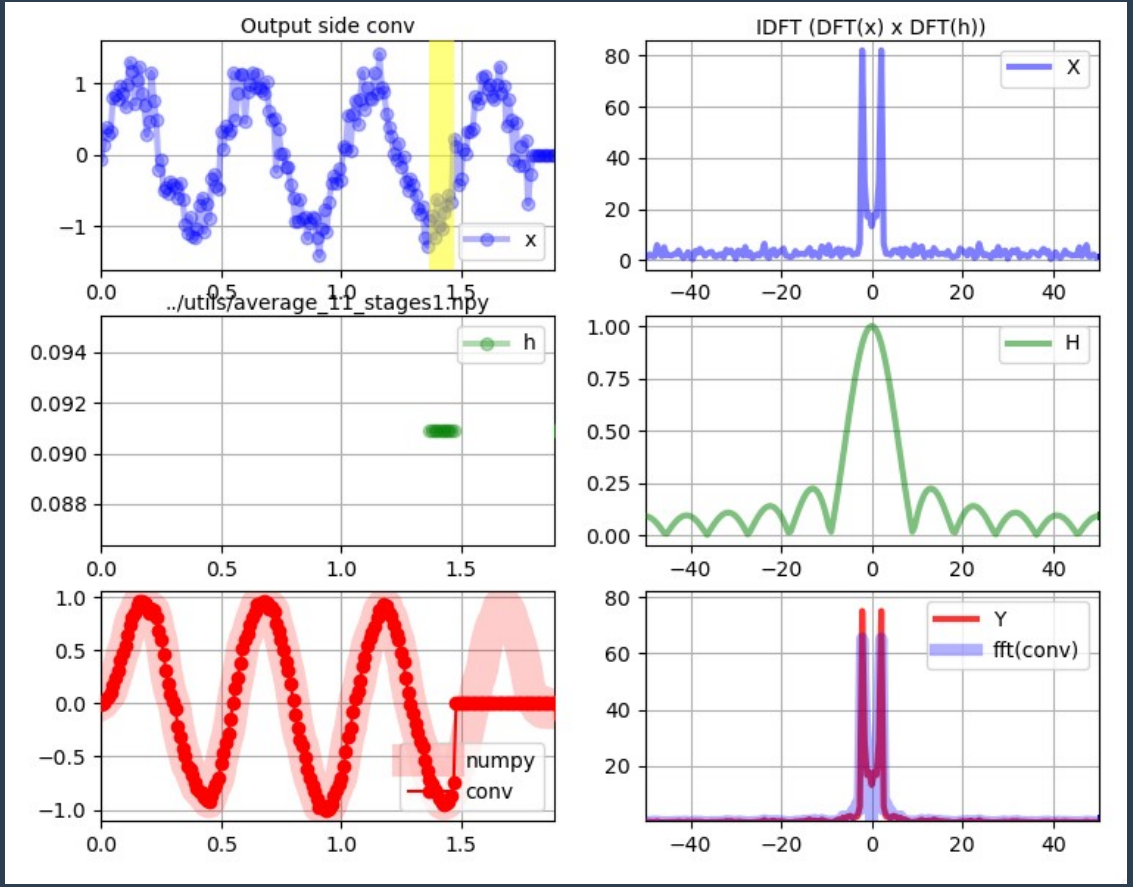




FIR – Moving average

FIR | Moving average | etapa simple

- Es uno de los filtros mas simples pero aun así con gran espectro de aplicación
- Principalmente para eliminar ruido blanco.
- Es simplemente el promedio de los puntos vecinos al punto de análisis
- El kernel entonces es un escalón de una altura igual a 1/M donde M es el numero de vecinos a promediar.



- Ver código: fir.py

FIR | Moving average | Recursiva

- Se acelera mucho el algoritmo si en vez de hacer la convolucion se utiliza el ultimo valor calculado para calcular el siguiente. Implementacion recursiva.

$$y[50] = x[47] + x[48] + x[49] + x[50] + x[51] + x[52] + x[53]$$

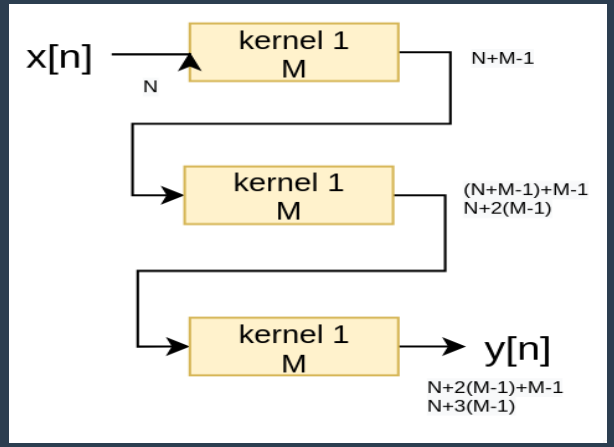
$$y[51] = x[48] + x[49] + x[50] + x[51] + x[52] + x[53] + x[54]$$

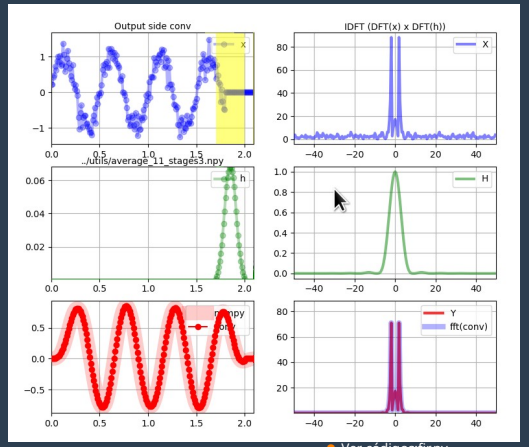
$$y[51] = y[50] + x[54] - x[47]$$

- Ver código: fir.py

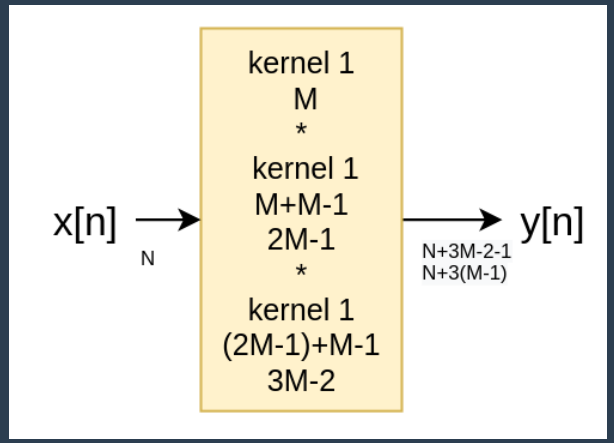
FIR | Moving average | multiples etapas

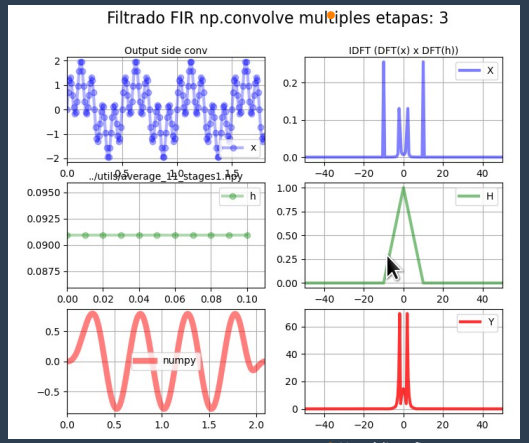
- Se puede mejorar mucho la performance pasando la señal repetidas veces por el mismo kernel
- Por la propiedad distributiva, se puede hacer un kernel que pasa sobre si mismo n veces y es equivalente
- Si aumentamos cada vez mas etapas, por el teorema central del limite, el kernel tiende a una gaussiana
- Se paga el precio de tener que procesar mas puntos





Ver códigos:fir.py



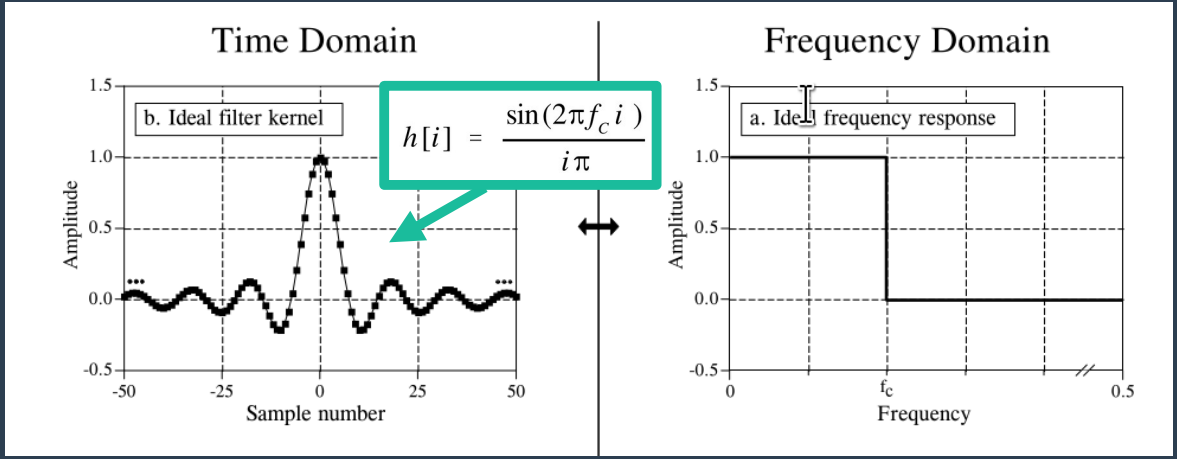


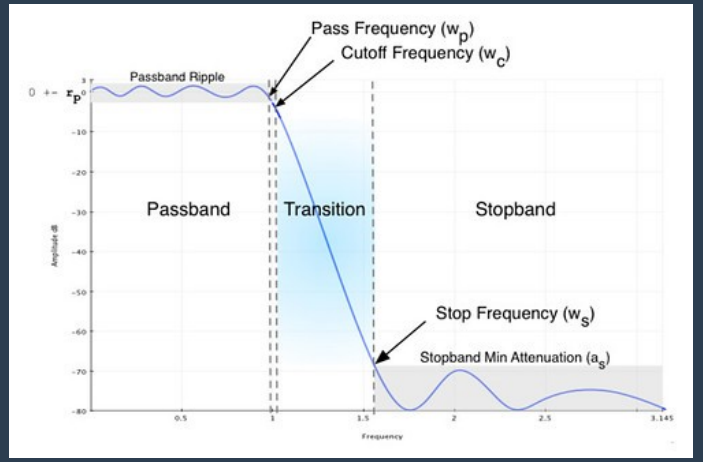
Ver códigos:fir_numpy.py

FIR - SINC

Discriminacion en frecuencia | Sinc

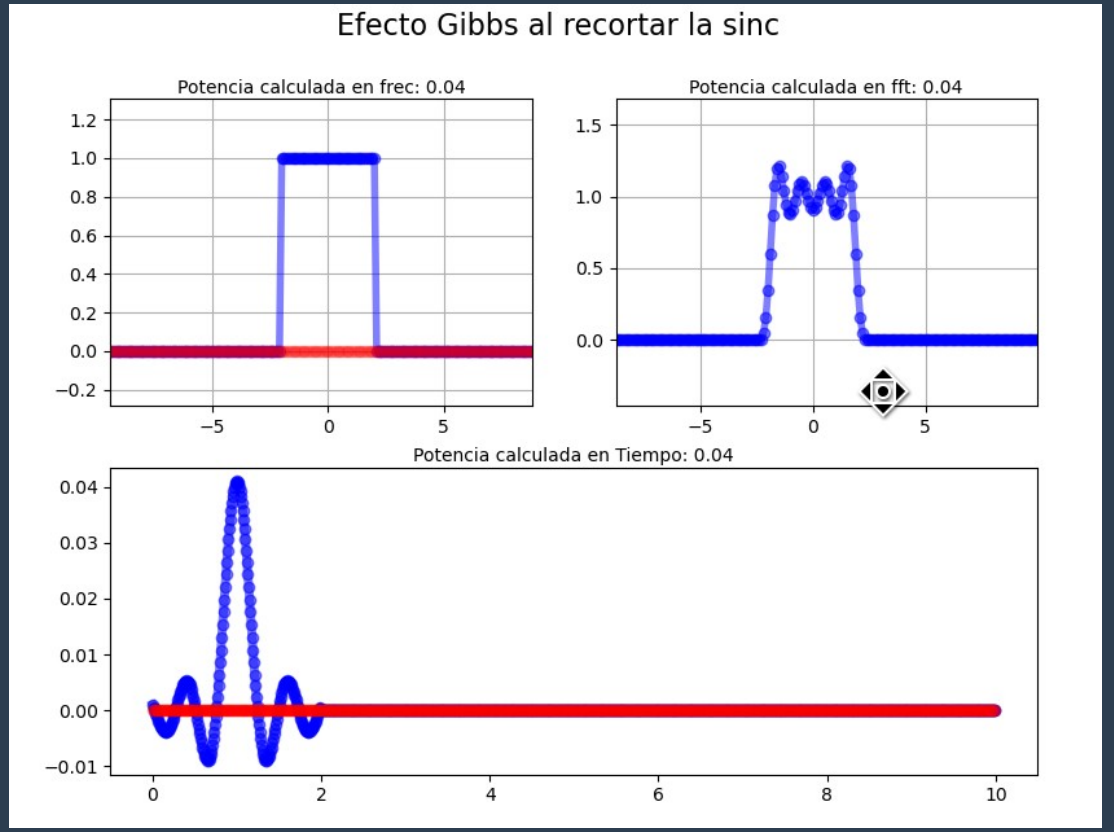
- Cuando lo que se busca es discriminar en frecuencia, el filtro ideal es un escalón
- La IDFT de un escalon, como se ve en la figura, es una sinc de longitud infinita => no es realizable.





Sinc truncada

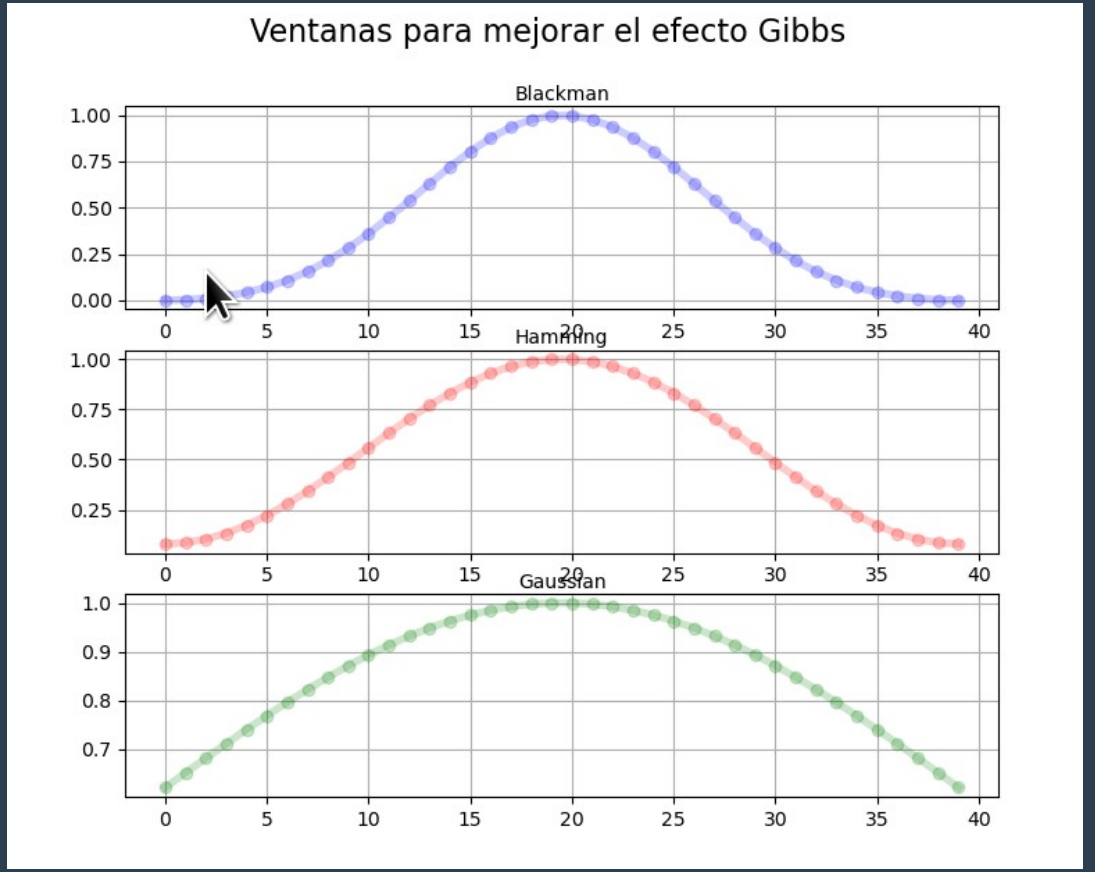
- Para poder utilizar en un sistema digital realizable una Sinc, hay que truncar hasta cierto punto
- El salto en los extremos de la sinc, genera ripple en la reconstrucción del escalón conocido como effecto Gibbs.
- Es un efecto poco deseable en un filtro



- Ver codigo: sinc.py

Ventanas

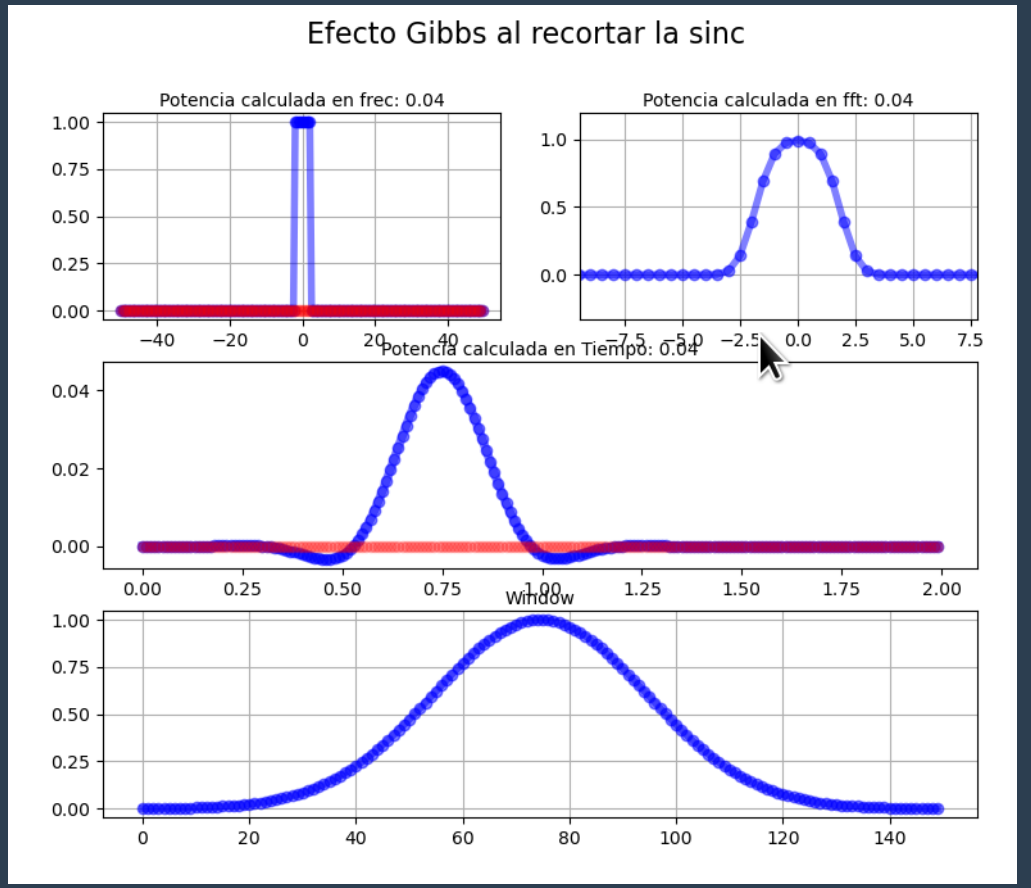
- Para mitigar el effecto Gibbs se multiplica la sinc por otra función ventana que suaviza los extremos.
- Hay diferentes funciones de ventanas como se ve en la figura.
- Cada una presenta diferentes propiedades y características Se deberá seleccionar la mas adecuada para cada aplicación.



- Ver codigo: windows.py

Sinc truncada y ponderada con ventana

- Se multiplica la ventana por la sinc truncada y se obtiene una mejora en el ripple de la respuesta en frecuencia del filtro como se ve en la figura
- Probar con varias ventanas y diferentes opciones de corte de la sinc para asimilar los resultados

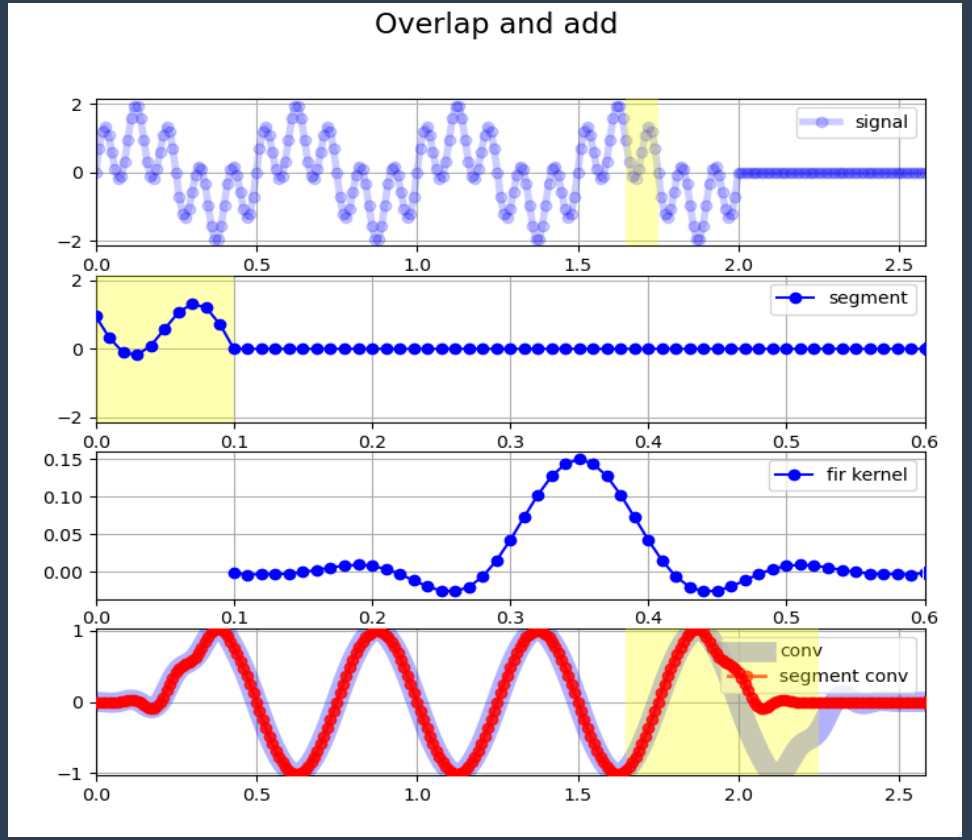


- Ver codigo: sinc_windowed.py

Overlap and add

Overlap and add | Convolucion en t

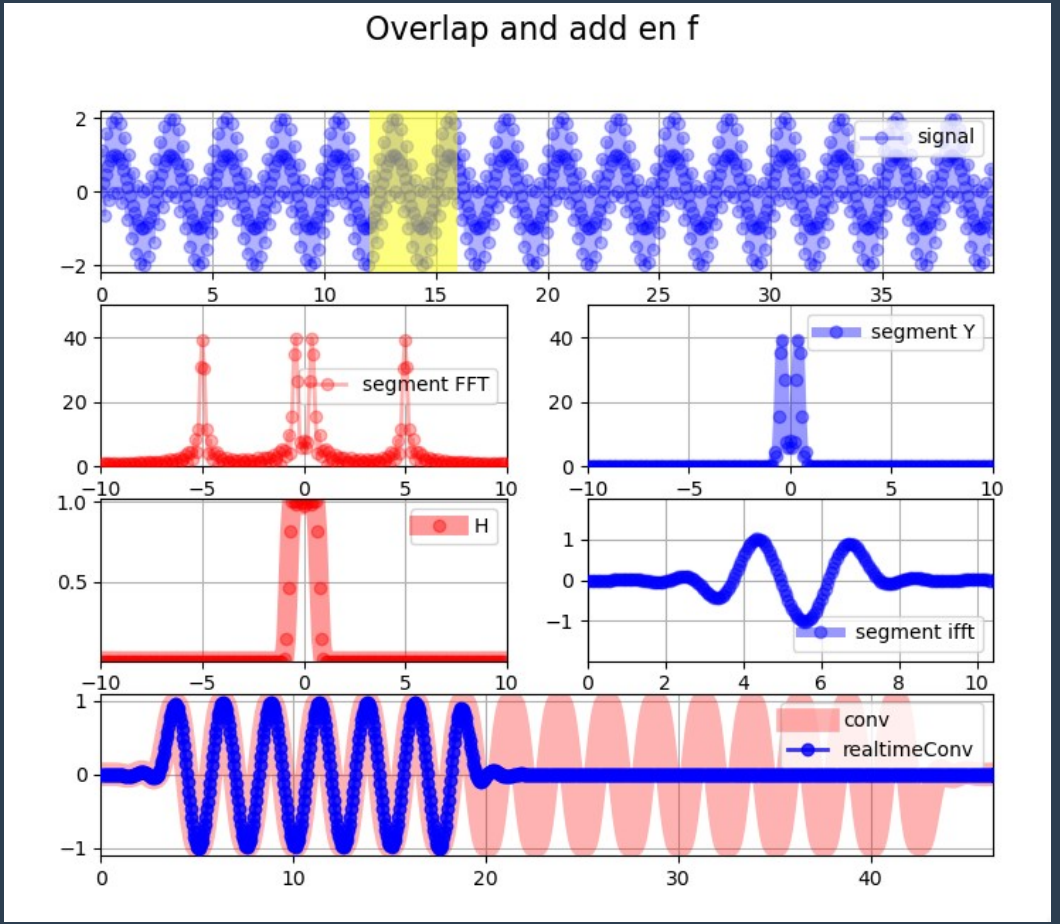
- Que sucede cuando la señal tiene muchos puntos y se desea realizar el filtrado?
- Si se espera tener todos los puntos para convolucionar el sistema podria responder muy lento y no siempre es posible utilizar la técnica de la convolución por FFT con un numero muy grande de puntos. Por ej. en la CMSIS el máximo es de 4096
- Si cortamos en trozos la señal y se opera por tramos , la primera y ultima parte de la señal resultante se descartan por el propio efecto de la convolución Con lo cual se pierde la continuidad.
- Una técnica para estos casos es partir la señal en trozos, pero solapar las salidas de cada operación de modo que si salen N+M-1 puntos en la primera convolución, la segunda arranca desde el punto N-1, solapando M puntos y sumando el resultado de la nueva operación en esa zona como se ve en la imagen.



- Ver códigos:
  - overlap_and_add_t.py
  - overlap_and_add_t_explain.py

Overlap and add | convolución con FFT

- Se puede hacer lo mismo pero en vez de convolucionar en tiempo, se utiliza el teorema de la convolución:
  - Se calcula la FFT del segmento.
  - Se calcula la FFT de la respuesta al impulso unitario (una única vez)
  - Se multimplican ambas.
  - Se toma la IFFT del resultado y se solapa en tiempo.
- Todo el proceso es mucho mas eficiente computacionalmente que la convolución por el hecho de que el orden del algoritmo FFT es de O(log2) mientras que el de la convolucion es de O(2)

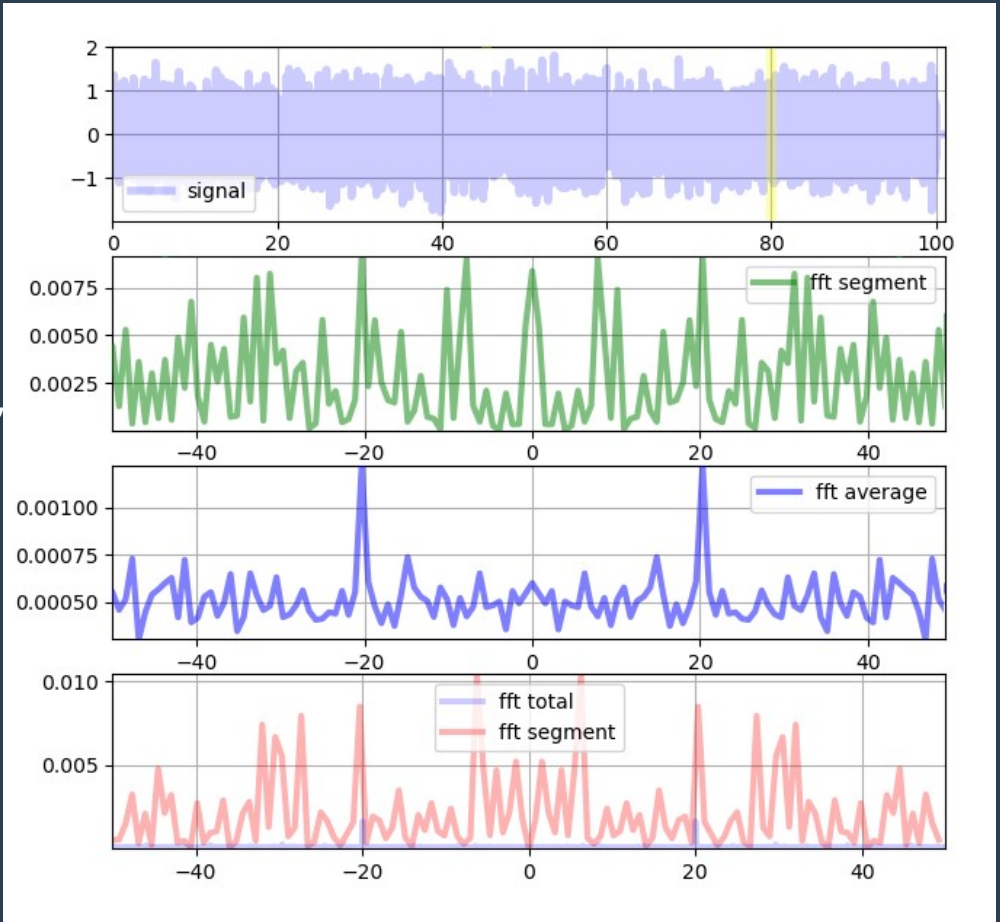


- Ver código: overlap_and_add_f.py

Promedio en el espectro

Promedio en el espectro

- Que sucede cuando la señal es muy larga y se requiere separar el contenido de información sumergida en ruido?
  - Se podría esperar a tener todos los puntos y tomar la fft de toda la señal para discriminarla. Incluso se podría aplicar un filtro pasabajos a la fft para eliminar parte del ruido.
  - El problema con dicha técnica es que en la medida que agregamos mas puntos a la fft, dividimos por N los datos, ruido y señal.
  - Por otro lado es muy costoso computacionalmente sino imposible hacer FFT's muy largas
  - Se pierde la posibilidad de análisis en tiempo real
- En cambio se podría trozar los datos en segmentos de resolución aceptable según lo que se busque y tomar el promedio de todos los espectros. De esta manera se resalta la señal de interés mientras que se diluye el ruido al tiempo que se puede usar una FFT mas corta y mantener el tiempo real, como se ve en la imagen

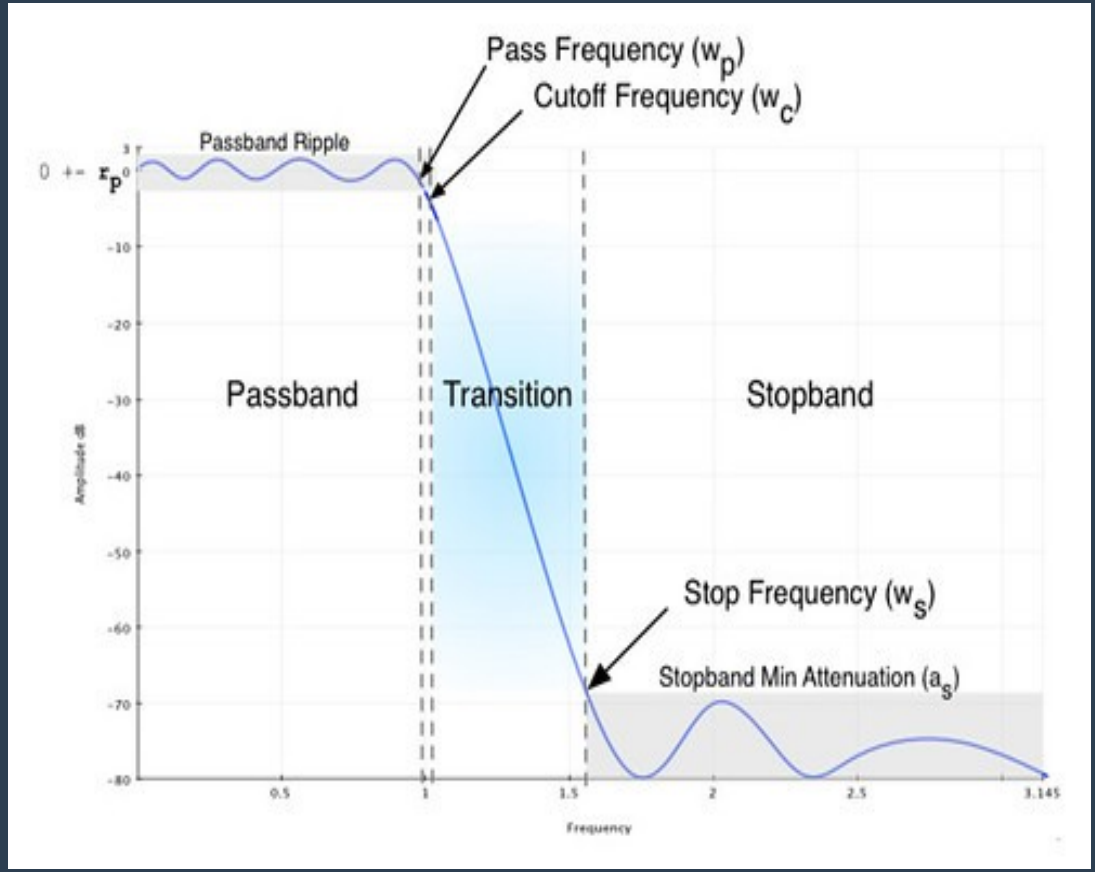


- Ver código: spectrum_average.py

Diseno de FIR con pyfdax

Diseño de FIR plantilla

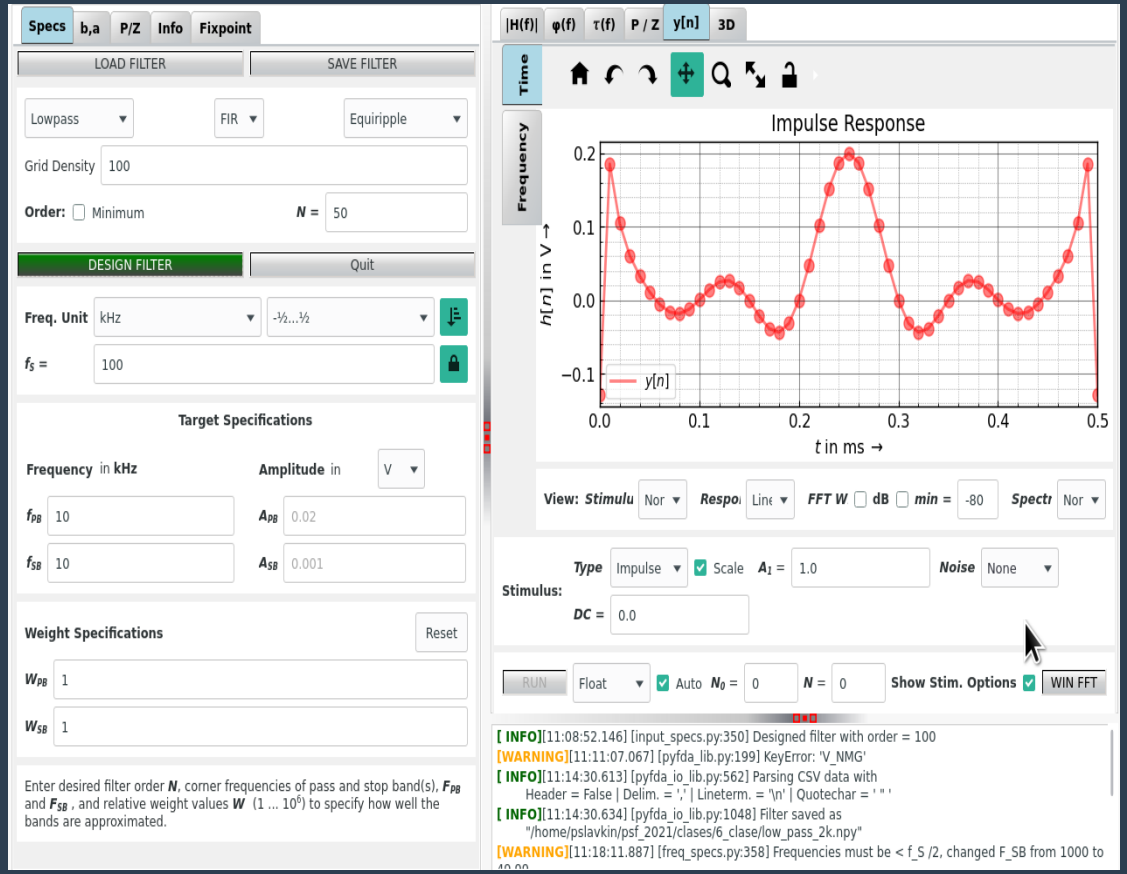
- Antes de utilizar la herramienta se recomienda especificar los requerimientos mas holgados que resuelven el sistema.
- Esto permitirá minimizar la cantidad de puntos para un determinado filtro impactando en la potencia de computo requerida



- Ver código: teorema_convolucion.py

Diseño de FIR con pyfdax

- Se instala con pip install pyfdax
- https://github.com/chipmuenk/pyfda
- Se puede diseñar FIR con o sin ventana
- Moving average de N etapas
- Se pueden extraer directamente los valores de los coeficientes en la solapa b/a
- Instalar y probar todas sus funcionalidades



- Ver código: teorema_convolucion.py

Filtrado con la CIAA

Pasaje de Pyfdax a header CIAA

- Se comparte un script de python para pasar los datos extraídos de pyfdax en formato binario .npy a un header de C para usarlo en la CIAA.
- Rellena con ceros.
- Indicar el N teniendo en cuanta que sea potencia de 2 para que se pueda implementar en la CIAA.
- Incluir el header en el código
- Se generan los vectores:
  - h[N+M-1] valores reales.
  - H[2*(N+M-1)] valores reales intercalados real,imag.
  - HAbs[N+M-1] valores reales.

```
#define h_LENGTH 128
#define h_PADD_LENGTH 383
#define H_PADD_LENGTH 383
q15_t h[]={
11,
15,
17,
18,
18,
17,
15,
9,
```

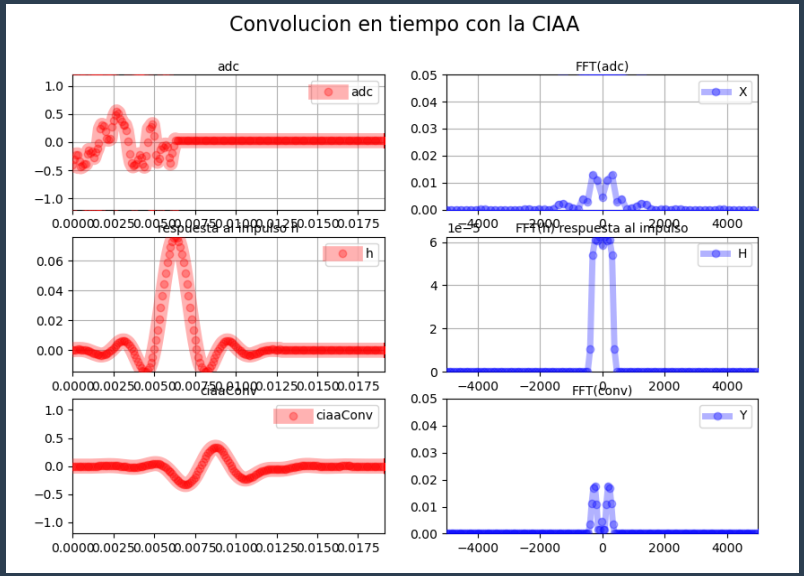
```
HAxe.set_xlim(-fs/2,fs/2)
#------------------------------------
def convertToC(h,H,fileName):
    cFile  = open(fileName,"w+")
    cFile.write("#define h_LENGTH {}\n".format(len(firData)))
    cFile.write("#define h_PADD_LENGTH {}\n".format(len(h)))
    cFile.write("#define H_PADD_LENGTH {}\n".format(len(H)))
    h*=2**15
    h=h.astype(np.int16)
    H*=2**15
    cFile.write("q15_t h[]={\n")
    for i in h:
        cFile.write("{},\n".format(i))
    cFile.write("};\n")
    cFile.write("q15_t H[]={\n")
    for i in H:
        cFile.write("{},{},\n".format(np.real(i).astype(np.int16),np.imag(i).astyp
    cFile.write("};\n")

convertToC(firExtendedData,HData,"fir.h")
plt.get_current_fig_manager().window.showMaximized()
plt.show()
```

- Ver código utils/fir_to_c.py

Filtrado CIAA | Convolución en t

- El filtrado en la CIAA se puede realizar convolucionando la señal con la respuesta al impulso del filtro
- Se mantiene información de fase y no se distorsiona la señal en el tiempo
- Se puede aplicar la técnica de convolucionar utilizando FFT gracias al teorema de la convolución multiplicando espectros de la señal con el espectro del kernel

```
adc[sample]       = (((int16_t )adcRead(CH1)-512)>>(10-BITS))<<(6+10-BITS);          // PISA el
//float t=((tick%(sweept*header.fs))/(float)header.fs);
//tick++;
//dacWrite( DAC, 512*arm_sin_f32 (t*B/2*(t/sweept)*2*PI)+512); // sweept
//dacWrite( DAC, 512*arm_sin_f32 (t*tone*2*PI)+512);       // tono
if ( ++sample==header.N ) {
   gpioToggle ( LEDR );                          // este led blinkea a fs/N
   sample = 0;

----------CONVOLUCION-----------------
   arm_conv_q15       ( adc,header.N,h,h_LENGTH,y);
   arm_conv_fast_q15  ( adc,header.N,h,h_LENGTH,y);

----------ENVIO DE TRAMA-----------------
   header.id++;
   uartWriteByteArray ( UART_USB ,(uint8_t*)&header ,sizeof(struct header_struct ));
   for (int i=0;i<(header.N+h_LENGTH-1 );i++) {
      uartWriteByteArray ( UART_USB ,(uint8_t* )(i<header.N?&adc[i]:&offset ),sizeof(adc[0]));
      uartWriteByteArray ( UART_USB ,(uint8_t* )(i<h_LENGTH?&h   [i]:&zero   ),sizeof(h[0])   );
      uartWriteByteArray ( UART_USB ,(uint8_t* )(           &y   [i]         ),sizeof(y[0])   );
   }
   adcRead(CH1); //why?? hay algun efecto minimo en el 1er sample.. puede ser por el blinkeo de
}
gpioToggle ( LED1 );                                          // este led blinkea a fs/2
while(cyclesCounterRead()< EDU_CIAA_NXP_CLOCK_SPEED/header.fs) // el clk de la CIAA es 204000006
   ;
```

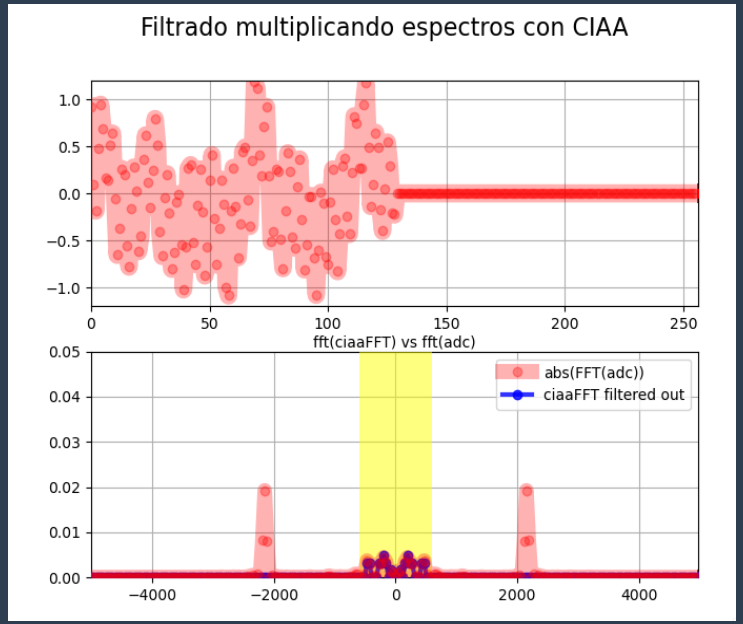


- Ver carpeta clase6/psf1

Filtrado CIAA | Multiplicación en f

- El filtrado en la CIAA se puede realizar multiplicando espectros de la señal con el espectro del kernel del filtro
- Se pierde información de fase, pero lo importante del método es separar en frecuencia. Para mantener la calidad de la señal en el tiempo conviene la convolución en t o la convolución en t utilizando FFT que mantiene la información de fase
- Atención con las multiplicaciones y exponenciaciones del q15 !

```
    if ( ++sample>=header.N ) {
       for(;sample<(header.N+h_LENGTH-1);sample++){ //relleno con ceros M-1 puntos
          adc[sample]          = 0;
          fftInOut[sample*2]   = 0;
          fftInOut[sample*2+1] = 0;
       }
       gpioToggle ( LEDR );                        // este led blinkea a fs/N

------------TRANSFORMADA-----------------
       init_cfft_instance ( &CS,(header.N+h_LENGTH-1 ));
       arm_cfft_q15       ( &CS ,fftInOut ,0 ,1      ) ;

------------MAGNITUD-----------------
       arm_cmplx_mag_squared_q15 ( fftInOut ,fftAbs ,(header.N+h_LENGTH-1 ));

------------FILTRADO MULTIPLICANDO ESPECTROS-----------------
       arm_mult_q15 ( fftAbs,HAbs ,fftAbs ,(header.N+h_LENGTH-1));

       header.id++;
       uartWriteByteArray ( UART_USB ,(uint8_t*)&header ,sizeof(struct header_struct ));

       for (sample=0; sample<(header.N+h_LENGTH-1);sample++ ) {
          uartWriteByteArray ( UART_USB ,(uint8_t* )&adc[sample]     ,sizeof(adc[0]) );
          uartWriteByteArray ( UART_USB ,(uint8_t* )&fftAbs[sample*1] ,sizeof(fftInOut[0]));
       }
       sample = 0;
```



- Ver carpeta clase6/psf2