Usando datos del ADC y el DAC

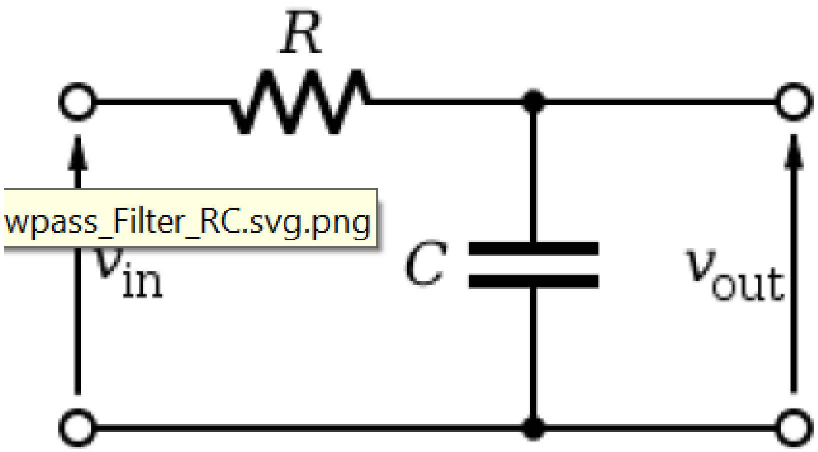
Agenda

- Polo simple versión discreta
- Promediar muestras
- Suma de convolución.

- Filtros FIR.
  - Herramientas.
  - Implementación
  - Consideraciones numéricas

- Filtros IIR
  - Herramientas
  - Implementación
  - Cuestiones numéricas
  - Secciones de segundo orden (SOS)

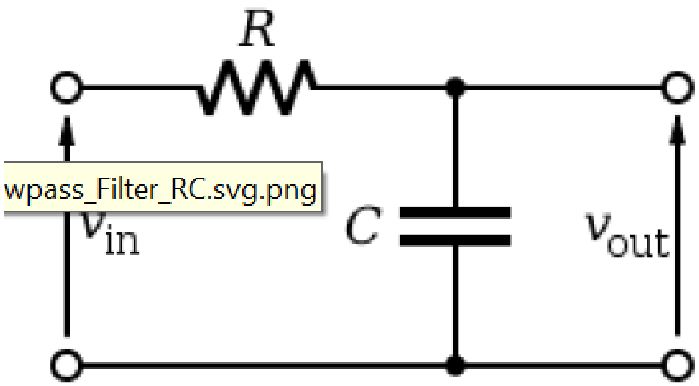
Polo simple



- $\frac{V_{O(s)}}{V_{I(s)}} = \frac{{}^{1}/_{RC}}{s+{}^{1}/_{RC}}$
- $\left|H_{(\omega)}\right| = \frac{1}{\sqrt{1+(\omega RC)^2}}$
- $\phi_{(\omega)=-arctan(\omega RC)}$
- $\tau_g = -\frac{d\phi_{(\omega)}}{d\omega} = \frac{RC}{1+(\omega RC)^2}$

- El filtro pasabajos más sencillo que podemos encontrar es el de un único polo.
- Este filtro presenta una atenuación de 20dB/década.
- Cómo primera aproximación al procesamiento digital de señales vamos a implementar este mismo filtro de manera digital.
- Para su implementación vamos a partir de sus ecuaciones temporales

Polo simple (2)



- $v_{IN(t)} - Ri_{(t)} - v_{OUT(t)} = 0$
- $i_{(t)} = C\frac{dv_{OUT(t)}}{dt}$
- $\boldsymbol{v_{IN(t)} - v_{OUT(t)} = RC\frac{dv_{OUT(t)}}{dt}}$

- Sí discretizamos y consideramos que:
  - $v_{IN(t)} \rightarrow \quad x[n]$
  - $v_{OUT(t)} \rightarrow y[n]$
  - T_s: *período de muestreo*

- Entonces:
  - $\boldsymbol{x[n] - y[n] = RC\frac{y[n]-y[n-1]}{Ts}}$

Polo simple (3)

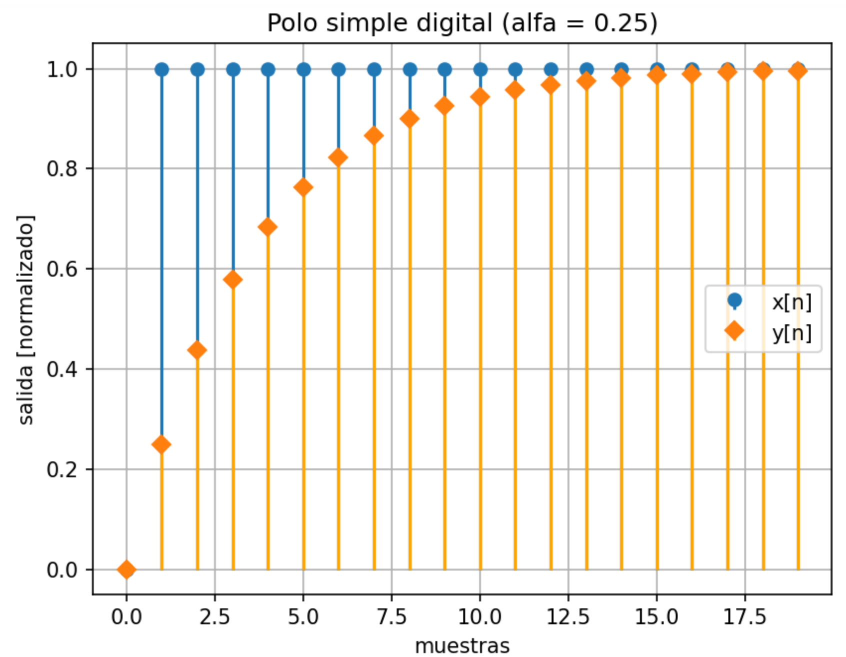
- Reordenando:
  - $x[n] - y[n] = RC\,\frac{y[n]-y[n-1]}{Ts}$
  - $y[n] = \frac{x[n]+\frac{RC}{T_s}y[n-1]}{1+\frac{RC}{T_s}}$
  - $\alpha = \frac{T_s}{RC+T_s} = \frac{1}{1+\frac{RC}{T_s}} \Longrightarrow 1-\alpha = \frac{\frac{RC}{T_s}}{1+\frac{RC}{T_s}}$
  - $0 \leq \alpha < 1$
  - $para\ RC\ y\ T_s\ positivos\ y\ reales$
  - $y[n] = \alpha x[n] + (1-\alpha)y[n-1]$

- α determina cual es el aporte a la señal a generar del valor de la entrada actual y de los valores pasados de la salida.
- Valores de α más cercanos a 1 hacen que la entrada actual (x[n]) tenga más peso que las salidas anteriores (y[n-1])
- Valores de α más cercanos a cero harán que el mayor peso sea de y[n-1]

Implementando el polo simple

- $y[n] = \alpha x[n] + (1 - \alpha) y[n-1]$
- Donde la implementación en punto flotante es directa y solo implica una suma y dos productos.

```
float polo_simple(float x, float alfa)
{
    static float yp = 0.0;
    return (yp = alfa*x + (1.0-alfa)*yp);
}
```

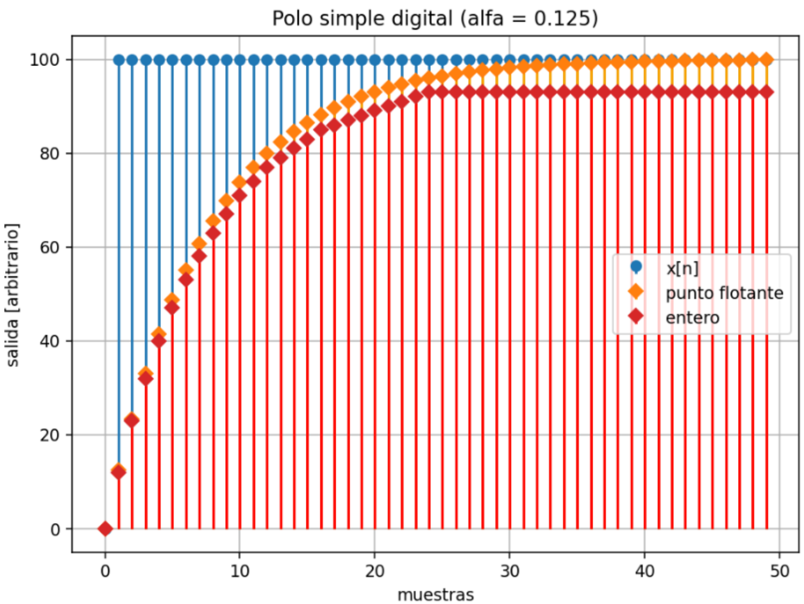


- El polo simple digital, requiere de un único estado anterior, pero esta implementación, así como está, necesita usar punto flotante.

Implementando el polo simple(2)

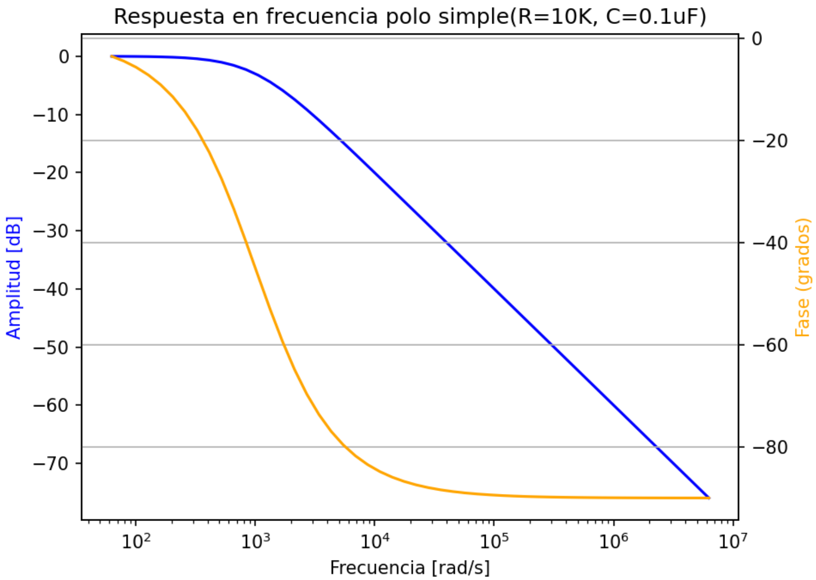
- $y[n] = \alpha x[n] + (1-\alpha)y[n-1]$
- $definimos: m = {}^{1}/_{\alpha}$
- $y[n] = y[n-1] + \frac{x[n]-y[n-1]}{m}$
- $sí\ ademas\ m = 2^k$
- $y[n] = y[n-1] + \frac{x[n]-y[n-1]}{2^k}$

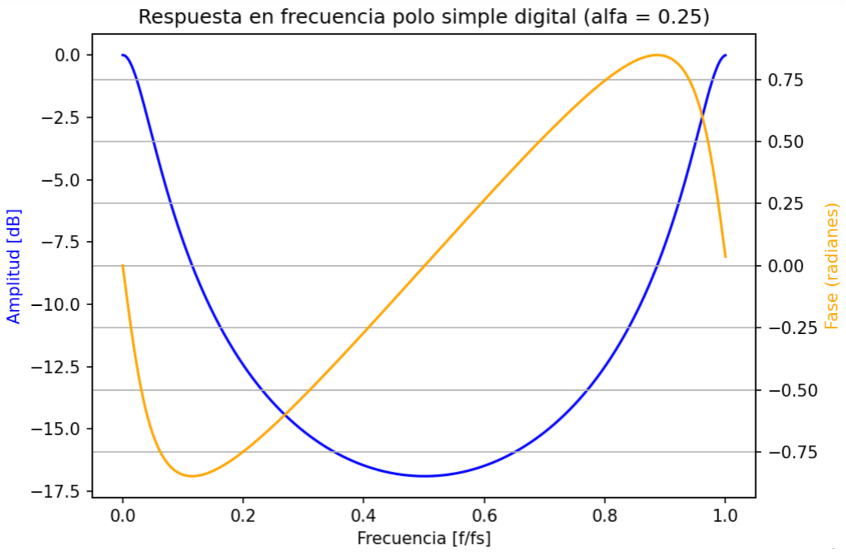
```
int polo_simple_entero(int x, int k)
{
    static int yp = 0;
    return (yp += (x-yp)>>k);
}
```



- El polo simple digital entero requiere dos sumas y un desplazamiento.
- Se genera error en su salida por acumular error por truncar en y[n-1]
- https://www.onlinegdb.com/HJbGns2bD

Polo simple, respuesta en frecuencia





- Hasta dónde es válida la respuesta en frecuencia del polo simple digital ¿Por qué?.

Polo simple digital. Resumen

- Es un filtro simple, asociado a el filtro analógico más simple.☺
- Consume pocos recursos del procesador (dos sumas y un desplazamiento).☺
- Acumula error, por lo que se necesita que el valor de y[n-1] tenga muchos más bits que el valor de y[n].☹
- El polo simple digital “no se lleva” con los enteros, si con el punto flotante.😐
- Veremos otra manera de filtrar ruido.

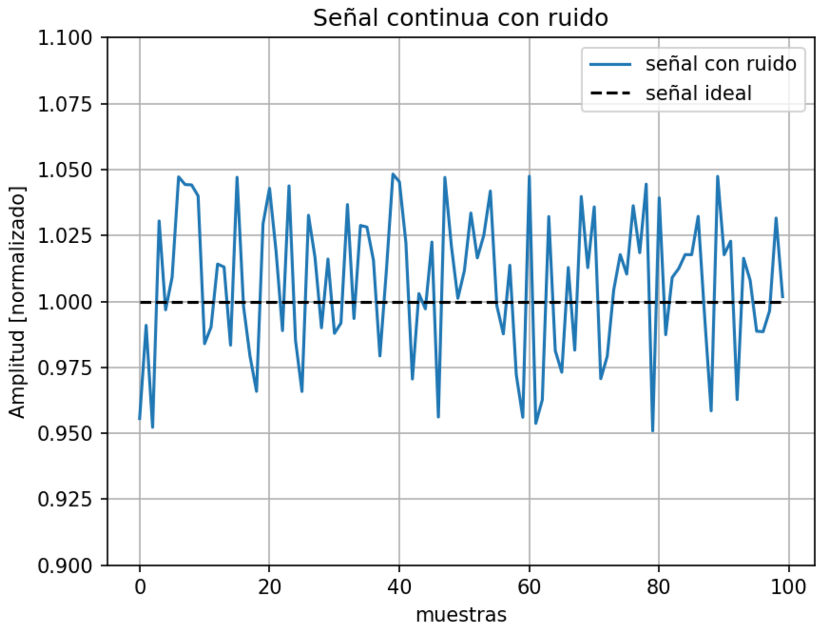
Otra aproximación.

- Consideremos una señal estática o de continua la cual se encuentra afectada por ruido sin continua y con el mismo aporte de potencia en todas las frecuencias (ruido blanco).

- Podemos decir que esta señal compuesta por el ruido y un valor de continua va a estar caracterizada por:
  - Valor medio (μ). Va a representar a la continua.
  - Desvío (σ). Va a representar al ruido que afecta a la señal.

- Para esta aproximación vamos a considerar a cada muestra que tomemos como una ***variable aletoria***.

- Por lo que consideraremos al conjunto de muestras como un conjunto de variables aleatorias independientes con la misma distribución.

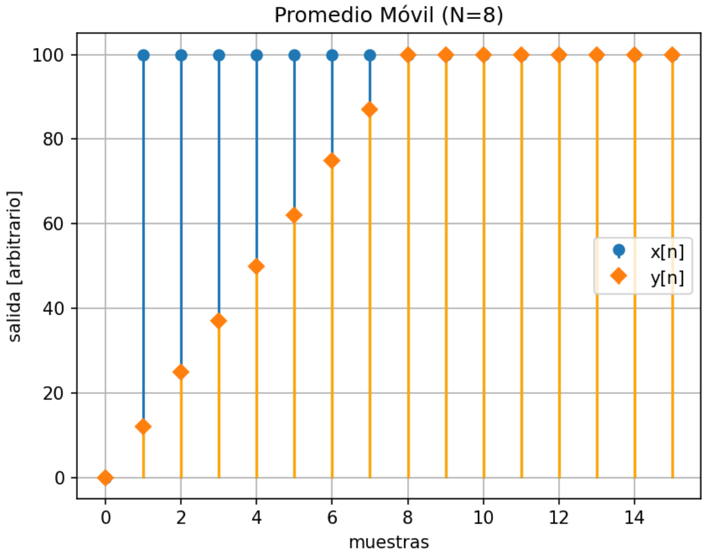


Teorema del límite central

- El teorema del límite central nos dice que sea X_1, X_2, X_n un conjunto de variables aleatorias, independientes e idénticamente distribuidas de una distribución con media μ y varianza $\sigma^2 \neq 0$. Entonces si n es lo suficientemente grande (~30), la variable aletoria:
  - $\bar{X} = \sum_{i=1}^{n} X_i$
  - Tiene:
    - $\mu_{\bar{X}} = \mu$
    - $\sigma_{\bar{X}}^2 = \frac{\sigma^2}{n}$
- Cómo el ruido de nuestra señal va a estar asociado al desvío ***si promediamos n muestras el desvío de la señal promediada será $\sqrt{n}$ veces menor que de la señal original, minimizando la señal de ruido.***

Promedio móvil

- $y[n] = \frac{1}{M}\sum_{i=0}^{M-1} x[n-i]$
- A diferencia del polo simple, las salidas anteriores no son necesarias.
- Son necesarias M sumas y una división para implementar el filtro.




```
void inicializar_promedio_movil(int *buffer, int *ix, int len)
{
	int i;
	*ix=0;
	for(i=0;i<len;i++)buffer[i]=0;
}

int promedio_movil(int x, int *buffer, int *ix, int len)
{
	int acc=0;
	buffer[*ix]=x;
	for(i=0;i<len;i++) acc+=buffer[i];
	if(++*ix==len)*ix=0;
	return acc/len;
}
```

- El error acumulado “vive” durante M muestras ya que el filtro no recuerda más allá de eso.
- https://onlinegdb.com/SkFrNa3bP

Optimizando el promedio móvil

- Si consideramos una señal y[n] a la que se le aplica un promedio móvil de 4 muestras y no dividimos la salida, tenemos:
  - $y[10] = x[10] + x[9] + x[8] + x[7]$
  - $y[11] = x[11] + x[10] + x[9] + x[8]$
  - $y[11] = x[11] + y[10] - x[7]$
- Dónde todo el filtro pasa a ser dos sumas y un buffer con la memoria de las últimas (en este caso) cuatro muestras.
- Para esta implementación es fundamental sumar y restar exactamente las mismas cantidades por lo que no es prácticamente implementable con números en punto flotante.

Promedio móvil. Implementación “recursiva”

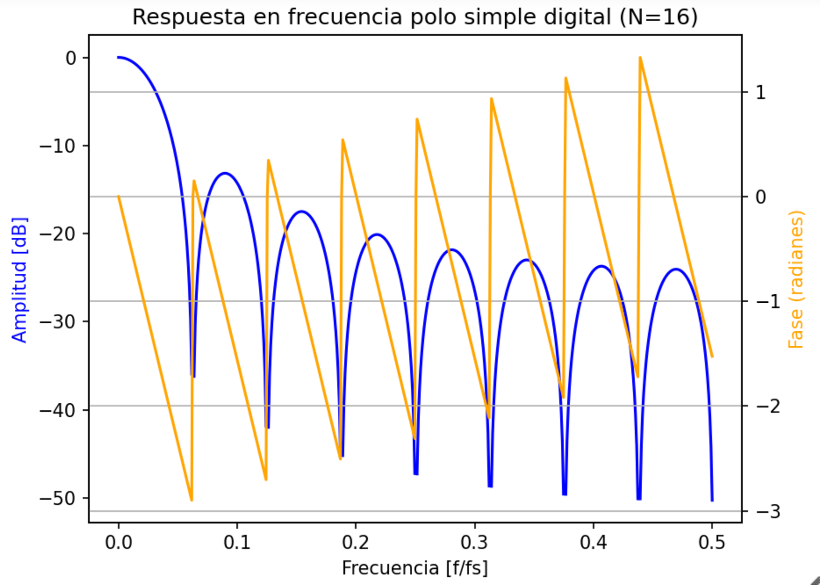
- Si bien la implementación se llama “recursiva” no es tal porque pasadas las muestras del largo del filtro “no se recuerda” el aporte de muestras anteriores.
- Es importante tener en cuenta que si las palabras de entrada son de “b” bits de largo y la cantidad de muestras a promediar son “M”, el acumulador necesita tener al menos b + log2(M) + 1 para evitar desbordes.
- Sí la cantidad de muestras en potencia de dos, la línea: “***return acc/len;***” se puede reemplazar por: “***return acc>>k;***” donde k es la cantidad de bits.

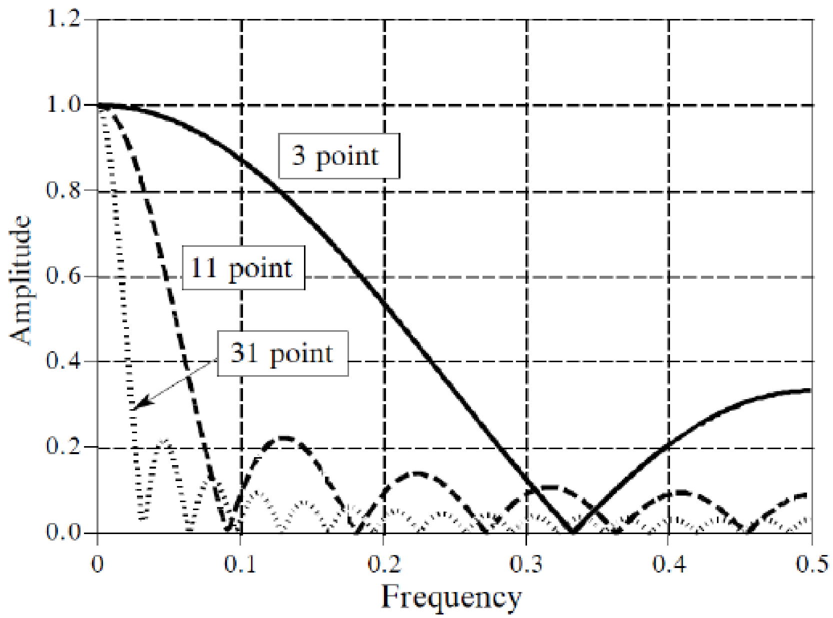
- https://onlinegdb.com/BJMWn6n-D
- https://youtu.be/gqig5EMWoCE
- https://gitlab.frba.utn.edu.ar/jalarcon/ejemplo_promedio_movil

```c
void inicializar_promedio_movil_2(  int *buffer,
                                    int *ix,
                                    int len,
                                    int *acc)
{
    int i;
    *ix=0;
    *acc=0;
    for(i=0;i<len;i++)buffer[i]=0;
}


int promedio_movil_2(   int x,
                        int *buffer,
                        int *ix,
                        int len,
                        int *acc)
{
    acc+=x-buffer[*ix];
    buffer[*ix]=x;
    if(++*ix==len)*ix=0;
    return acc/len;
}
```

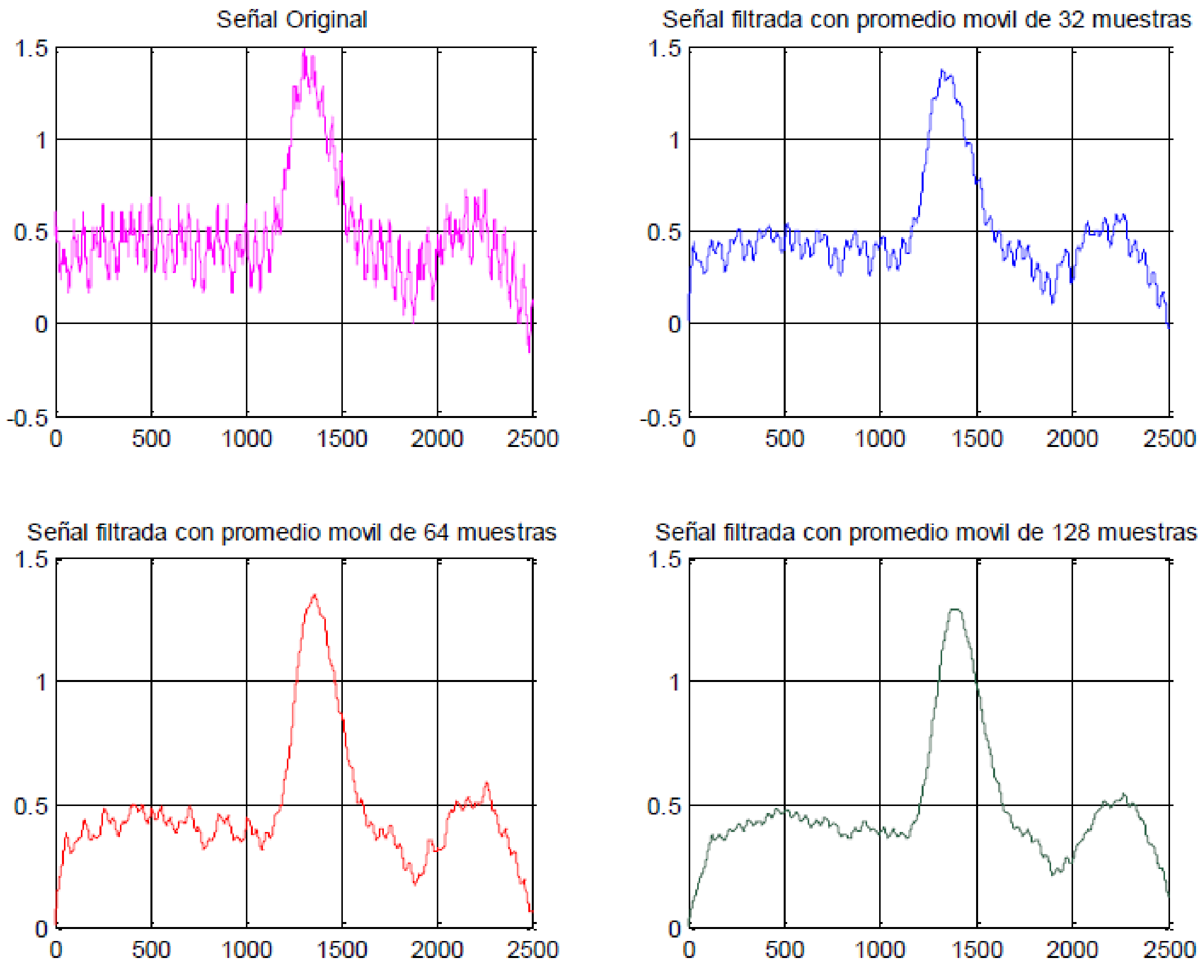
Promedio móvil. Respuesta en frecuencia

- $$H(f) = \frac{sen(\pi f N)}{N sen(\pi f)}$$





Ejemplo de aplicación



Promedio móvil. Resumen

- Es el filtro más simple de todos.☺
- Consume muy rápido, sólo necesita hacer dos sumas y un desplazamiento.☺
- No puede ser demasiado largo, necesita el buffer de muestras y los bits en el acumulador para que no “desborde”.☹
- Es el filtro por excelencia para implementar con números enteros. ☺
- Al tener forma de “sinc”, es un pésimo filtro para usar como pasabajos. ☹
- Es el filtro óptimo para filtrar ruido blanco. ☺

Suma de convolución

```
void sumaconvolucion(int *x, int xlen, int *h, int hlen, int *y)
{
    //Asume que y tiene al menos xlen+hlen-1 elementos y que están
    //inicializados a cero.
    int i,j;
    for (i=0;i<hlen;i++)
    {
        for (j=0; j<xlen; j++)
        {
            y[i+j] += x[j]*h[i];
        }
    }
}
```

- La forma más simple de aplicar algún procesamiento de señal es a través de la suma de convolución.
- Este algoritmo no es útil para utilizar en tiempo real, ya que necesita toda la señal (x[n]) para calcular la convolución.

Convolución en tiempo real

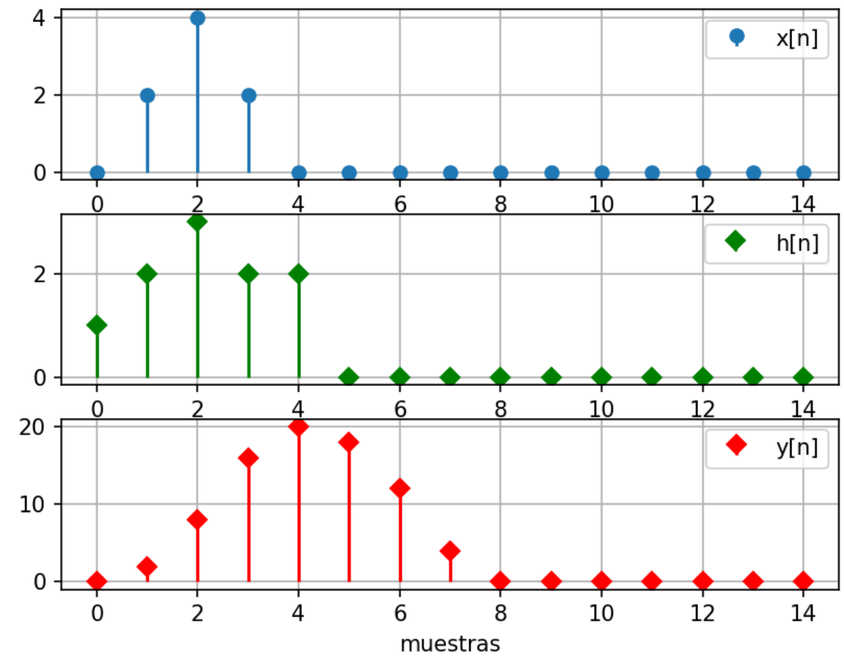
- Para calcular la convolución muestra a muestra es necesario tener un buffer las últimas n muestras.

- $y[m] = h[0]*x[n] + h[1]*x[n-1] + h[2]*x[n-2] + \ldots + h[n]*x[0]$

- Hay que tener en cuenta que si x es una variable de xb bits, h es de hb bits, se van a necesitar variables de xb+bh+n+1 bits para evitar desbordes

Convolución en tiempo real (2)

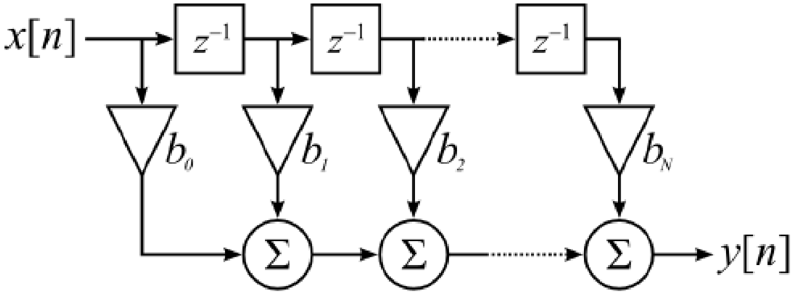
```c
typedef struct {
  int32_t *h;
  int32_t *buf;
  uint32_t len;
  uint32_t i_h;
} estado_convolucion;


int32_t convolucion(estado_convolucion *s, int32_t x)
{
    uint32_t ni,i;
    uint32_t y=0;
    s->buf[s->i_h]=x;
    for(i=0;i<s->len;i++)
    {
        ni = (s->len + s->i_h - i) % s->len;
        y += s->h[i]*s->buf[ni];
    }
    if(++(s->i_h)==s->len) s->i_h=0;
    return y;
}

void init_convolucion( estado_convolucion *s,
                       int32_t h[],
                       int32_t buf[],
                       uint32_t len)
{
  uint32_t i;
  s->h = h;
  s->buf = buf;
  s->len = len;
  s->i_h = 0;
  for(i=0;i<len; i++) s->buf[i]=0;
}
```



- $y[n] = \sum_{i=0}^{N-1} h_i x[n-i]$
- https://onlinegdb.com/SJ057l6Wv

Filtros FIR

- Los filtros de respuesta finita al impulso (FIR) o todos ceros se pueden implementar por suma de convolución.
- Al no tener polos, son filtros que ***siempre son estables.***
- Otra característica de estos filtros es que pueden diseñarse para que tengan ***fase lineal.***



$$
\begin{aligned}
y[n] &= b_0 x[n] + b_1 x[n-1] + \cdots + b_N x[n-N] \\
&= \sum_{i=0}^{N} b_i x[n-i]
\end{aligned}
$$

Filtros FIR. Fase lineal

- Una de las características más importantes de los filtros FIR es la capacidad de poder diseñarlos con fase lineal:
  - $H(\omega) = |H(\omega)|e^{j\phi(\omega)}$ $donde$ $\phi(\omega) = -\omega n_0$

- No todos los filtros FIR son de fase lineal, hay cuatro tipos de FIR que lo pueden satisfacer:
  - $h[n] = \pm h[M-1-n],\ n = 0,1,\dots,M-1$

Impulse response	# coefs	$H(\omega)$	Type
$h(n) = h(M-1-n)$	Odd	$e^{-j\omega(M-1)/2}\left(h\left(\frac{M-1}{2}\right) + 2\sum_{k=1}^{(M-3)/2} h\left(\frac{M-1}{2} - k\right)\cos(\omega k)\right)$	1
$h(n) = h(M-1-n)$	Even	$e^{-j\omega(M-1)/2} 2\sum_{k=1}^{(M-3)/2} h\left(\frac{M}{2} - k\right)\cos\left(\omega\left(k - \frac{1}{2}\right)\right)$	2
$h(n) = -h(M-1-n)$	Odd	$e^{-j[\omega(M-1)/2-\pi/2]}\left(2\sum_{k=1}^{(M-1)/2} h\left(\frac{M-1}{2} - k\right)\sin(\omega k)\right)$	3
$h(n) = -h(M-1-n)$	Even	$e^{-j[\omega(M-1)/2-\pi/2]} 2\sum_{k=1}^{(M-1)/2} h\left(\frac{M}{2} - k\right)\sin\left(\omega\left(k - \frac{1}{2}\right)\right)$	4

Filtros FIR. Observaciones

- Tipo 1. El más versátil, se pueden implementar pasabajos y pasa altos (no aptos para derivadores).

- Tipo 2. La respuesta en frecuencia es siempre nula para $\omega = \pi$ (no se puede usar para pasa altos).

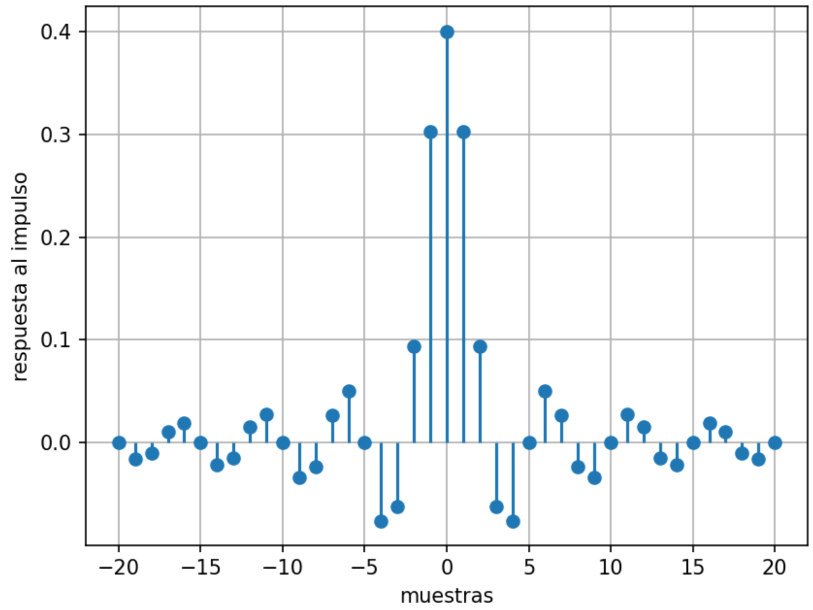
- Tipos 3 y 4. Introducen un cambio de fase de $\pi/_2$ y tienen siempre respuesta en frecuencia nula para $\omega = \pi$. No se los puede usar para construir pasa altos

Diseñando filtros FIR.

- Los métodos tradicionales para el diseño de filtros FIR son:
  - Truncando la respuesta al impulso. Se parte una ventana rectangular en frecuencia discreta, se obtiene la respuesta en el tiempo (una sinc) y la se trunca a una cantidad finita de coeficientes.
  - Métodos de las ventanas. Se trunca la respuesta al impulso con diferentes “ventanas”. Es un método simple y conveniente. No siempre se obtiene el mejor filtro desde el punto de vista de los coeficientes.
  - Métodos de diseño óptimos. Métodos algorítmicos, generalmente iterativos, para obtener el filtro (cuadrados mínimos, equiripple)

Pasabajos ideal

- $H_d(\omega) = \begin{cases} 1 \text{ sí} & |\omega| \leq \omega_c \\ 0 \text{ sí } \omega_c < |\omega| < \pi \end{cases}$
- $h_d[n] = \frac{\omega_c}{\pi} sinc(\omega_c n)$
- Donde esta respuesta al impulso es infinita y anticausal.
- Obviamente no es implementable.
- Para implementar un pasabajos a partir de esta técnica es necesario demorar y truncar la respuesta al impulso.
- ¿Qué operación representa truncar en el tiempo?¿Y en frecuencia?

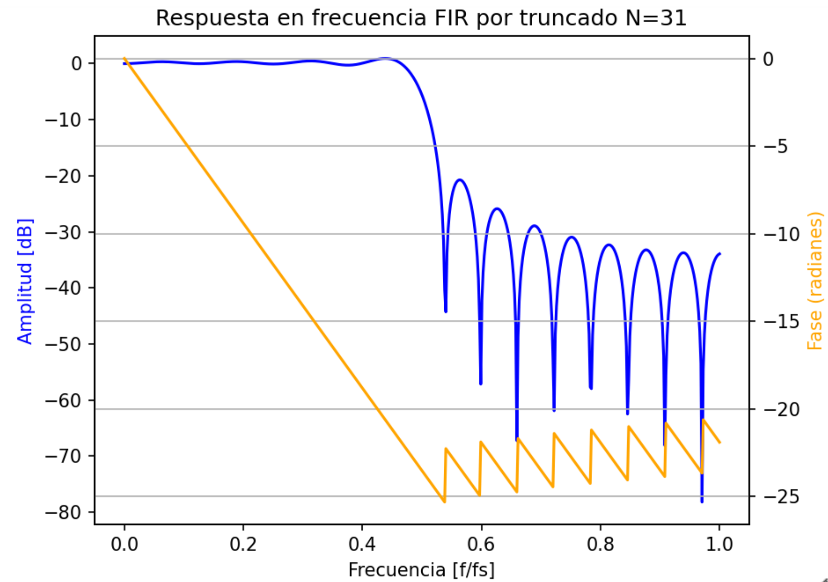


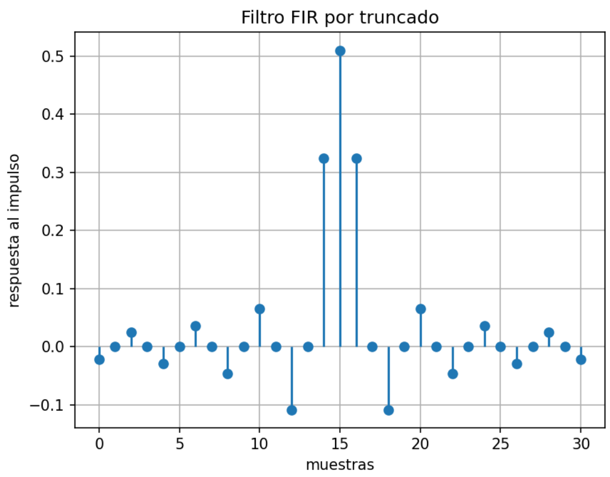
Pasabajos FIR por truncado

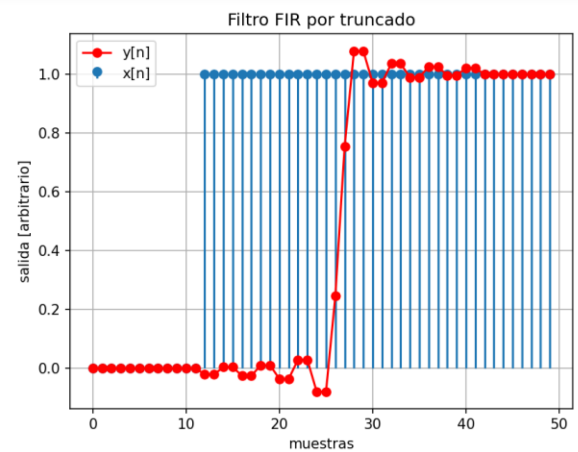
```
%matplotlib notebook
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
```

```
n  = 31
fc = 0.5
b  = signal.firwin(n, fc, window='boxcar')
print(b)
```

```
[-2.16497154e-02  1.98849410e-17  2.49804409e-02 -1.98849410e-17
 -2.95223392e-02  1.98849410e-17  3.60828591e-02 -1.98849410e-17
 -4.63922474e-02  1.98849410e-17  6.49491463e-02 -1.98849410e-17
 -1.08248577e-01  1.98849410e-17  3.24745732e-01  5.10109403e-01
  3.24745732e-01  1.98849410e-17 -1.08248577e-01 -1.98849410e-17
  6.49491463e-02  1.98849410e-17 -4.63922474e-02 -1.98849410e-17
  3.60828591e-02  1.98849410e-17 -2.95223392e-02 -1.98849410e-17
  2.49804409e-02  1.98849410e-17 -2.16497154e-02]
```







Pasabajos FIR. Ventanas

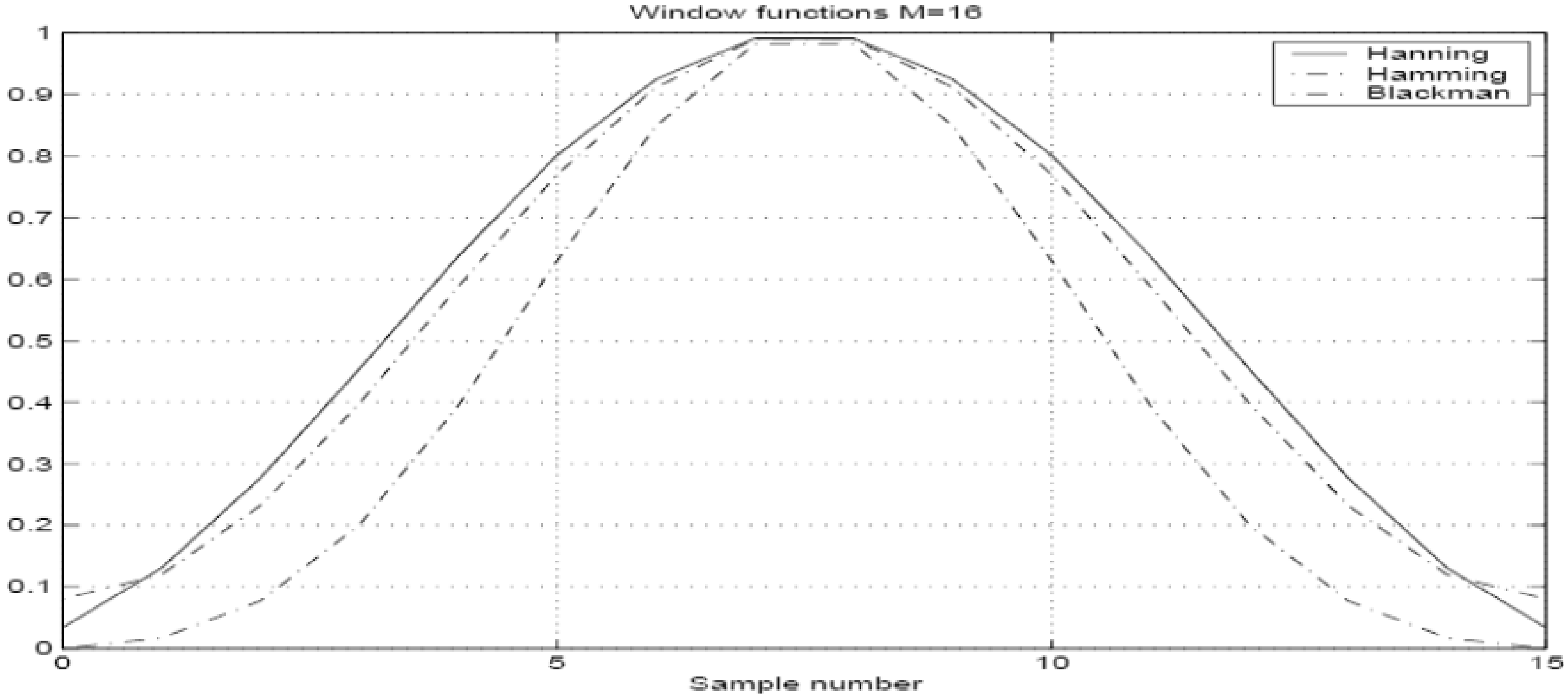
- Truncar la respuesta al impulso, es equivalente a multiplicar por un pulso en el tiempo lo que es igual a hacer una convolución en frecuencia discreta con una sinc por lo que aparece ripple en la banda de atenuación y en la banda de paso.
- Una forma de evitar ese efecto es multiplicar en el tiempo por señales más suaves que un escalón para minimizar ripple en la banda de paso y de atenuación pagando con una zona de transición más ancha. Donde interesa la relación entre el lóbulo principal y el lateral, además del ripple
- Las funciones por las que se afecta a la respuesta al impulso se las conoce como ***ventanas.*** Las más comunes son:
  - Hanning
  - Hamming
  - Blackman

The suite of window functions for filtering and spectral estimation.

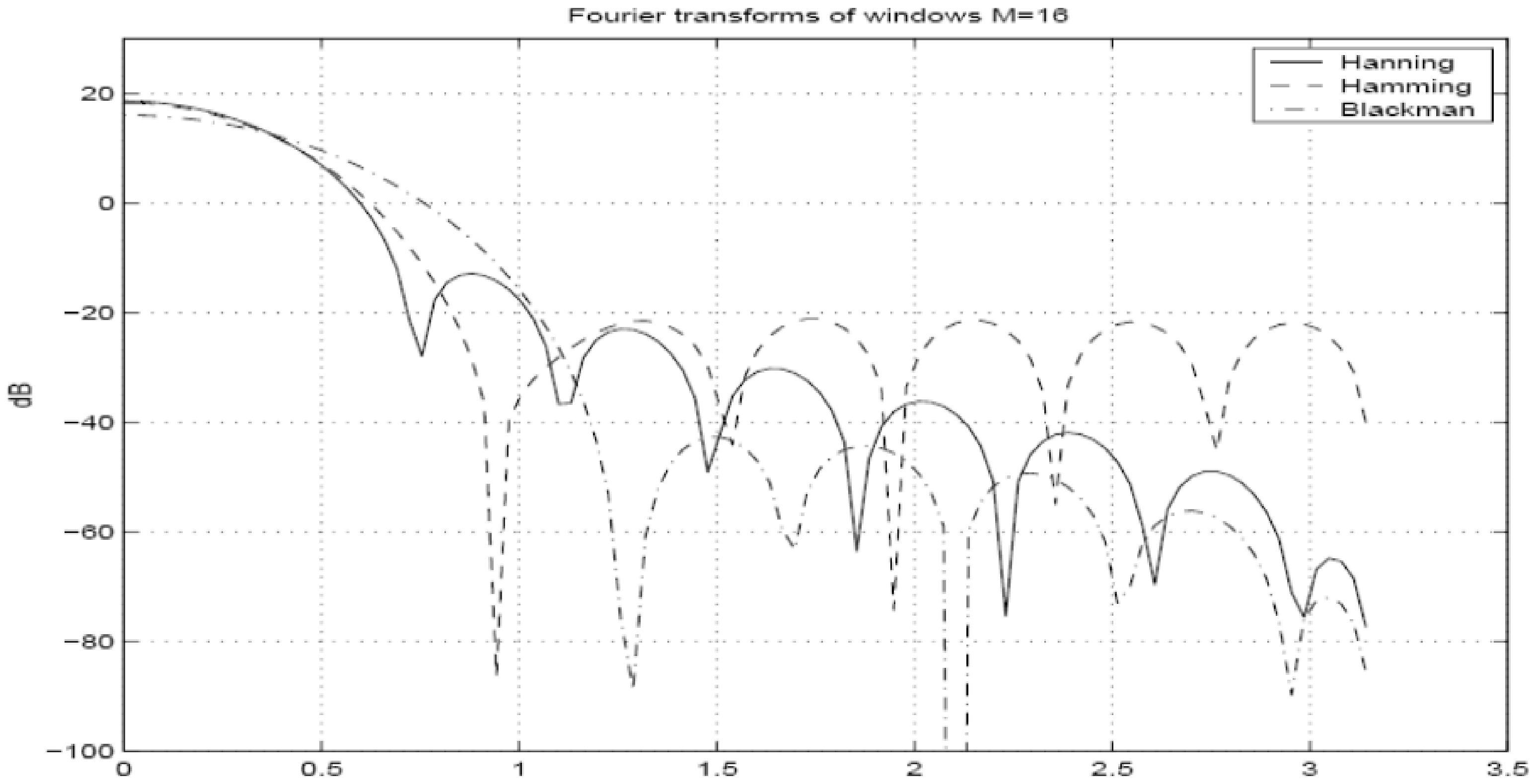
get_window(window, Nx[, fftbins])	Return a window of a given length and type.
barthann(M[, sym])	Return a modified Bartlett-Hann window.
bartlett(M[, sym])	Return a Bartlett window.
blackman(M[, sym])	Return a Blackman window.
blackmanharris(M[, sym])	Return a minimum 4-term Blackman-Harris window.
bohman(M[, sym])	Return a Bohman window.
boxcar(M[, sym])	Return a boxcar or rectangular window.
chebwin(M, at[, sym])	Return a Dolph-Chebyshev window.
cosine(M[, sym])	Return a window with a simple cosine shape.
dpss(M, NW[, Kmax, sym, norm, return_ratios])	Compute the Discrete Prolate Spheroidal Sequences (DPSS).
exponential(M[, center, tau, sym])	Return an exponential (or Poisson) window.
flattop(M[, sym])	Return a flat top window.
gaussian(M, std[, sym])	Return a Gaussian window.
general_cosine(M, a[, sym])	Generic weighted sum of cosine terms window
general_gaussian(M, p, sig[, sym])	Return a window with a generalized Gaussian shape.
general_hamming(M, alpha[, sym])	Return a generalized Hamming window.
hamming(M[, sym])	Return a Hamming window.
hann(M[, sym])	Return a Hann window.
hanning(*args, **kwds)	hanning is deprecated, use scipy.signal.windows.hann instead!
kaiser(M, beta[, sym])	Return a Kaiser window.
nuttall(M[, sym])	Return a minimum 4-term Blackman-Harris window according to Nuttall.
parzen(M[, sym])	Return a Parzen window.
slepian(M, width[, sym])	Return a digital Slepian (DPSS) window.
triang(M[, sym])	Return a triangular window.
tukey(M[, alpha, sym])	Return a Tukey window, also known as a tapered cosine window.

https://docs.scipy.org/doc/scipy/reference/signal.windows.html

Ventanas en el tiempo



Ventanas en frecuencia



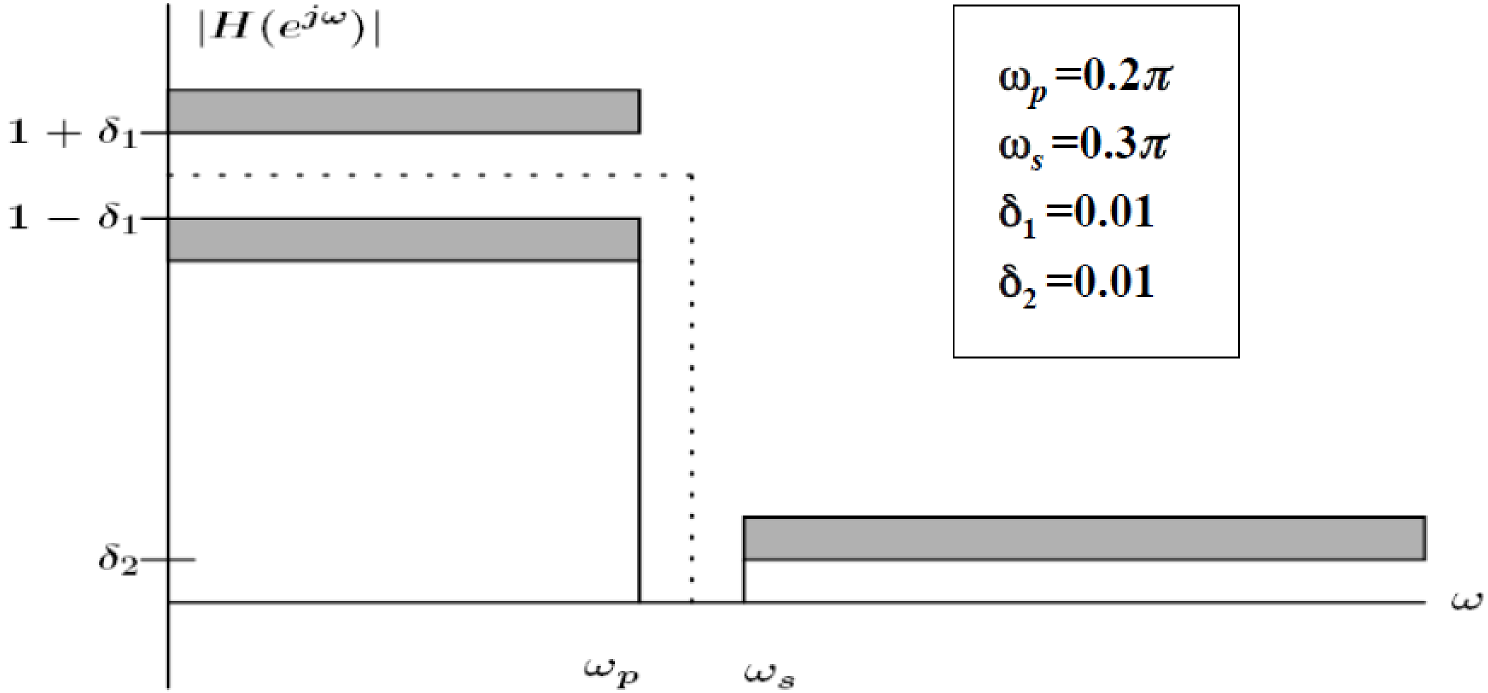
Ventanas. Resumen

Window's name	Mainlobe	Mainlobe/sidelobe	Peak $20\log_{10}\delta$
Rectangular	$4\pi/\ M$	-13dB	-21dB
Hanning	$8\pi/M$	-32dB	-44dB
Hamming	$8\pi/M$	-43dB	-53dB
Blackman	$12\pi/M$	-58dB	-74dB

- Donde vemos que la zona de transición (ancho del lóbulo principal) se hace más ancho en las ventanas pero con menos ripple y lóbulos laterales más pequeños

Diseñando un filtro por ventanas

- Vamos a diseñar un filtro pasabajos con la siguiente especificación



$\omega_p = 0.2\pi$
$\omega_s = 0.3\pi$
$\delta_1 = 0.01$
$\delta_2 = 0.01$

Diseñando un FIR por ventanas (1)

Window's name	Mainlobe	Mainlobe/sidelobe	Peak $20\log_{10}\delta$
Rectangular	$4\pi/\ M$	-13dB	-21dB
Hanning	$8\pi/M$	-32dB	-44dB
Hamming	$8\pi/M$	-43dB	-53dB
Blackman	$12\pi/M$	-58dB	-74dB

- $\delta_2 = 0.01 \Longrightarrow 20log_{10}(\delta_2) = 20log_{10}(0.01) = -40dB \ \Longrightarrow Hanning$
- $\omega_s - \omega_p = 0.3\pi - 0.2\pi = 0.1\pi$
- $0.1\pi = 8\pi/M \Longrightarrow M \geq 80$

```
%matplotlib notebook
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
```

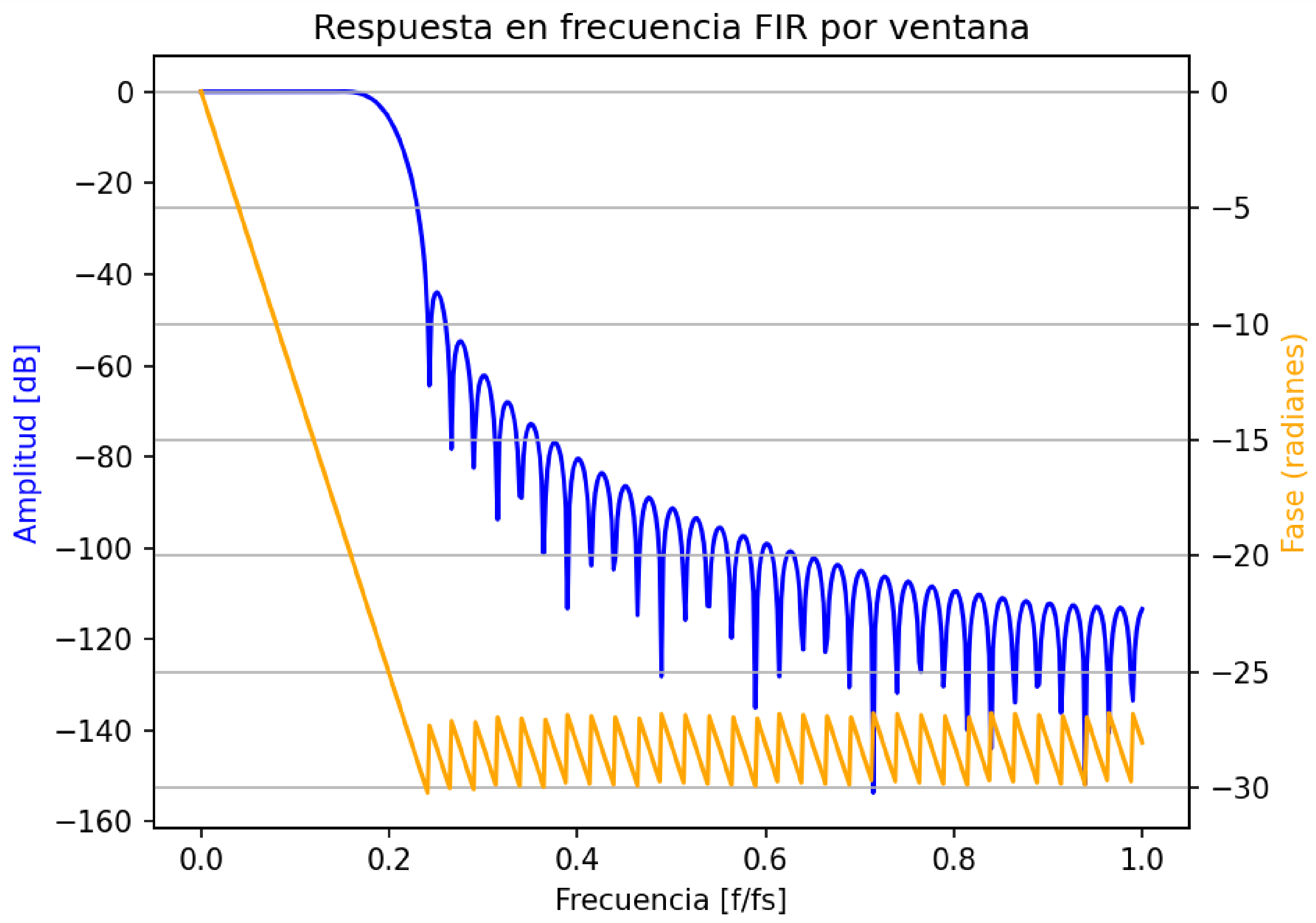
```
n  = 81
fp = 0.2
b  = signal.firwin(n, fp, window='hann')
```

```
w, h = signal.freqz(b)
```

```
fig, ax1 = plt.subplots()
ax1.plot(w/(max(w)), 20 * np.log10(abs(h)), 'b')
ax1.set_title('Respuesta en frecuencia FIR por ventana')
ax1.set_ylabel('Amplitud [dB]', color='b')
ax1.set_xlabel('Frecuencia [f/fs]')
ax2 = ax1.twinx()
angles = np.unwrap(np.angle(h))
ax2.plot(w/(max(w)), angles, 'orange')
ax2.set_ylabel('Fase (radianes)', color='orange')
ax2.grid()
ax2.axis('tight')
plt.show()
```

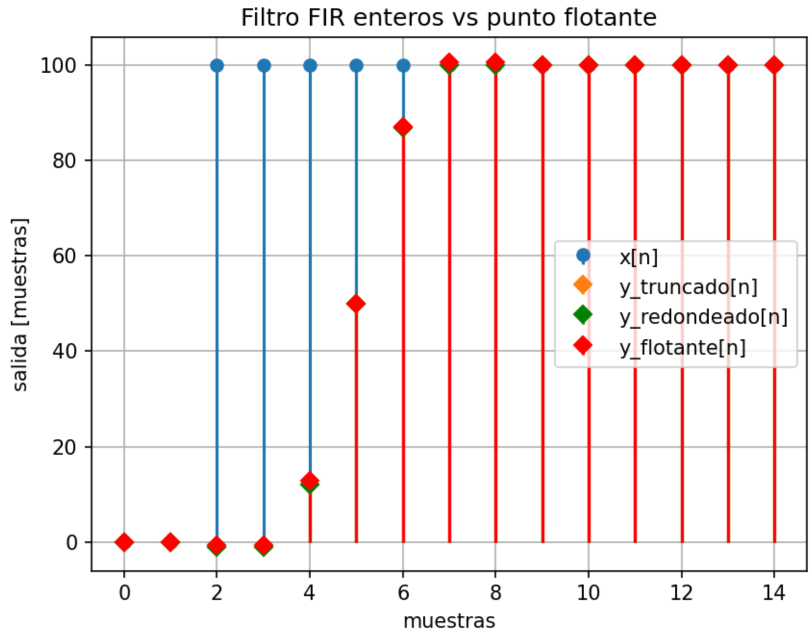
Se puede correr este ejemplo en https://www.kaggle.com/esalarcon1900/ejemplo-filtro-fir-por-ventanas

Diseñando un FIR por ventanas (2)



Filtros FIR. Implementando

- Partamos de un filtro de 8 coeficientes con una $\omega_C = 0.4\pi$, 8 coeficientes y ventana de Hamming.
- Vamos a intentar implementarlo en punto fijo en C.
- Como es un filtro sólo de ceros y con una cantidad finita de coeficientes lo podemos implementar usando la suma de convolución que hicimos y reescalando la salida.
- https://onlinegdb.com/rJRUdJRZw




```
n = 8
Run this cell
b = signal.firwin(n,f)
print(b)
print(np.round(b*1024))
```

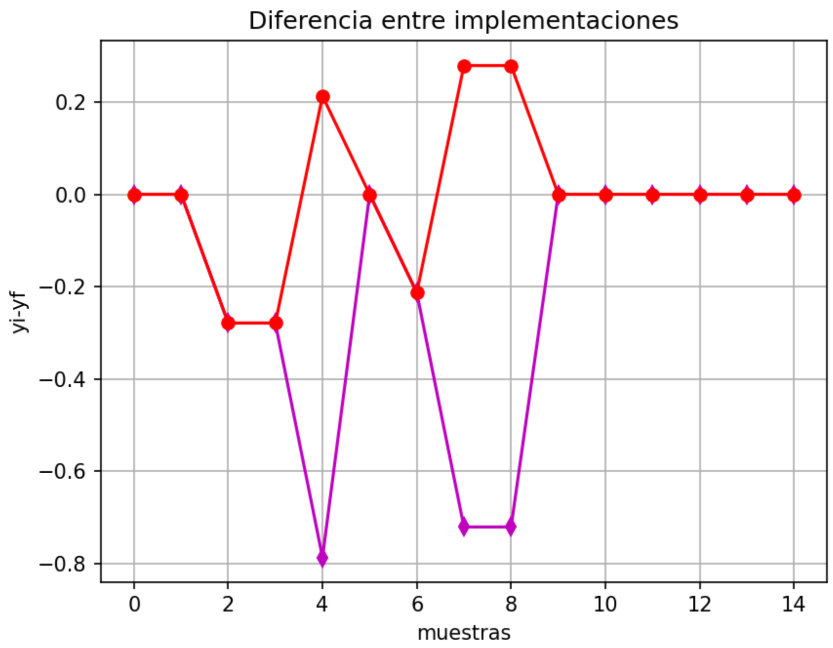
```
[-7.20980764e-03  4.11358853e-18  1.35079273e-01  3.72130534e-01
  3.72130534e-01  1.35079273e-01  4.11358853e-18 -7.20980764e-03]
[ -7.   0. 138. 381. 381. 138.   0.  -7.]
```

Filtros FIR. Implementando

```
int
main (int argc, char *argv[])
{
  int i;
  uint32_t h[8] = { -7, 0, 138, 381, 381, 138, 0, -7 };
  uint32_t b[8];
  uint32_t x[15] =
    { 0, 0, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100 };
  estado_convolucion c;

  init_convolucion (&c, h, b, 8);
  for (i = 0; i < 15; i++)
    printf ("%d;%d\n", x[i], convolucion (&c, x[i]) >> 10);
  return EXIT_SUCCESS;
}
```

- La aplicación de https://onlinegdb.com/rJRUdJRZw implementa un filtro FIR con coeficientes enteros con la salida reescalada y truncando. En cambio https://onlinegdb.com/rkkD1lRWP implementa con salida redondeada.
- En la implementación redondeada ¿Qué significa `(convolucion (&c, x[i])+512) >> 10`?
- Es importante para esta aplicación conocer la cantidad de bits de los coeficientes (h) y de los posibles valores de x para evitar desbordes.

Filtros FIR. Error por truncado y redondeo



- La implementación en enteros genera una diferencia en la salida por el truncado (eliminar la parte decimal) que si se calcula en la implementación en punto flotante.
- Cómo los filtros FIR no utilizan sus salidas pasadas este error ***no se acumula*** en las sucesivas salidas***.***
- Truncando se mantuvo debajo de una cuenta, redondeando, debajo de media.

Filtros FIR. Por cuadrados mínimos

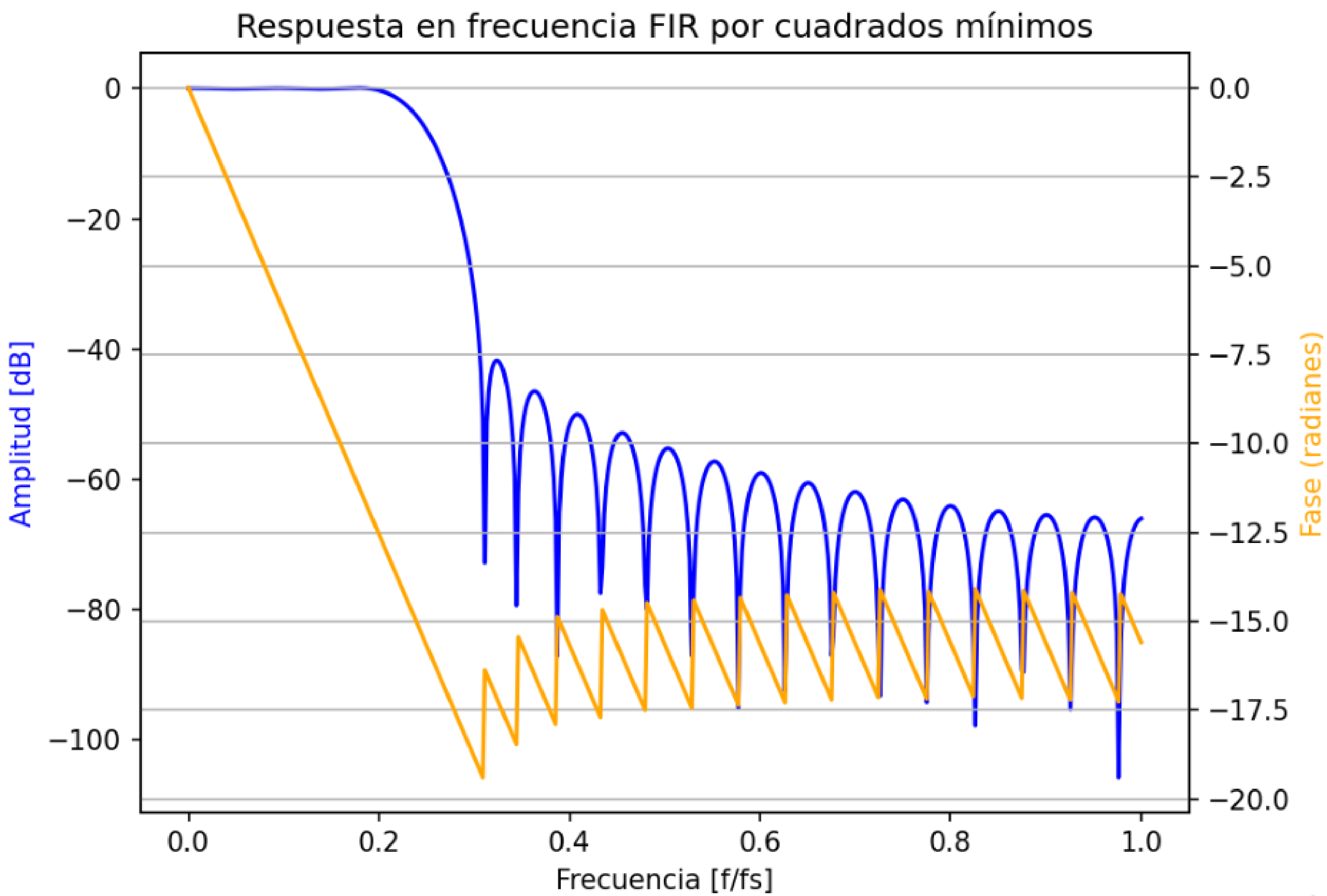
- Los cuadrados mínimos es un método por el cual se minimiza el cuadrado del error entre una función dada y una función objetivo para encontrar la un conjunto de coeficientes "óptimo" desde ese punto de vista

- Para el diseño de filtros FIR se minimiza la función ε^2:
  - $E(\omega) = W(\omega)[H_d(\omega) - H(\omega)]$
    - $W(\omega)$. Pesos
    - $H_d(\omega)$. Transferencia ideal
    - $H(\omega)$. Transferencia medida
  - $\varepsilon^2 = \int E(\omega)^2 \, \mathrm{d}\omega$

```
%matplotlib notebook
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
```

```
n  = 41
b  = signal.firls(n,[0, 0.2, 0.3, 1],[1, 1, 0, 0])

w, h = signal.freqz(b)
fig, ax1 = plt.subplots()
ax1.plot(w/(max(w)), 20 * np.log10(abs(h)), 'b')
ax1.set_title('Respuesta en frecuencia FIR por ventana')
ax1.set_ylabel('Amplitud [dB]', color='b')
ax1.set_xlabel('Frecuencia [f/fs]')
ax2 = ax1.twinx()
angles = np.unwrap(np.angle(h))
ax2.plot(w/(max(w)), angles, 'orange')
ax2.set_ylabel('Fase (radianes)', color='orange')
ax2.grid()
ax2.axis('tight')
plt.show()
```

Filtro FIR por cuadrados mínimos



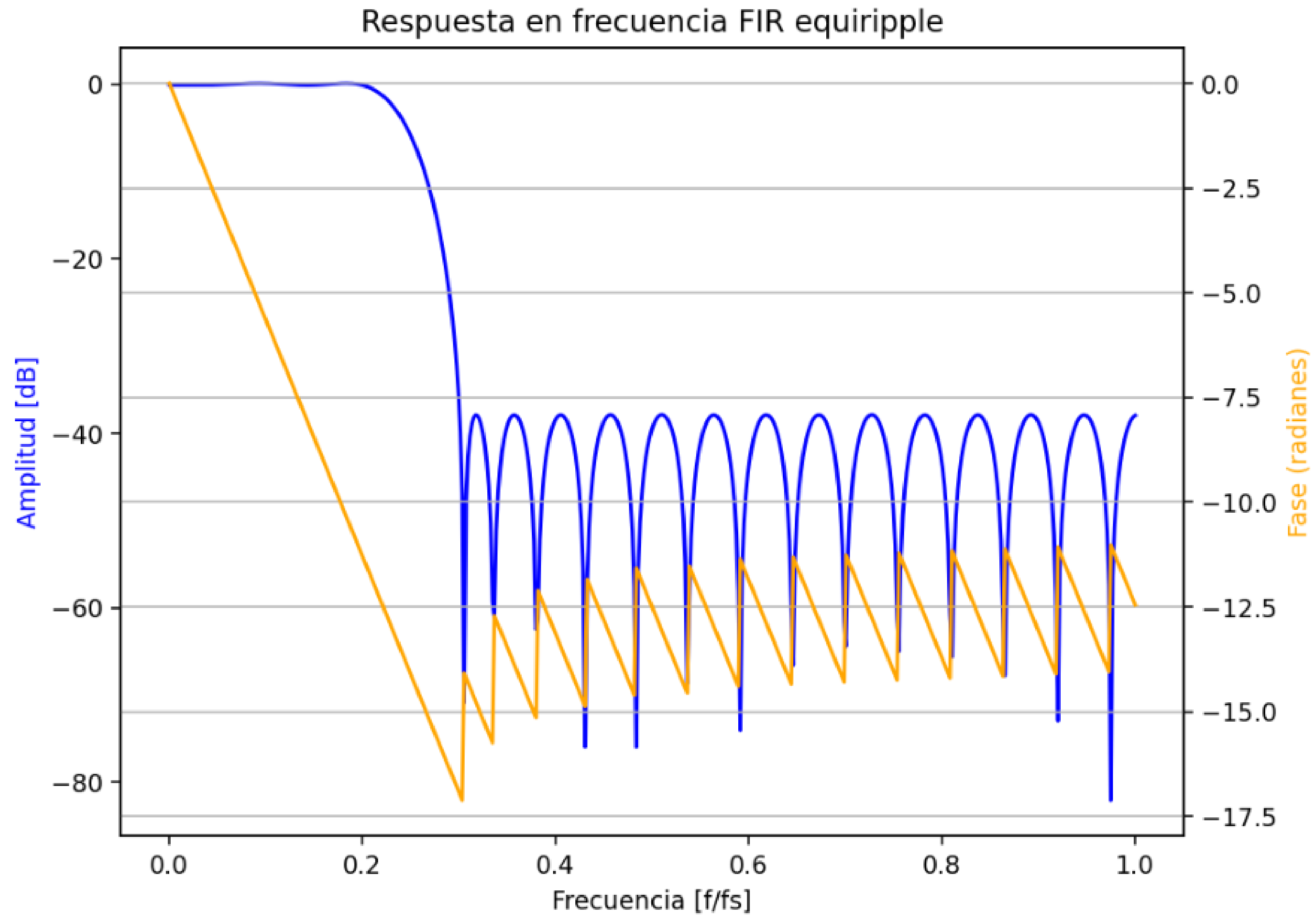
Filtro FIR. Equiripple

- El diseño de filtros FIR por criterio equiripple es similar al de cuadrados mínimos pero la ecuación a minimizar es diferente.
- El diseño equiripple es óptimo en el sentido que busca minimizar el máximo ripple en las bandas de interés.
  - $\min_{H_d} \max_{\omega} |E(\omega)|$

```
%matplotlib notebook
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
```

```
n  = 37
b  = signal.remez(n,[0, 0.2/2, 0.3/2, 1/2],[1, 0])

w, h = signal.freqz(b)
fig, ax1 = plt.subplots()
ax1.plot(w/(max(w)), 20 * np.log10(abs(h)), 'b')
ax1.set_title('Respuesta en frecuencia FIR equiripple')
ax1.set_ylabel('Amplitud [dB]', color='b')
ax1.set_xlabel('Frecuencia [f/fs]')
ax2 = ax1.twinx()
angles = np.unwrap(np.angle(h))
ax2.plot(w/(max(w)), angles, 'orange')
ax2.set_ylabel('Fase (radianes)', color='orange')
ax2.grid()
ax2.axis('tight')
plt.show()
```

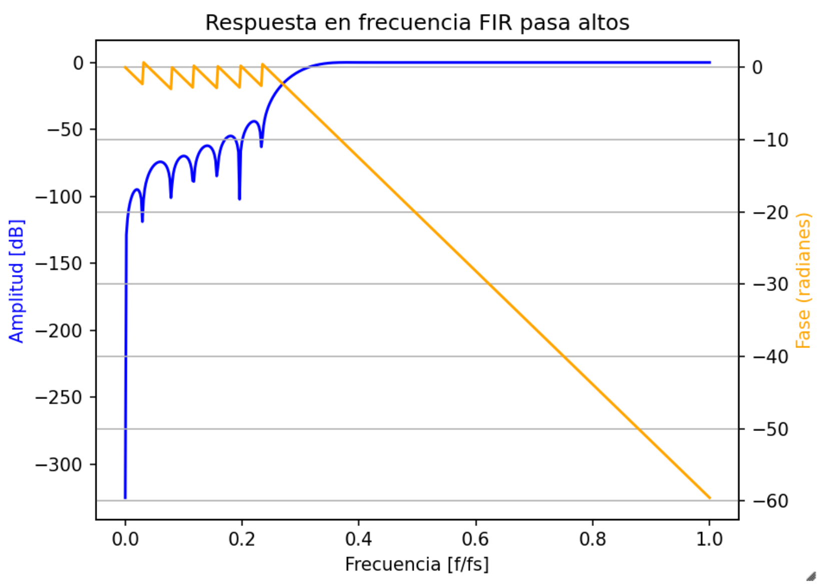
Filtro FIR. Equiripple



Filtros FIR. Pasa altos.

- Si bien las herramientas presentadas pueden generar filtros, pasabajos, pasaaltos, pasabanda y eliminabanda la forma más simple de generar un pasaaltos (FIR tipo 1) teniendo un pasabajos es invirtiendo el espectro.

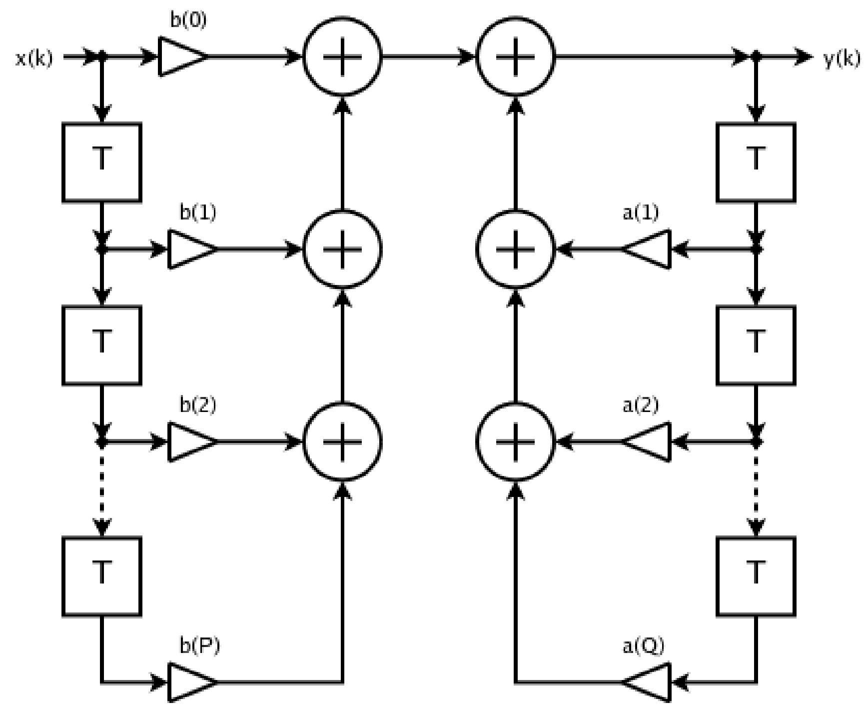
- Para generar pasabandas y eliminabandas con esta técnica es cuestión de combinar pasabajos y pasaaltos.




```python
n  = 51
b  = signal.firwin(n,0.3, window = "hanning")
b = -b
b[n//2] = b[n//2] + 1
```

Filtros IIR.

- Los filtros de respuesta infinita al impulso (IIR) son filtros que van a presentar polos y pueden o no presentar ceros.
- Son filtros que es necesario ***pueden ser inestables.*** Ya que presentan polos
- Son filtros que no van a tener fase lineal pero pueden tener buenas aproximaciones a las misma.
- A diferencia de los FIR, se pueden derivar de filtros analógicos conocidos



$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{i=0}^{P} b_i z^{-i}}{\sum_{j=0}^{Q} a_j z^{-j}}$$

$$y[n] = \frac{1}{a_0}(b_0 x[n] + b_1 x[n-1] + \cdots + b_P x[n-P] - a_1 y[n-1] - a_2 y[n-2] - \cdots - a_Q y[n-Q])$$

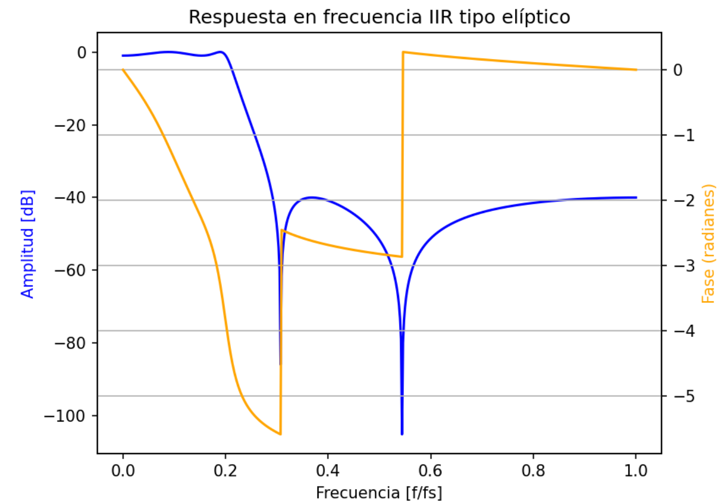
Filtros IIR. Diseño

- Las formas habituales para diseñar este tipo de filtros son:
  - Indirectas (a partir de prototipos analógicos)
    - Invarianza al impulso
    - Transformación bilineal
    - Aproximación de derivadas
  - Directa
    - Por cuadrados mínimos
    - Aproximación de Padé
  - La herramienta permite sintetizar:
    - Butterworth
    - Chebyshev I
    - Chebyshev II
    - Cauer / elípticos
    - Bessel

```
%matplotlib notebook
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
```

```
wc = 0.2
ws = 0.3
gpass = 1
gstop = 40
b,a=signal.iirdesign(wc,ws,gpass,gstop,ftype='ellip')
print(b)
print(a)
```

```
[ 0.01967691 -0.01714282  0.03329653 -0.01714282  0.01967691]
[ 1.         -3.03302405  3.81183153 -2.29112937  0.5553678 ]
```



Filtro IIR. Implementación

```c
typedef struct {
  float *b;
  float *a;
  float *buf_b;
  float *buf_a;
  int    len_b;
  int    len_a;
  int    i_b;
  int    i_a;
} estado_iir;


float filtro_iir(estado_iir *s, float x)
{
    int ni,i;
    float y=0.0;
    s->buf_b[s->i_b]=x;
    for(i=0;i<s->len_b;i++)
    {
        ni = (s->len_b + s->i_b - i) % s->len_b;
        y += s->b[i]*s->buf_b[ni];
    }
    for(i=1;i<s->len_a; i++)
    {
      ni = (s->len_a + s->i_a - i) % s->len_a;
      y -= s->a[i]*s->buf_a[ni];
    }
    y /= s->a[0];
    s->buf_a[s->i_a] = y;
    if(++(s->i_a)==s->len_a) s->i_a=0;
    if(++(s->i_b)==s->len_b) s->i_b=0;
    return y;
}
```

```c
void init_iir(estado_iir *s,
                  float a[], float buf_a[], int len_a,
                  float b[], float buf_b[], int len_b)
{
  int i;
  s->a = a;
  s->b = b;
  s->buf_a = buf_a;
  s->buf_b = buf_b;
  s->len_a = len_a;
  s->len_b = len_b;
  s->i_a = 0;
  s->i_b = 0;
  for(i=0;i<len_a; i++) s->buf_a[i]=0;
  for(i=0;i<len_b; i++) s->buf_b[i]=0;
}

int main()
{
    int i;
    float b[5]={ 0.01967691, -0.01714282,  0.03329653, -0.01714282,  0.01967691};
    float a[5]={ 1,          -3.03302405,  3.81183153, -2.29112937,  0.5553678};
    float bb[5];
    float ba[5];
    estado_iir f;

    init_iir(&f,a,ba,5,b,bb,5);
    printf("%0.03f;%0.03f\n",0.0, filtro_iir(&f,0.0));
    for(i=0; i<99;i++)
      printf("%0.03f;%0.03f\n",1.0,filtro_iir(&f,1.0));
    return EXIT_SUCCESS;
}
```

https://onlinegdb.com/ryGh4QR-v

Filtros IIR. Problemas numéricos

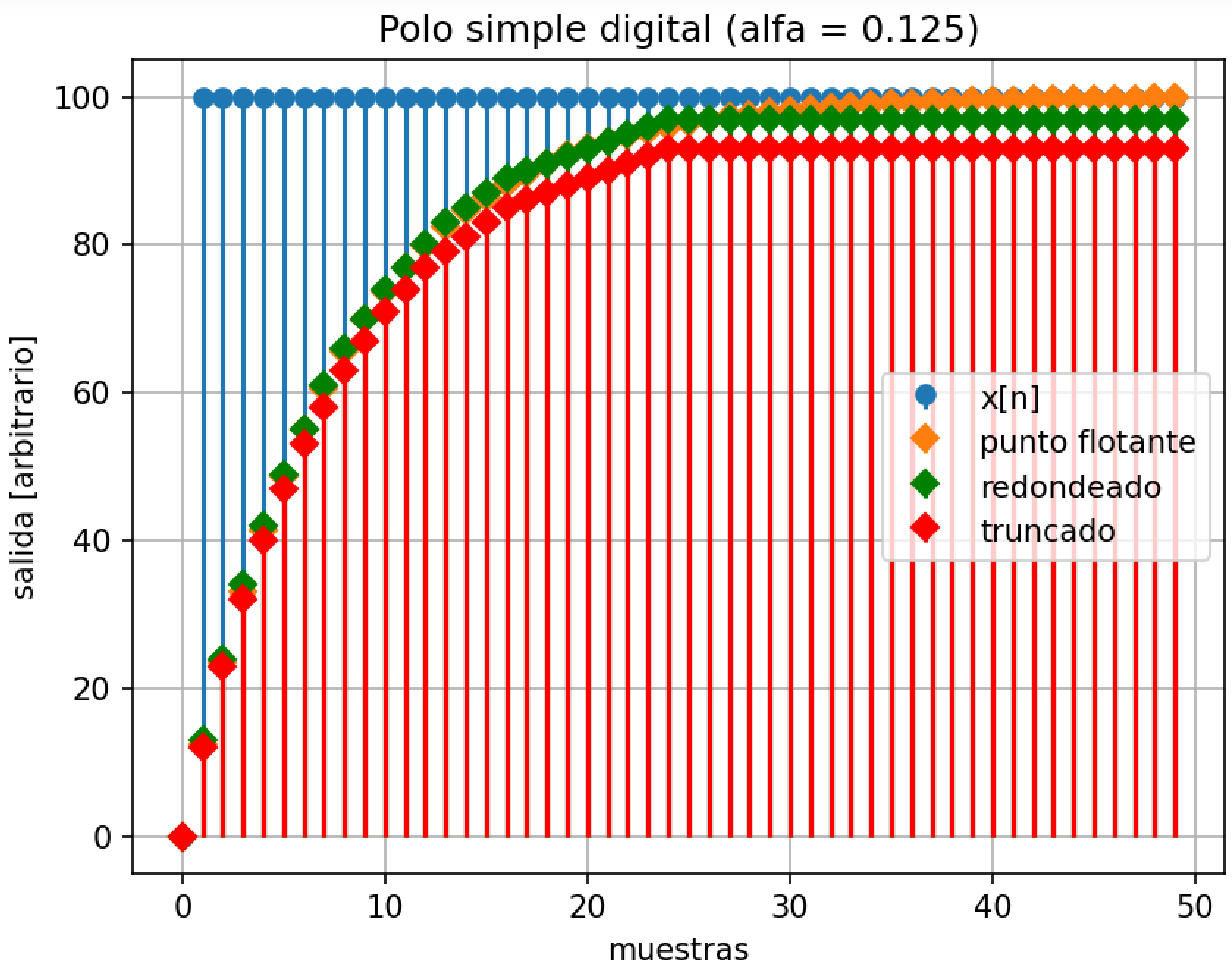
- Vamos a tomar el filtro IIR más sencillo de todos, un polo digital simple e implementarlo en punto flotante, truncando y redondeando y tomamos α=0.125

- $y[n] = \alpha x[n] + (1 - \alpha) y[n-1]$

- https://onlinegdb.com/r1Aqi70bD

```c
float
polosimple (float x, float alfa)
{
  static float yp = 0.0;
  return (yp = alfa * x + (1.0 - alfa) * yp);
}

int
polo_simple_entero (int x, int k)
{
  static int yp = 0;
  return (yp += (x - yp) >> k);
}

int
polo_simple_redondeo (int x, int k)
{
  static int yp = 0;
  int r = 1 << (k - 1);
  return (yp += (x - yp + r) >> k);
}
```

Filtros IIR. Problemas numéricos

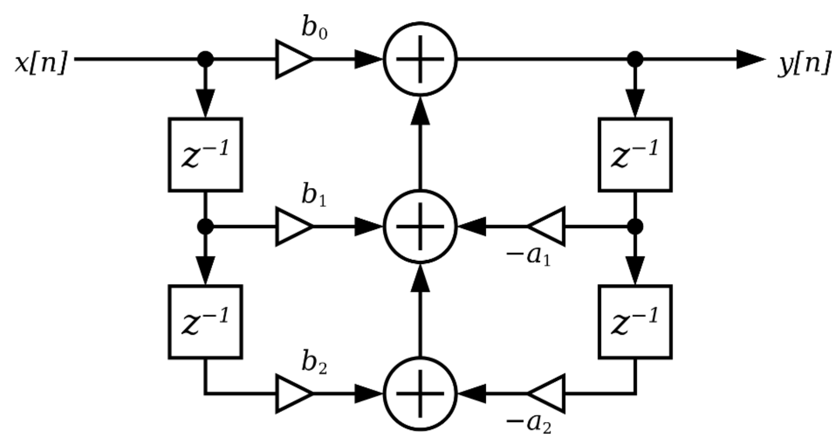


Filtros IIR. Secciones de segundo orden

- Un filtro IIR puede volverse inestable por problemas numéricos

- En la medida que un filtro IIR tiene más coeficientes es más complicado asegurar su “resistencia” a los problemas numéricos.

- Una solución a tener filtros IIR de muchos coeficientes es partirlos en secciones de segundo orden (SOS) y ponerlas en cascada



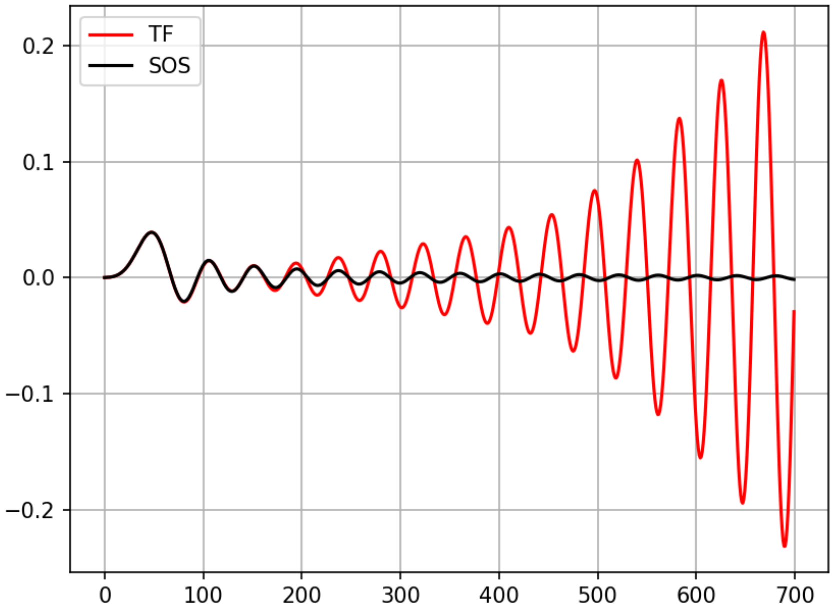
$$y[n] = \frac{1}{a_0}\left(b_0 x[n] + b_1 x[n-1] + b_2 x[n-2] - a_1 y[n-1] - a_2 y[n-2]\right)$$

```
sys = signal.iirdesign(wc,ws,gpass,gstop,ftype='ellip',output='sos')
print(sys)
```

```
[[ 0.01967691  0.00530872  0.01967691  1.          -1.50881467  0.62864308]
 [ 1.         -1.14100949  1.          1.          -1.52420938  0.88343898]]
```

Filtros IIR. Inestabilidad por errores numéricos

```
%matplotlib notebook
import matplotlib.pyplot as plt
from scipy import signal
b, a = signal.ellip(13, 0.009, 80, 0.05, output='ba')
sos = signal.ellip(13, 0.009, 80, 0.05, output='sos')
x = signal.unit_impulse(700)
y_tf = signal.lfilter(b, a, x)
y_sos = signal.sosfilt(sos, x)
plt.plot(y_tf, 'r', label='TF')
plt.plot(y_sos, 'k', label='SOS')
plt.legend(loc='best')
plt.grid(True)
plt.show()
```



Filtros IIR. Implementando

```c
#include <stdio.h>

typedef struct
{
  float y[3];
  float x[3];
  float b[3];
  float a[3];

} sos_stat;

void sos_init (sos_stat * s, float *c)
{
  int i;
  for (i = 0; i < 3; i++)
    {
      s->y[i] = 0.0;
      s->x[i] = 0.0;
      s->b[i] = c[i];
      s->a[i] = c[i + 3];
    }
}

float sos_cal (sos_stat * s, float x)
{
  float y;
  s->y[2] = s->y[1];
  s->y[1] = s->y[0];
  s->x[2] = s->x[1];
  s->x[1] = s->x[0];
  s->x[0] = x;
  y = s->b[0] * s->x[0] + s->b[1] * s->x[1] + s->b[2] * s->x[2];
  y -= s->a[1] * s->y[1] + s->a[2] * s->y[2];
  y = y / s->a[0];
  s->y[0] = y;
  return y;
}
```

```c
int main (void)
{
  int i, j;
  float y[2];
  sos_stat s[2];
  float c[2][6] =
    { {0.01967691, 0.00530872, 0.01967691, 1, -1.50881467, 0.62864308},
  {1, -1.14100949, 1, 1, -1.52420938, 0.88343898}
  };

  for (i = 0; i < 2; i++)
    sos_init (&s[i], c[i]);
  for (i = 0; i < 50; i++)
    {
      for (j = 0; j < 2; j++)
    {
      if (!j)
        y[j] = sos_cal (&s[j], 100.0);
      else
        y[j] = sos_cal (&s[j], y[j - 1]);
    }
      printf ("%d;%f\n", 100, y[1]);
    }
  return 0;
}
```

https://onlinegdb.com/BJjST4RbP

Referencias

- Steven W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. http://www.dspguide.com
- SciPy Signal Processing Toolbox. https://docs.scipy.org/doc/scipy/reference/tutorial/signal.html
- Vijak K. Madisetti, Douglas B. Williams. *Digital Signal Processing Handbook.* Chapman & Hall/CRCnetBase.
- J.G. Proakis, D.G. Manolakis. *Tratamiento digital de señales.* Prentice Hall.