DORA MARIA BALLESTEROS - DIEGO RENZA

Procesamiento digital de señales

utilizando Python

Universidad Militar Nueva Granada

editorial redipe

Investigación Educativa & Pedagógica Iberoamericana

editorial redipe © 2023

Título original
Procesamiento digital de señales utilizando Python

Autores:
Dora Maria Ballesteros
Diego Renza

Universidad Militar Nueva Granada
Facultad de Ingeniería- Ingeniería en Telecomunicaciones

Editorial
REDIPE Red Iberoamericana de Pedagogía
Capítulo Estados Unidos
Bowker Books in Print

Editor
Julio César Arboleda Aparicio

Diagramación
Oliver García Ramos

ISBN: 978-1-957395-22-7
Primera edición: Marzo 2023

Red Iberoamericana de Pedaogía
editorial@redipe.org
www.redipe.org

Impreso en Cali, Colombia
Printed in Cali, Colombia

Tabla de Contenido

Índice de Figuras

Prólogo

Hoy en día el procesamiento digital de señales se ha convertido en una herramienta indispensable en diversas áreas del conocimiento, desde la medicina hasta las comunicaciones, pasando por la industria musical y el diseño de sistemas electrónicos. En cualquiera de estas áreas, así como de muchas otras, que requieran análisis y manipulación de la información, para, por ejemplo, mejorar su calidad o identificar patrones.

Dentro del ámbito del procesamiento digital de señales, el lenguaje de programación Python se ha convertido en una herramienta muy popular debido a su facilidad de uso, gran cantidad de librerías disponibles y su capacidad para manejar grandes volúmenes de datos. Es por ello que este libro titulado "Procesamiento digital de señales utilizando Python" está escrito para proporcionar una base sólida en la temática, con énfasis en la implementación práctica apoyándose en este lenguaje de programación. Por ello, cada capítulo incluye ejemplos de código en Python y problemas resueltos para ayudar a los estudiantes a aplicar los conceptos teóricos presentados. Está enfocado a estudiantes de pregrado de ingeniería, especialmente de sistemas, electrónica, telecomunicaciones, mecatrónica, multimedia y programas afines.

El libro está organizado en seis capítulos que abarcan desde la digitalización de señales análogas, hasta el procesamiento digital de imágenes. Se recomienda su lectura de forma secuencial, para un mejor entendimiento de las explicaciones, ecuaciones, ejemplos y códigos presentados en el documento.

En el primer capítulo se introduce el concepto de señales digitales y la importancia de su análisis y procesamiento a través de operaciones de muestreo y cuantización, así como las implicaciones que tiene la selección de la frecuencia de muestreo y el número de bits de cuantización, tanto en la calidad de la señal muestreada, como en el almacenamiento y transmisión de la señal.

En el segundo capítulo se presentan la Transformada Z de señales discretas de duración finita, la función de transferencia de sistemas LTI, diagramas de bloques de sistemas discretos, y la diferencia entre filtros FIR e IIR a partir de la ecuación de entrada-salida, función de transferencia y respuesta al impulso del sistema. Por su parte, en el tercer capítulo se explica el diseño de filtros de promedio, el filtrado de señales 1D, su comportamiento en frecuencia, así como la relación entre el orden

del filtro y la frecuencia de corte. Adicionalmente, se presenta el filtro Leaky, del mismo modo que sus semejanzas y diferencias con el filtro de promedio.

En los capítulos cuatro y cinco se abordan los métodos de diseño de filtros FIR e IIR, las gráficas de polos y ceros, de la misma manera que su relación con la frecuencia de corte y el tipo de filtro diseñado (pasa-bajo, pasa-alto, pasa-banda). En el último capítulo, se introduce al lector en conceptos básicos de procesamiento de imágenes, como tipos de imágenes (blanco-negro, escala de grises, e imágenes a color), modelos de color RGB y HSV, ecualización de imágenes, tipos de ruido en imágenes, filtros espaciales, convolución en imágenes, detección de bordes en imágenes y compresión de imágenes.

Esperamos que este libro sea de gran utilidad para aquellos estudiantes que deseen aprender métodos y técnicas de procesamiento digital de señales, utilizando Python, y que les permita resolver problemas reales de ingeniería, de esta era digital.

CAPÍTULO 1.
Del Mundo Análogo al Mundo Digital

En este capítulo encontrarás una breve introducción al procesamiento digital de señales, específicamente en relación con los conceptos de muestreo, cuantización y costo de almacenamiento/transmisión asociados al proceso de conversión análogo a digital (A/D).

Al finalizar el capítulo, deberás estar en capacidad de:

1. Explicar el concepto de muestreo de señales análogas/continuas.
2. Explicar el concepto de cuantización de muestras.
3. Seleccionar adecuadamente los parámetros de frecuencia de mues treo y bits de resolución en la conversión A/D, de acuerdo con el espectro de la señal y su comportamiento en el dominio del tiempo.
4. Explicar el costo de almacenamiento/transmisión del proceso de conversión A/D de señales continuas/análogas.
5. Explicar el efecto en frecuencia de muestrear una señal análoga/ continua.

El Procesamiento Digital de Señales es un conjunto de técnicas y métodos que permiten manipular una señal para obtener información de ella (patrones), o para modificarla o transformarla. Por ejemplo, la señal de voz es una señal análoga en tiempo continuo que contiene información de entonación, género del hablante, idioma, entre otros, que puede ser utilizada para identificar qué persona está pronunciando un mensaje o discurso. En este caso, el procesamiento digital de la señal se enfoca en identificar *patrones de voz* que permitan caracterizar al hablante, y compararlo con una base de datos previamente almacenada en el sistema. También, hoy en día encontramos dispositivos celulares que utilizan reconocimiento facial como medio para desbloquear el acceso al sistema, sustituyendo o reemplazando la opción clásica de clave numérica; por lo cual, el celular debe identificar *"características faciales"* que permitan corroborar si el rostro que está frente a la cámara es el autorizado para desbloquearlo.

Pero ¿cómo pasamos del mundo análogo/continuo al mundo digital/discreto? Gran parte de las señales que encontramos en la naturaleza son análogas (infinitos valores de amplitudes posibles) que se van actualizando a lo largo de la variable independiente, que típicamente es el tiempo (con infinitos valores de tiempo posibles), que deben ser transformadas antes de poder ser utilizadas por un sistema digital.

El proceso se conoce como conversión análogo-digital (ó A/D), el cual consiste en seleccionar un número finito de valores de tiempo en los que representaremos sus amplitudes en un número finito de bits. De tal forma que tanto la variable independiente (tiempo), como la variable dependiente (amplitud de la señal) son discretizados. En la Figura 1 encontrarás una gráfica ilustrativa de la conversión A/D.

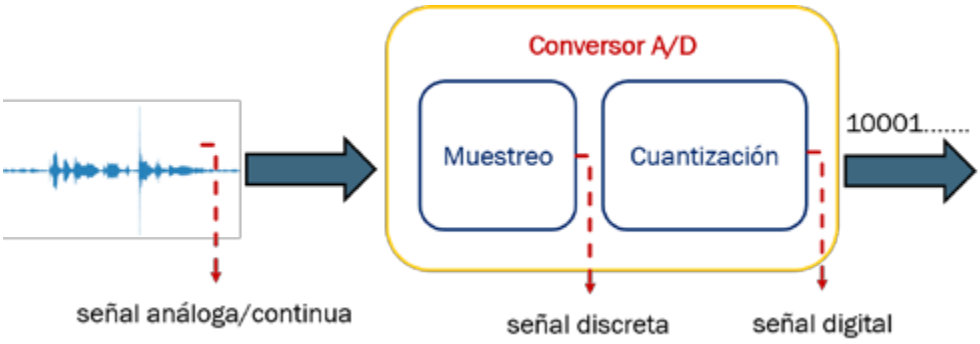


Figura 1. Diagrama general de un proceso de conversión A/D.

En las siguientes subsecciones encontrarás en detalle los conceptos de muestreo y cuantización, su costo de almacenamiento y transmisión, así como el efecto del muestreo en el espectro de la señal.

1.1. MUESTREO DE LA SEÑAL ANÁLOGA/CONTINUA

En la primera parte del proceso de conversión A/D, se selecciona un número de muestras por segundo de la señal, conocido como frecuencia de muestreo (f_s). De esta forma, si, por ejemplo, la señal tiene una duración de 10 segundos y la frecuencia de muestreo es de 8 kHz, entonces, la cantidad total de muestras es de 80.000. El valor de f_s, en el caso de muestreo equi-espaciado, debe satisfacer el criterio de Nyquist, el cual establece que:

$$f_s \geq 2 * f_{max} \qquad \text{Ecuación 1}$$

Donde f_{max} corresponde a la frecuencia máxima de la señal de tiempo continuo. Por ejemplo, si el espectro de nuestra señal tiene el comportamiento de la Figura 2, entonces la frecuencia de Nyquist es de 8 kHz. En otras palabras, una $f_s = 8\ kHz$ solo es adecuada para señales cuya $f_{max} = 4\ kHz$.

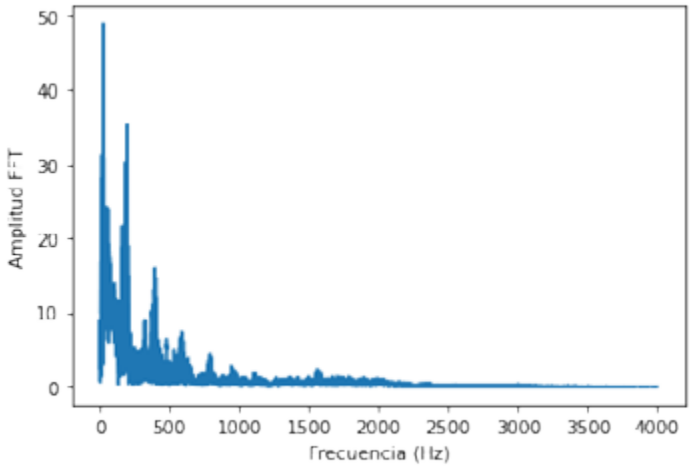


Figura 2. Ejemplo de espectro de señal de voz, f_s = 8 kHz.

La lectura y procesamiento de un archivo de voz (ej. en formato wav), está soportado en muchos lenguajes de programación. Para el caso del lenguaje Python, podemos utilizar la librería *Librosa* con el fin de cargar la señal en el entorno de ejecución (por ejemplo, en un *Jupyter* notebook como *CoLaboratory)*. Esta librería permitirá también visualizar la señal o conocer la f_s con la que fue muestreada.

Específicamente, en lenguaje Python escribimos el siguiente código para la visualización de la señal en el dominio del tiempo:

```
import librosa
import librosa.display
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import IPython
from scipy.io import wavfile
from scipy.fft import fftshift

plt.rcParams["figure.figsize"] = (14,5)
filename = 'audio.wav'

# Se debe asignar sr=None para que se conserve la fs original del audio.
# En caso contrario, se re-muestrea a 22050 Hz.
audio, fs = librosa.load(filename, sr=None)
librosa.display.waveplot(audio, sr=fs);

print("frecuencia de muestreo de la señal:", fs, "Hz")
print("cantidad de muestras de la señal:", len(audio))
```

Obteniendo como resultado:

```
frecuencia de muestreo de la señal: 8000 Hz
cantidad de muestras de la señal: 24000
```

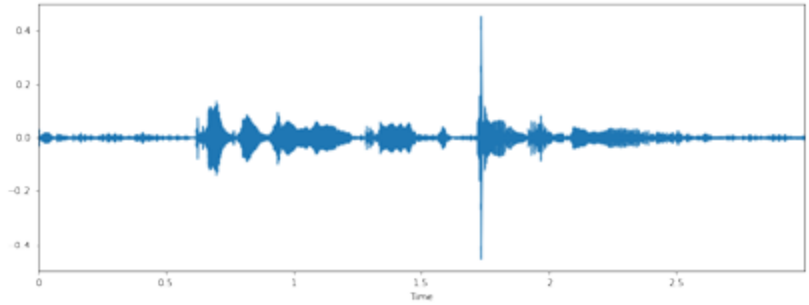


Figura 3. Ejemplo de señal de voz en el dominio del tiempo.

De acuerdo con la Figura 3, esta señal de voz tiene una duración de 3 segundos, y su amplitud se encuentra comprendida entre [-0.45 0.45], aproximadamente. Adicionalmente, en 1.7 segundos, se percibe un incremento significativo de la amplitud de la señal (tanto positiva como negativa) en relación con los demás valores de amplitud a lo largo de los 3 segundos. Teniendo en cuenta que la $f_s = 8\ kHz$, el total de muestras de la señal es de 24K.

Si queremos que el audio se ajuste al máximo volumen posible, podemos escalar su amplitud, así:

```
norm = max(np.absolute([min(audio), max(audio)]))
audio= audio /norm
librosa.display.waveplot(audio, sr=fs);
```

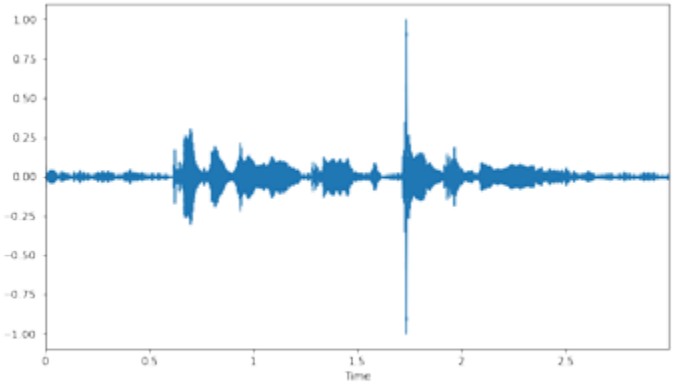


Figura 4. Ejemplo de señal de voz en el dominio del tiempo, con normalización de amplitud.

Esta nueva señal tiene una amplitud mayor a la señal original, y ahora se encuentra en el rango de [-1 1]. Adicionalmente, podemos reproducir el audio, con el siguiente código:

```
IPython.display.Audio(audio, rate=fs)
```

El cual genera un botón de reproducción

Posteriormente, es posible graficar el espectro de la señal con el siguiente código en Python:

```
import scipy.fftpack as  fourier

L=len(audio)
transformada = fourier.fft(audio)
magnitud = abs(transformada)
magnitud_lateral = magnitud[0:L//2]
fase = np.angle(transformada)
frecuencias = fs*np.arange(0, L//2)/L

plt.plot(frecuencias, magnitud_lateral)
plt.xlabel('Frecuencia (Hz)', fontsize='10')
plt.ylabel('Amplitud FFT', fontsize='10')
plt.show()
```

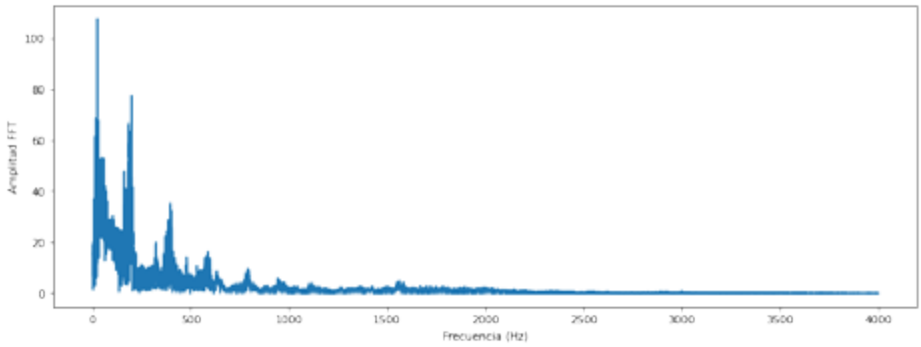


Figura 5. Espectro de la señal de voz de la Figura 4.

Pero ¿cómo sabemos si la frecuencia de muestreo de la señal en el proceso de conversión A/D fue adecuada? La respuesta la obtenemos en su espectro. Por ejemplo, para nuestro caso, las amplitudes de la FFT para frecuencias mayores de $2\ kHz$ son muy cercanas a cero y distan significativamente de las amplitudes en frecuencias inferiores a $1\ kHz$. De tal forma que, la mayor parte de la energía de la señal se encuentra en las frecuencias menores a 1kHz, y entonces $f_s = 8\ kHz$ es adecuada. Si por el contrario, en frecuencias cercanas a $4\ kHz$ las amplitudes de la FFT fuesen

comparativamente altas en relación con frecuencias menores, muy posiblemente la f_s seleccionada sería incorrecta, y tendríamos que escoger un valor mayor.

Supongamos que nuestra señal corresponde a un fragmento de música de un concierto de violín (Figura 6), cuyo espectro se presenta en la Figura 7.

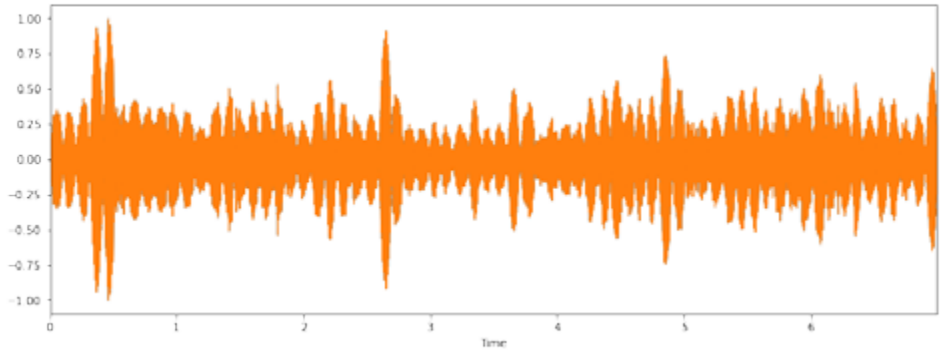


Figura 6. Ejemplo de señal de música en el dominio del tiempo.

A diferencia de la señal de voz, las amplitudes de la FFT cercanas a $4\ kHz$ no son significativamente pequeñas en relación con las amplitudes en frecuencias menores a $1\ kHz$, por lo que utilizar una $f_s > 8\ kHz$ es necesario, por ejemplo $f_s = 22\ kHz$.

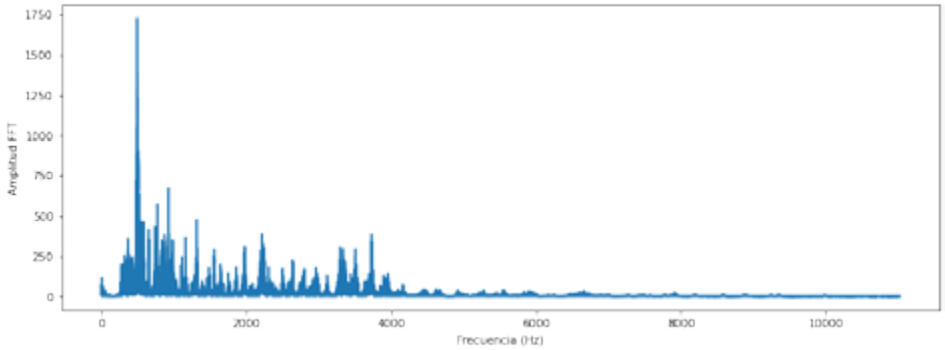


Figura 7. Espectro de la señal de música de la Figura 6.

Hasta aquí, hemos comprendido que no todas las señales necesitan la misma frecuencia de muestreo, y que a medida que la frecuencia máxima de la señal continua es mayor, debemos muestrear la señal con un número mayor de muestras por segundo. ¿Pero qué ocurriría si seleccionamos una f_s no adecuada, es decir que no cumpla el criterio de Nyquist? La respuesta se ilustrará a través de ejemplos.

Supongamos entonces que la señal de voz de la Figura 4 la re-muestreamos a $1\ kHz$, es decir, solamente conservaremos las componentes de frecuencias de los 0 Hz hasta los 500 Hz, como se presenta en la Figura 8.

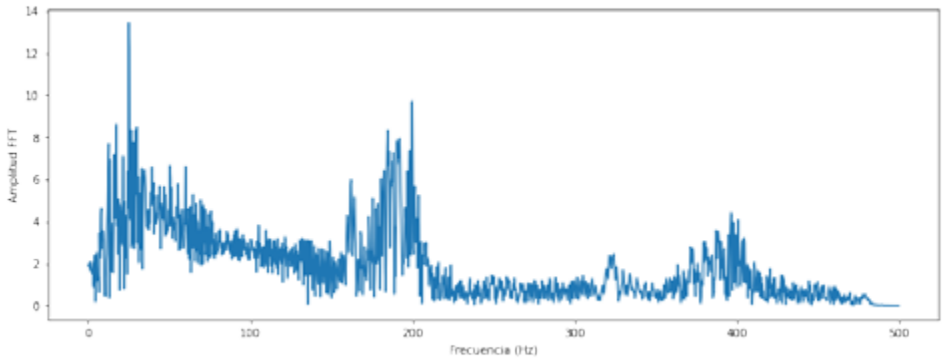


Figura 8. Espectro de la señal de voz de la Figura 4, re-muestreada a 1 kHz.

Esta nueva señal tiene el efecto de escucharse la voz ahogada, dado que, no cuenta con componentes de frecuencias altas, relacionadas con el detalle de la señal. Este fenómeno, el cual ocurre cuando la frecuencia de muestreo no es al menos el doble de la frecuencia máxima de la señal, se conoce como *aliasing*.

Finalmente, es necesario aclarar que, aunque en un dispositivo digital como un PC podemos graficar una señal con *apariencia de continua/análoga,* estas señales son en realidad *discretas/digitales*. Internamente, se realiza un proceso de interpolación que permite unir las amplitudes discretas para que luzcan como una señal que varía para valores infinitos de tiempo.

Específicamente en lenguaje Python, se puede utilizar la librería *Matplotlib* para graficar señales uni-dimensionales (1D), con dos opciones de visualización: *plot* para tiempo continuo, *stem* para tiempo discreto. A diferencia de la librería de *Librosa,* es necesario definir un vector de tiempos previo a la visualización.

A continuación, se presenta el código en Python para las dos formas de visualización.

```
t = np.arange(0,len(audio)/fs,1/fs)
plt.rcParams["figure.figsize"] = (14,8)
ax = plt.subplot(2, 1, 1)
plt.plot(t[8000:8100], audio[8000:8100])
plt.title("Gráfica señal de voz utilizando plt.plot")
ax = plt.subplot(2, 1, 2)
plt.stem(audio[8000:8100])
plt.title("Gráfica señal de voz utilizando plt.stem")
```

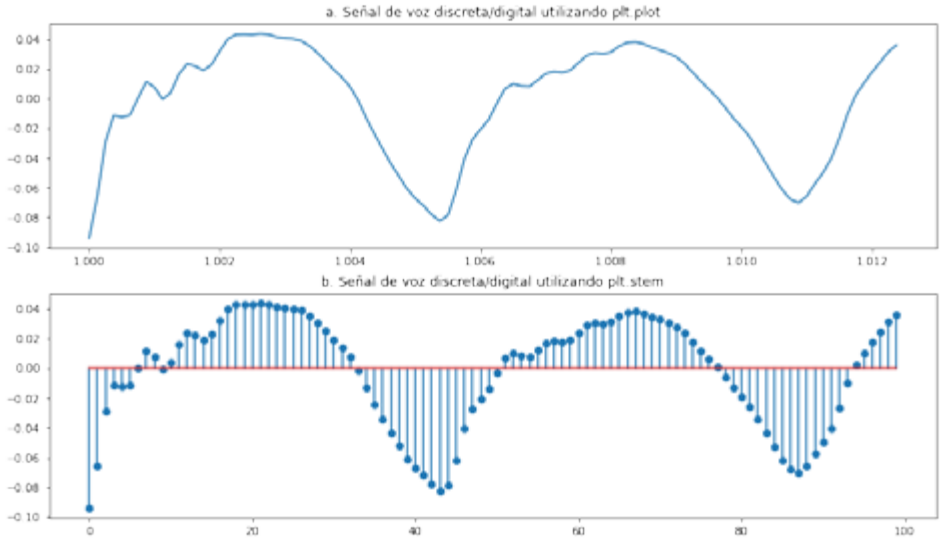


Figura 9. Ejemplo de señal de voz con dos formas distintas de visualización.

En la Figura 9a se graficaron 0.0125 segundos de la señal de voz de la Figura 4, comprendidos en el rango [1 1.0125) segundos, que corresponden a la interpolación de 100 muestras de la señal de voz en el rango [8000 8100). Aunque en este libro utilicemos en algunas ocasiones *plot* y en otras *stem*, el estudiante deberá siempre tener en cuenta que se están graficando señales discretas en el tiempo, con un número finito de bits de resolución.

1.2. CUANTIZACIÓN DE LAS MUESTRAS

Una vez se ha muestreado la señal, el siguiente paso (el cual se realiza casi de forma paralela en el conversor A/D) consiste *en representar mediante bits* a la amplitud de la señal discreta. Existen diversos formatos de representación de datos, por ejemplo, *magnitud, magnitud + signo, punto flotante,* entre otros. Supongamos que nuestro conversor trabaja con el formato *magnitud + signo,* donde el MSB *(Most Significant Bit: bit más significativo)* corresponde al signo del dato, y los restantes bits a la magnitud. De tal forma que, si el MSB es igual a 1, entonces la amplitud es negativa; en caso contrario, la amplitud es positiva.

Ahora bien, los conversores permiten trabajar con diferente número de bits de conversión por muestra, lo que se conoce como bits de resolución. A mayor cantidad de bits, la señal digital se escuchará más fiel a la señal análoga. Típicamente, podemos encontrar resoluciones de 8, 16, 24 y 32 bits.

Supongamos que nuestro audio que inicialmente se encontraba en el rango [-1 1] lo cuantizamos con 4 bits en formato *magnitud + signo*. Entonces, tenemos 3 bits

para la magnitud de la señal y 1 bit para el signo, de tal forma que cada incremento de amplitud de $1/2^3$ tendrá un nuevo código digital. Es decir, a todas las amplitudes del audio en el rango $[0 \quad 1/2^3)$ se les asigna el código 0000, a todas las amplitudes en el rango $[1/2^3 \quad 2/2^3)$ se les asigna el código 0001, y así sucesivamente. La Tabla 1 presenta la asignación de códigos por rangos de amplitud de la señal.

Tabla 1. Ejemplo de cuantización con 4 bits con formato magnitud + signo, para una señal en el rango [-1 1].

rango	código	Rango	código
$[-1 \quad -0.875)$	1111	$[0 \quad 0.125)$	0000
$[-0.875 \quad -0.75)$	1110	$[0.125 \quad 0.25)$	0001
$[-0.75 \quad -0.625)$	1101	$[0.25 \quad 0.375)$	0010
$[-0.625 \quad -0.5)$	1100	$[0.375 \quad 0.5)$	0011
$[-0.5 \quad -0.375)$	1011	$[0.5 \quad 0.625)$	0100
$[-0.375 \quad -0.25)$	1010	$[0.625 \quad 0.75)$	0101
$[-0.25 \quad -0.125)$	1001	$[0.75 \quad 0.875)$	0110
$[-0.125 \quad 0)$	1000	$[0.875 \quad 1)$	0111

De acuerdo con la Tabla 1, dos amplitudes que solo se diferencien en el signo (ej. -0.6 y 0.6) tendrán el mismo código excepto en su MSB (en este caso, 1100 y 0100). A medida que aumenta el número de bits de resolución, el rango de amplitudes que comparte el mismo código se va haciendo más pequeño. Por ejemplo, si la señal con rango análogo de [-1 1] la cuantizamos a 16 bits (15 de magnitud y 1 de signo), cada $1/2^{15}$ (es decir $30.5*10^{-6}$) tendrá un código digital distinto.

Para ilustrar el impacto de la cantidad de bits de resolución, utilicemos las mismas señales discretas de la sección anterior, la señal de voz y la de audio, para ilustrar el impacto de los bits de resolución en la calidad de la señal digital/discreta. El archivo audio.wav tiene 16 bits de resolución. Vamos a re-cuantizarlo a 8 bits (Figura 10), con el siguiente código en Python:

```
bits = 8
audio_8bit = (audio* 2**bits).astype(int)
audio_8bit = audio_8bit / 2**bits
librosa.display.waveplot(audio_8bit, sr=fs)
```

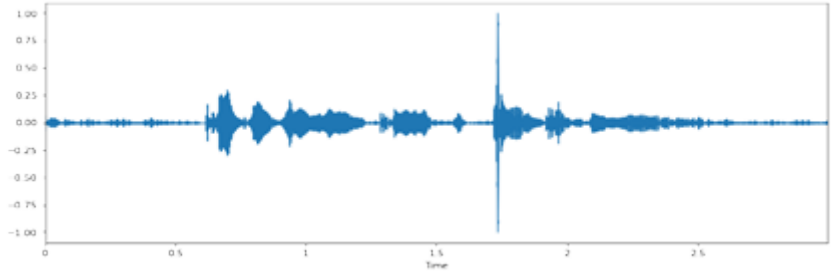


Figura 10. Ejemplo de señal de voz cuantizada a 8-bits.

y reproducimos la señal, así:

```
IPython.display.Audio(audio_8bit, rate=fs)
```

El efecto es que escuchamos *ruido de fondo* en la señal, pero el mensaje seguirá siendo legible. Ahora, disminuiremos la resolución a 6 bits (Figura 11), y compararemos los resultados con los obtenidos previamente.

```
bits = 6
audio_6bit = (audio* 2**bits).astype(int)
audio_6bit = audio_6bit / 2**bits
librosa.display.waveplot(audio_6bit, sr=fs);
IPython.display.Audio(audio_6bit, rate=fs)
```

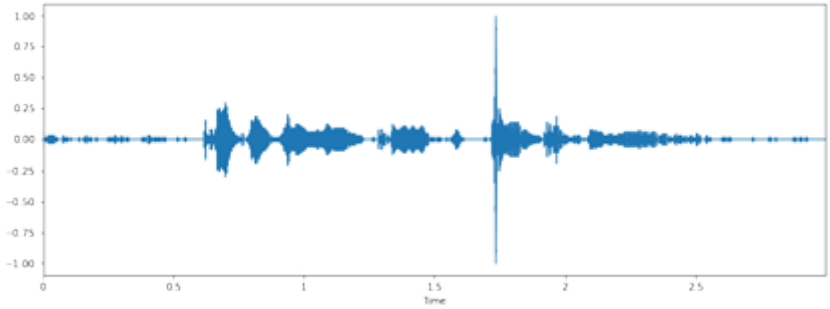


Figura 11. Ejemplo de señal de voz cuantizada a 6-bits.

En el audio re-cuantizado a 6 bits se escuchan *saltos de amplitud* en el mensaje. La calidad del audio en términos de legibilidad ha disminuido.

Finalmente, se re-cuantiza el audio a 3 bits (Figura 12). A diferencia de los casos anteriores, la señal re-cuantizada no tiene contenido inteligible (es decir, no se entiende lo que se dice), dado que la amplitud dista significativamente de la señal original cuantizada a 16 bits (Figura 4).

```
bits = 3
audio_3bit = (audio* 2**bits).astype(int)
audio_3bit = audio_3bit / 2**bits
librosa.display.waveplot(audio_3bit, sr=fs)
IPython.display.Audio(audio_3bit, rate=fs)
```

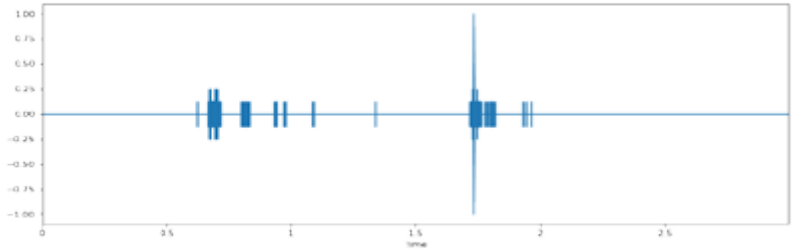


Figura 12. Ejemplo de señal de voz cuantizada a 3-bits.

Posteriormente, seleccionamos el archivo music.wav el cual tiene también 16 bits de resolución. Aplicaremos dos re-cuantizaciones: de 8 bits y de 3 bits.

```
# Re-cuantización a 8 bits del registro de música
bits = 8
music_8bit = (music* 2**bits).astype(int)
music_8bit = music_8bit / 2**bits
fig, ax = plt.subplots(nrows=2, sharex=True, sharey=True)
librosa.display.waveplot(music_8bit, sr=fs2, ax=ax[0])
ax[0].set(title='Música re-cuantizada a 8 bits')
ax[0].label_outer()

# Re-cuantización a 3 bits del registro de música
bits = 3
music_3bit = (music* 2**bits).astype(int)
music_3bit = music_3bit / 2**bits
librosa.display.waveplot(music_3bit, sr=fs2, ax=ax[1])
ax[1].set(title='Música re-cuantizada a 3 bits')
ax[1].label_outer()
```

Con la re-cuantización a 8 bits (Figura 13a), la señal es muy similar a la original cuantizada a 16 bits (Figura 6); mientas que la re-cuantizada a 3 bits (Figura 13b), tanto gráficamente como de forma auditiva, se aleja de la señal original.

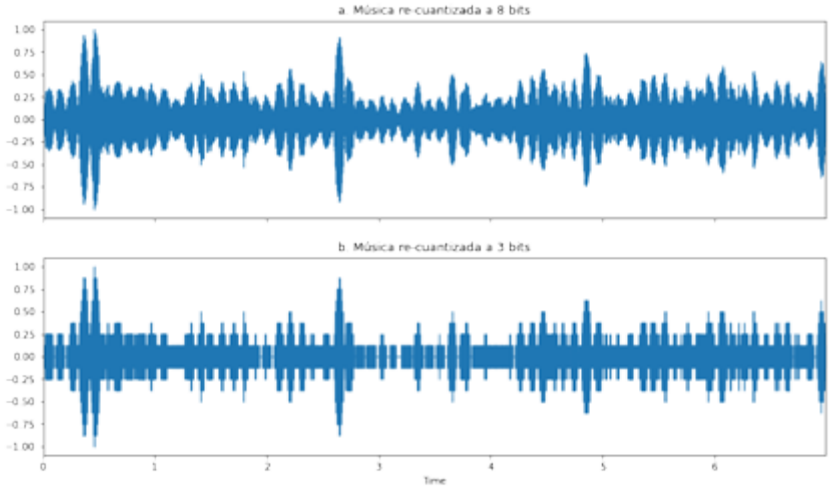


Figura 13. Ejemplo del efecto de re-cuantización de la señal de música a 8 bits y a 3 bits.

Por lo anterior, es evidente que la selección de la cantidad de bits de resolución juega un papel muy importante en la calidad de la señal discreta/digital. En la siguiente subsección abordaremos las implicaciones que tiene a nivel de costo de almacenamiento y de transmisión el valor de bits de resolución seleccionado.

1.3. COSTO DE ALMACENAMIENTO/TRANSMISIÓN EN TÉRMINOS DE LA FRECUENCIA DE MUESTREO Y NÚMERO DE BITS DE RESOLUCIÓN

Hasta aquí hemos evidenciado la importancia de seleccionar adecuadamente el valor de frecuencia de muestreo y de bits de resolución cuando vamos a convertir una señal continua/análoga en discreta/digital. Recordemos que la notación *"continua"* y *"discreta"* hace alusión a la variable independiente (típicamente el tiempo), mientras que *"análoga"* y *"digital"* corresponde con la amplitud de la señal (ej. voltios., amperes, entre otros). Si la cantidad de valores en un rango de tiempo es infinita, la señal es continua; en caso contrario, es discreta. De forma similar, si la cantidad de valores diferentes de amplitud en un rango es infinita, la señal es análoga; en caso contrario, es digital.

Aunque podríamos pensar que tanto la f_s seleccionada como los bits de resolución deberían ser los más altos posibles en beneficio de la calidad de la señal (similitud con la señal continua/análoga original), debemos tener presente que existe un costo asociado con el almacenamiento y la transmisión de la señal. Este concepto lo explicaremos a través de dos casos.

Caso 1:

Supongamos que se ha digitalizado una señal de voz de 2.777 horas (exactamente 10.000 segundos), con $f_s = 24$ kHz y 32 bits de resolución. Entonces, la cantidad de bits total de la señal digital/discreta es:

$$cantidad = Time * f_s * res \ [bits] \qquad \textbf{Ecuación 2}$$

Donde Time es la duración de la señal en segundos, f_s es la frecuencia de muestreo, y res es la cantidad de bits de resolución.

De tal forma que, $cantidad$ = 10.000 $*$ 24.000 $*$ 32 = 7.68 Gb, que corresponde a 960 MB.

Caso 2:

La misma señal de voz del Caso 1 se digitalizó con $f_s = 8$ kHz y 16 bits de resolución. Entonces, $cantidad$ = 10.000 $*$ 8.000 $*$16 = 1.28 Gb, que equivale a 160 MB.

Supongamos ahora que nuestro plan de Wi-Fi es de 10 MBps (donde Bps: bytes por segundo) y queremos descargar la señal de voz que se encuentra en dos páginas de internet. La primera página utilizó los parámetros de conversión del Caso 1; mientras que la segunda página utilizó los parámetros de conversión del Caso 2. Enton-

ces, la descarga del archivo en la primera página de internet tomaría 96 segundos (1 minuto y 36 segundos), mientras que, en la segunda página de internet tomaría 16 segundos. Es evidente que preferiríamos descargar el archivo de la segunda página de internet porque nos tomaría la sexta parte en relación con el tiempo de descarga en la primera página.

Pero ¿la calidad de la señal discreta/digital obtenida con los parámetros de conversión del Caso 2 es lo suficientemente buena? La respuesta es sí, dado que, tanto la f_s como la resolución son adecuados para señales de voz. No es necesario discretizar una señal que solo contiene voz con una $f_s = 24\ kHz$, dado que, como vimos previamente, la mayor parte de la energía de la señal se encuentra en las frecuencias inferiores a $1\ kHz$. Adicionalmente, la resolución de 16 bits permite cambios en el código digital para valores de amplitud muy pequeños.

Como conclusión, los valores de f_s y bits de resolución no deberían ser tan pequeños que nos degraden la calidad de la señal, pero tampoco excesivamente altos, que impliquen altos costos de almacenamiento y/o transmisión de la señal.

1.4. EFECTO EN EL ESPECTRO DE MUESTREAR UNA SEÑAL DE TIEMPO CONTINUO

En el canal de YouTube[1] podrás encontrar el video titulado *"Espectro señales discretizadas"* en el que se explica paso a paso el efecto de muestrear una señal continua en términos de su espectro. Este concepto lo explicaré de forma matemática a continuación.

Primero, partimos de una señal continua en el dominio del tiempo, la cual posee un número infinito de valores de tiempo en el rango de $[t_i \quad t_f]$, donde t_i es el tiempo inicial de la señal, y t_f es el tiempo final. Por ejemplo, supongamos que $t_i = 0\ s$, mientras que $t_f = 10\ s$. Esa señal la vamos a denominar $x(t)$ y su espectro $X(f)$.

Es decir,

$$x(t) \overset{FT}{\rightarrow} X(f) \qquad \textbf{Ecuación 3}$$

Supongamos que el espectro de la señal $x(t)$ está comprendido en el rango $[0 \quad 4]kHz$, por lo que decidimos muestrear la señal con f_s = 8 kHz. La forma de hacerlo es multiplicar $x(t)$ con un tren de impulsos periódico de amplitud igual a 1 y $T = 1/f_s$, que denominaremos $m(t)$. En nuestro caso, el periodo del tren de impulsos es $T=1/8kHz=125\ \mu s$. El espectro de m(t) lo denominaremos $M(f)$, el cual corresponde a otro tren de impulsos cuya amplitud es $1/T$ y espaciado cada f_s.

1 https://www.youtube.com/channel/UCrasAFtm_6B9vOIShGtI1ig

Es decir,

$$m(t) \overset{FT}{\rightarrow} FT\ M(f) \qquad \text{Ecuación 4}$$

El efecto en el dominio del tiempo de multiplicar $x(t)$ con $m(t)$, es que la señal continua queda muestreada cada T segundos, obteniendo una señal discreta que denominaremos $x[n]$. En el dominio de la frecuencia, el efecto es la convolución entre los espectros de $X(f)$ y $M(f)$, es decir, se generan "réplicas" del espectro $X(f)$ cada f_s Hz. A este espectro resultante lo denominaremos $X_m(f)$, así:

$$x[n] \overset{DFT}{\rightarrow} X_m(f) \qquad \text{Ecuación 5}$$

El efecto de réplicas en el espectro se explica recordando que cuando se convoluciona una señal por un impulso desplazado en k, el resultado es la misma señal desplazada en k. De tal forma que la convolución de $X(f)$ con el impulso ubicado en el origen es el mismo espectro $X(f)$; la convolución de $X(f)$ con el impulso ubicado en f_s es $X(f - f_s)$; la convolución de $X(f)$ con el impulso ubicado en $2\ f_s$ es $X(f - 2\ f_s)$; y así sucesivamente. Teniendo en cuenta que la señal $m(t)$ contiene infinitos impulsos separados f_s, entonces, la cantidad de réplicas de $X(f)$ es también infinita y están separadas f_s. Adicionalmente, su amplitud se verá afectada por el valor $1/T$.

Hasta aquí, vamos a resumir lo explicado anteriormente:

Tabla 2. Muestreo con tren de impulsos de duración infinita y su efecto en frecuencia.

Dominio del Tiempo (continuo o discreto)	Dominio de la Frecuencia
$x(t)$	$X(f)$
$m(t) = \sum_{k=-\infty}^{\infty} \delta(t - kT), k \in \mathbb{Z}$	$M(f) = \frac{1}{T} \sum_{k=-\infty}^{\infty} \delta(t - kf_s), k \in \mathbb{Z}$
$\mathbb{Z}$ representa el conjunto de números enteros	$\mathbb{Z}$ representa el conjunto de números enteros
$x[n] = x(t) * m(t)$	$X_m(f) = X(f) \otimes M(f)$
$*$ significa multiplicación (efecto: señal muestreada cada T segundos)	$\otimes$ significa convolución (efecto: réplicas del espectro $X(f)$ cada f_s)

Ahora bien, teniendo en cuenta que en la práctica el tren de impulsos es de duración finita, podemos multiplicar $m(t)$ por una ventana $w[n]$, para limitar la duración del tren de impulsos en el rango $[t_i \quad t_f]$. Entonces, en el dominio de la frecuencia, el espectro del tren de impulsos, $M(f)$, se convoluciona con el espectro de la ventana, $W(f)$.

Dado que existen diversos tipos de ventana y que cada una tiene un espectro diferente, se expresará de forma general, tanto la señal en el dominio del tiempo, como en el dominio de la frecuencia, así:

$$w[n] \overset{DFT}{\rightarrow} W(f) \qquad \text{Ecuación 6}$$

En la Tabla 3 se presenta el efecto de muestreo de la señal $x(t)$ con el tren de impulsos de duración finita. Al final de todo este proceso, el espectro de la señal $x(t)$ no solamente se replica, sino que se distorsiona ligeramente, debido a la convolución en el dominio de la frecuencia entre $(X(f) \otimes M(f))$ con $W(f)$.

Antes de transmitir la señal muestreada, se aplica un filtro pasa-bajo, para obtener únicamente la réplica ubicada en el origen.

Tabla 3. Muestreo con tren de impulsos de duración finita y su efecto en frecuencia.

Dominio del Tiempo	Dominio de la Frecuencia
$x(t)$	$X(f)$
$m(t) = \sum_{k=-\infty}^{\infty} \delta(t - kT), k \in \mathbb{Z}$	$M(f) = \frac{1}{T} \sum_{k=-\infty}^{\infty} \delta(t - kf_s), k \in \mathbb{Z}$
$w[n]$	$W(f)$
$x[n] = x(t) * m(t) * w[n]$	$X_m(f) = X(f) \otimes M(f) \otimes W(f)$
$*$ *significa multiplicación (efecto: señal muestreada cada T)*	$\otimes$ *significa convolución (efecto: réplicas del espectro $X(f)$ cada f_s con una ligera distorsión)*

CAPÍTULO 2.
Notación en el Mundo discreto

Después de abordar el puente entre el mundo análogo/continuo con el mundo digital/discreto, nos introduciremos en la notación que se utiliza en procesamiento digital de señales.

Al finalizar el capítulo, deberás estar en capacidad de:

1. Expresar en el dominio Z una señal en tiempo discreto de duración finita.
2. Convertir una ecuación de entrada-salida en una función de transferencia en el dominio Z.
3. Dibujar en diagramas de bloques un sistema FIR.
4. Identificar si un filtro digital es FIR o IIR a partir de la ecuación de entra da-salida, la respuesta al impulso, o la función de transferencia del sistema.

En muchos libros de Procesamiento Digital de señales encontrarás como tema infaltable la Transformada Z (conocida como TZ) con su matemática, y ecuaciones, y de pronto pensarás que es un tema muy difícil de abordar. Pues estás equivocado, la TZ es una representación muy amigable de las señales discretas, la cual nos ayuda a modelar el comportamiento de un sistema discreto a través de su entrada y salida. De una forma muy intuitiva vamos poco a poco a conocer en qué consiste la TZ y cómo nos apoya en la representación y diseño de los filtros digitales para señales 1D.

2.1. REPRESENTACIÓN DE UNA SEÑAL DISCRETA EN TÉRMINOS DE IMPULSOS DESPLAZADOS Y NOTACIÓN Z

Partamos de la señal $x[n]$ de la Figura 14, la cual contiene 10 muestras comprendidas en el rango [-4 5]. Recuerda que en el dominio discreto no hablamos de segundos, sino de muestras, y que solamente existen valores de n enteros.

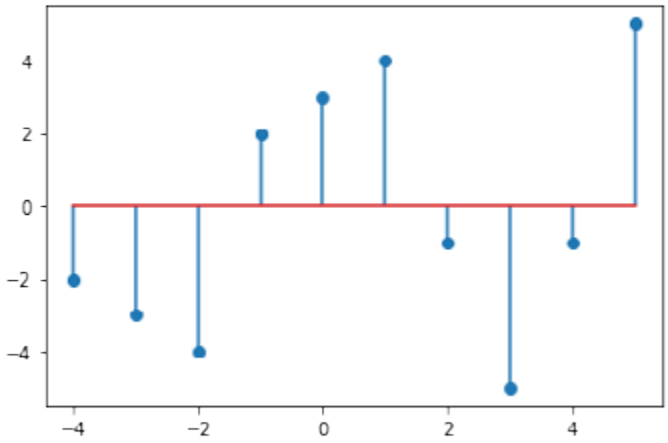


Figura 14. Ejemplo de señal en el dominio discreto.

Esta señal la podemos dibujar en lenguaje Python, con el siguiente código:

```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
n = np.linspace(-4,5, 10)
print(n)
x = np.array([-2, -3, -4, 2, 3, 4, -1, -5, -1, 5])
print(x)
plt.stem(n,x, use_line_collection="True")
```

Podemos representar esta señal de varias formas, por ejemplo:

x(-4) = -2
x(-3) = -3
x(-2) = -4
x(-1) = 2
x(0) = 3
x(1) = 4
x(2) =- 1
x(3) =- 5
x(4) =- 1
x(5) = 5

En términos de impulsos desplazados, así:

$x[n] = -2\delta[n+4] - 3\delta[n+3] - 4\delta[n+2] + 2\delta[n+1] + 3\delta[n] + 4\delta[n-1] - \delta[n-2] - 5[n-3] - \delta[n-4] + 5\delta[n-5]$

Y de forma compacta, así:

$$x[n] = \sum_{k=-4}^{5} a_k \delta[n-k]$$

Con $a_{-4} = -2, a_{-3} = -3, a_{-2} = -4, a_{-1} = 2, a_0 = 3, a_1 = 4, a_2 = -1, a_3 = -5, a_4 = -1, a_5 = 5.$

Supongamos que ahora transformamos la señal al dominio Z, es decir que:

$$x[n] \xrightarrow{TZ} X(z)\ ) \qquad \text{Ecuación 7}$$

Obteniendo para esta señal:

$$X(z)=-2z^4 - 3z^3 - 4z^2 + 2z^1 + 3z^0 + 4z^{-1} - z^{-2} - 5z^{-3} - z^{-4} + 5z^{-5}$$

¿Qué similitudes encuentras entre x[n] con X(z)?

Resuelve esta pregunta antes de leer la respuesta que se encuentra a continuación.

Podemos observar que:

a) Los impulsos que se encuentran ubicados a la izquierda del origen, su *TZ* corresponde a una potencia positiva de z.
b) Los impulsos que se encuentran ubicados a la derecha del origen, su *TZ* corresponde a una potencia negativa de z.
c) Las amplitudes y signos de los impulsos se conservan.
d) La *TZ* del impulso ubicado en el origen corresponde a la amplitud del impulso (dado que $z^0=1$).

De forma intuitiva hemos llegado a la ecuación que relaciona el dominio discreto con el dominio z, así:

$$X(z) \equiv \sum_{k=-\infty}^{\infty} x(k)\, z^{-k} \qquad \text{Ecuación 8}$$

Donde $x(k)$ es la amplitud de la señal para $n=k$, mientras que z es una variable compleja con la cual se transforma del dominio del tiempo al dominio de la frecuencia. Para señales discretas, podemos decir que si la Transformada de Fourier existe, entonces su resultado coincide con la *TZ* de la señal haciendo $z=e^{j\omega}$, es decir, para $|z|=1$.

2.2. MODELANDO SISTEMAS DISCRETOS

Un sistema discreto es aquel en el que tanto la señal de entrada, como la de salida, son discretas, es decir, que tienen un número finito de muestras en un rango (de tiempo) seleccionado. Estos sistemas se pueden representar por ecuaciones de entrada-salida, funciones de transferencia y diagramas de bloques.

Supongamos que tenemos un sistema discreto con la siguiente ecuación de entrada-salida:

$$y[n]=\frac{x[n] + x[n - 1] + x[n - 2]}{3}$$

Entonces, para calcular la salida en el tiempo discreto actual necesitamos conocer la entrada en el mismo tiempo discreto actual, la entrada en el tiempo discreto anterior, y la entrada en dos tiempos discretos anteriores. Posteriormente, se promedian esos tres valores.

Para reescribir la ecuación de entrada-salida en el dominio *Z,* es necesario que conozcamos el efecto de un retardo de la señal en el dominio temporal.

Específicamente,

si $x[n] \overset{TZ}{\rightarrow} X(z), \Rightarrow x[n-1] \overset{TZ}{\rightarrow} z^{-1} X(z)$ Ecuación 9

Y de forma general,

$$x[n-k] \overset{TZ}{\rightarrow} z^{-k} X(z)$$ Ecuación 10

Entonces, $x[n-2] \overset{TZ}{\rightarrow} z^{-2} X(z)$.

Aplicando el concepto anterior, reescribimos la ecuación de entrada-salida del sistema en el dominio *Z*, así:

$$Y(z)=\frac{X(z) + z^{-1} X(z) + z^{-2} X(z)}{3}$$

Factorizamos el término X(z), y lo pasamos a dividir al lado izquierdo de la ecuación, obteniendo:

$$\frac{Y(z)}{X(z)} = \frac{1 + z^{-1} + z^{-2}}{3}$$

El resultado anterior se conoce como la Función de Transferencia del sistema, $H(z)$.

Te preguntarás si existe alguna relación entre $H(z)$ y $h[n]$. La respuesta es que sí. Específicamente, $H(z)$ es la TZ de la respuesta al impulso del sistema, es decir,

$$h[n] \overset{TZ}{\rightarrow} H(z) \equiv \frac{Y(z)}{X(z)}$$ Ecuación 11

Recuerda que el operador "≡" significa *"por definición es igual a".*

Entonces, un sistema LTI (Lineal e Invariante en el Tiempo) se puede caracterizar tanto por $h[n]$ en el dominio del tiempo discreto, como por $H(z)$ en el dominio de Z.

Finalmente, nos queda la representación del sistema por diagrama de bloques. Para ello, dibujamos nuestra ecuación de entrada-salida, partiendo de *x[n],* incluyendo bloques de retardo (es decir bloques z^{-1}), sumadores, y amplificadores (o atenuadores), para finalmente obtener *y[n]*.

Para el sistema que hemos utilizado en esta sección, su diagrama de bloques es:

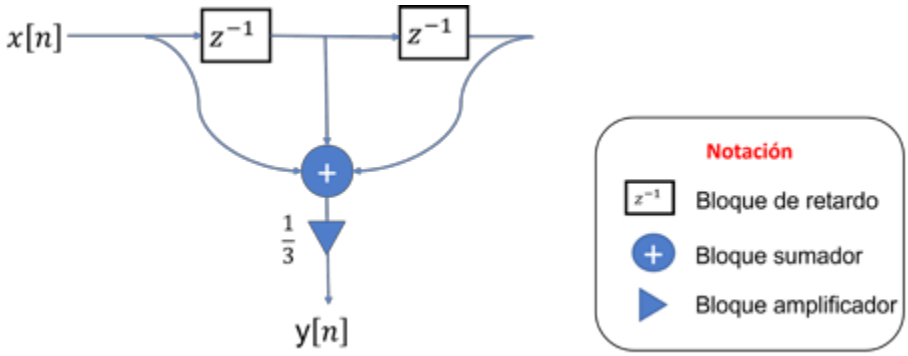


Figura 15. Ejemplo de diagrama de bloques de un sistema discreto.

En este caso, dado que tanto *x[n], x[n-1]*, como *x[n-2],* están ponderados por el mismo escalar, entonces, el bloque de amplificación se ubica después del sumador. En otros casos en los que cada término tenga su propia ponderación, es necesario incluir un bloque de amplificación por término de la ecuación, previo al bloque sumador.

2.3. INTRODUCCIÓN A LOS SISTEMAS DISCRETOS FIR vs IIR

En esta sección nos centraremos en la diferencia que existe entre los filtros FIR y los filtros IIR, de acuerdo con su respuesta al impulso.

Para ello, vamos a utilizar cuatro casos.

Caso 1:

Nuestro sistema discreto tiene la siguiente relación entrada-salida:

$$y[n] = \sum_{k=-2}^{2} a_k x[n-k]$$

Donde a_k es un escalar, y k está comprendida entre [-2 2]. Este sistema contiene cinco términos, los cuales son x[n + 2], x[n + 1], x[n], x[n - 1] y x[n - 2] .

Vamos a reescribir la ecuación en el dominio Z, así:

$$Y(z) = \sum_{k=-2}^{2} a_k z^{-k} X(z)$$

Y ahora pasamos a dividir $X(z)$ a la parte izquierda de la ecuación, obteniendo:

$$\frac{Y(z)}{X(z)} = \sum_{k=-2}^{2} a_k z^{-k}$$

Expandamos los términos de la función de transferencia, así:

$$H(z) = a_{-2}z^2 + a_{-1}z^1 + a_0 z^0 + a_1 z^{-1} + a_2 z^{-2}$$

¿Qué señal en el dominio del tiempo discreto tiene como transformada Z el valor de $H(z)$ que acabamos de encontrar?

La respuesta es,

$$h[n] = a_{-2}\delta[n+2] + a_{-1}\delta[n+1] + a_0\delta[n] + a_1\delta[n-1] + a_2\delta[n-2]$$

Teniendo en cuenta que la cantidad de impulsos es finita, este sistema es FIR (Finite Impulse Response). Adicionalmente, es simétrico (espejo) respecto al origen y no causal. Por lo cual, la salida depende de la señal de entrada en valores futuros, y no se puede trabajar en tiempo real.

Caso 2:

La relación entrada-salida del sistema, es:

$$y[n] = \sum_{k=0}^{4} a_k x[n-k]$$

Donde a_k es un escalar, para los valores de k comprendidos entre $[0 \quad 4)]$. Este sistema, al igual que el del Caso 1, contiene cinco términos (pero ahora desde $x[n]$ hasta $x[n-4]$).

En el dominio Z, la ecuación queda expresada de la siguiente forma:

$$Y(z) = \sum_{k=0}^{4} a_k z^{-k} X(z)$$

Y su función de transferencia, así:

$$H(z) = \sum_{k=0}^{4} a_k z^{-k} = a_0 z^0 + a_1 z^{-1} + a_2 z^{-2} + a_3 z^{-3} + a_4 z^{-4}$$

En el dominio del tiempo, la respuesta al impulso es:

$$h[n] = a_0\,\delta[n] + a_1\,\delta[n-1] + a_2\,\delta[n-2] + a_3\,\delta[n-3] + a_4\,\delta[n-4]$$

De forma similar al Caso 1, el sistema es FIR. No obstante, para este ejemplo la respuesta al impulso no está centrada en el origen, sino que inicia en $n=0$. Entonces, es un sistema causal y la salida del sistema se puede calcular en tiempo real.

Caso 3:

En este tercer caso, la relación entrada-salida del sistema, es:

$$y[n] = \sum_{k=0}^{\infty} x[n-k]$$

A diferencia de los ejemplos anteriores, la cantidad de términos a la derecha de la igualdad no es finita. Tenemos x[n], x[n-1], x[n-2], y así sucesivamente hasta x[n-∞].

En el dominio Z, la ecuación queda así:

$$Y(z) = \sum_{k=0}^{\infty} z^{-k} X(z)$$

Y su función de transferencia, es:

$$H(z) = \sum_{k=0}^{\infty} z^{-k} = z^0 + z^{-1} + z^{-2} + \cdots + z^{-\infty}$$

En el dominio del tiempo, la respuesta al impulso es:

$$h[n] = \delta[n] + \delta[n-1] + \delta[n-2] + \cdots + \delta[n-\infty]$$

Como la cantidad de impulsos en $h[n]$ es infinita, este sistema es IIR (Infinite Impulse Response). Por otro lado, como todos los impulsos se ubican a la derecha del origen (o en el origen), entonces el sistema es causal. Teniendo en cuenta que el diagrama de bloques requeriría de un número infinito de términos de retardo y de multiplicadores (si cada impulso tuviese una amplitud diferente), entonces, es común que se reescriba el sistema, como se expresa en el siguiente caso.

Caso 4:

La relación entrada-salida del sistema se define por la ecuación:

$$y[n] = x[n] + y[n-1]$$

Para obtener la señal de salida, es necesario conocer la señal de entrada en el mismo instante de tiempo discreto, y la señal de salida un instante anterior.

Este sistema es el mismo presentado en el Caso 3, como verificaremos a continuación.

Partiendo de,

$$y[n] = x[n] + x[n-1] + x[n-2] + x[n-3] + \cdots + x[n-\infty]$$

Al retardar en una posición todos los términos de la ecuación anterior, tendremos:

$$y[n-1] = x[n-1] + x[n-2] + x[n-3] + x[n-4] + \cdots + x[n-\infty]$$

Por lo cual, podremos re-escribir $y[n]$ así:

$$y[n] = x[n] + \underbrace{x[n-1] + x[n-2] + x[n-3] + \cdots + x[n-\infty]}_{y[n-1]} = x[n] + y[n-1]$$

Entonces, hemos verificado que nuestro sistema del Caso 4 es el mismo sistema del Caso 3. Por lo tanto, es IIR y causal.

Por otro lado, en el dominio Z obtenemos que:

$$Y(z) = X(z) + z^{-1} Y(z)$$

Y reordenando el resultado anterior:

$$Y(z) - z^{-1} Y(z) = X(z),$$

$$Y(z)\{1 - z^{-1}\} = X(z)$$

Llegamos a la función de transferencia del sistema:

$$\frac{Y(z)}{X(z)} = H(z) = \frac{1}{1-z^{-1}}$$

De forma general, si en la ecuación de entrada-salida existe algún término a la derecha de la ecuación de la forma $y[n-k]$, para k entero positivo o negativo, entonces el sistema es IIR. También es IIR si se necesita conocer infinitos valores de la señal de entrada, de la forma $x[n-k]$. Por otro lado, en términos de causalidad, si es necesario conocer valores pasados y/o presentes de la entrada y/o salida, entonces el sistema es causal; en caso contrario es no causal. Por lo cual, al unir los conceptos anteriores, tenemos que el sistema $y[n] = 0.9x[n] + 0.1y[n-2]$ es IIR causal, mientras que, el sistema $y[n] = 0.9x[n] + 0.1y[n+2]$ es IIR no causal.

Adicionalmente, en términos de la función de transferencia $H(z)$, si solamente tiene un polinomio en el numerador (dependiente de z) y el número de términos es finito, entonces el filtro es FIR. En caso contrario, el filtro es IIR. Si los polinomios solamente tienen términos de z negativos, el filtro es causal; en caso contrario, es no causal.

CAPÍTULO 3.
Mis primeros filtros digitales

Sé que en este punto del libro ya querrás conocer ejemplos concretos de filtros digitales y su efecto en señales 1D (uni-dimensionales). Este Capítulo está diseñado precisamente para que empieces a filtrar señales 1D con filtros muy sencillos, conocidos como filtros de promedio. Adicionalmente, conocerás su contraparte IIR denominada Integrador Leaky.

Al finalizar el capítulo, deberás estar en capacidad de:

1. Diseñar filtros pasa-bajos para señales 1D, específicamente filtros de promedio.
2. Filtrar señales 1D con filtros de promedio.
3. Explicar el comportamiento en frecuencia de los filtros de promedio, tanto para valores de M par como impar.
4. Explicar las diferencias entre el filtro de promedio y el filtro Integrador Leaky.

3.1. INTRODUCCIÓN AL FILTRO DE PROMEDIO

Para entender en qué consiste el filtro de promedio, es necesario que previamente recordemos cómo se caracteriza un sistema Lineal e Invariante en el Tiempo (LTI). Específicamente, la salida del sistema, $y[n]$, se encuentra calculando la convolución entre la señal de entrada, $x[n]$, y la respuesta al impulso, $h[n]$.

Es decir, si el sistema es LTI, se cumple que:

$$y[n] = x[n] \otimes h[n] = \sum_{k=-\infty}^{\infty} x[k]h[n - k] \qquad \text{Ecuación 12}$$

Adicionalmente, es necesario recordar en qué consiste convolucionar x[n] con un impulso ubicado en el origen, o desplazado, por ejemplo:

$$x[n] \otimes \delta[n] = x[n],$$

$$x[n] \otimes \delta[n - 1] = x[n - 1],$$

$$x[n] \otimes \delta[n - 2] = x[n - 2],$$

$$x[n] \otimes \delta[n+1] = x[n+1],$$

$$x[n] \otimes \delta[n+2] = x[n+2]$$

Entonces,

$$x[n] \otimes \delta[n-k] = x[n-k] \quad k \in Z \qquad \text{Ecuación 13}$$

Si unimos el concepto de la Ecuación 10 con el de la Ecuación 11, podremos identificar que si $y[n] = x[n-k]$, entonces su respuesta al impulso es $h[n] = \delta[n-k]$.

De forma general,

$$\text{si } y[n] = \sum_{k=0}^{M-1} x[n-k] \,,$$

$$\text{se tiene que } h[n] = \sum_{k=0}^{M-1} \delta[n-k] \,.$$

Ahora bien, si a la respuesta al impulso obtenida anteriormente la escalamos por el factor $1/M$, obtenemos un filtro de promedio causal.

> En resumen, un filtro de promedio (MAF: Moving Average Filter) es un sistema LTI cuya respuesta al impulso contiene M impulsos consecutivos de amplitud 1/M, que típicamente inicia en el origen y termina en M-1, donde M-1 corresponde al orden del filtro, y M es la cantidad de términos (pasados, presente y/o futuros) de la señal de entrada. El mínimo valor de M=2 (es decir, filtro de primer orden).

La respuesta al impulso de los filtros en promedio causales se define como:

$$h[n] = \frac{1}{M} \sum_{k=0}^{M-1} \delta[n-k] \qquad \text{Ecuación 14}$$

Cuya función de transferencia, es:

$$H(z) = \frac{1}{M} \sum_{k=0}^{M-1} z^{-k} \qquad \text{Ecuación 15}$$

Gráficamente, la respuesta al impulso de un filtro de promedio causal con M=11, es:

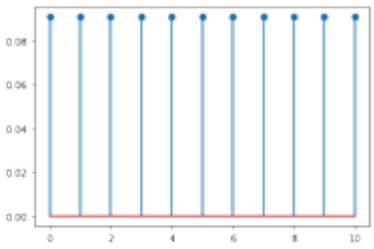


Figura 16. Respuesta al impulso de un filtro de promedio causal, M=11.

La cual se puede dibujar con el siguiente código en Python:

```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
M=11
n = np.linspace(0,M-1,M)
x = np.ones([M])/M
plt.stem(n,x, use_line_collection="True")
```

Este filtro de promedio también puede ser simétrico respecto al origen. En ese caso, típicamente se trabaja con *M* impar, cuya respuesta al impulso se define así:

$$h[n] = \frac{1}{M}\sum_{k=-(M-1)/2}^{M-1} \delta[n - k] \qquad \text{Ecuación 16}$$

Suponiendo que M=11, entonces:

$$h[n] = \frac{1}{11}\sum_{k=-5}^{5} \delta[n - k]$$

El cual corresponde a un filtro de promedio no causal (Figura 17). Entonces, para calcular la salida del sistema es necesario conocer la entrada en el tiempo actual, cinco valores pasados y cinco valores futuros del tiempo actual. Es decir, $y[n] = 1/11\ \{x[n] + x[n - 1] + x[n - 2] + x[n - 3] + x[n - 4] + x[n - 5] + x[n + 1] + x[n + 2] + x[n + 3] + x[n + 4] + x[n + 5]\}$.

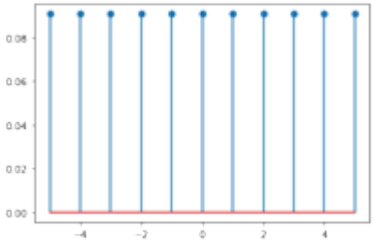


Figura 17. Respuesta al impulso de un filtro de promedio no causal, M=11.

Aunque el orden del filtro de la Figura 16 es el mismo del de la Figura 17, la principal diferencia radica en que en el primero se puede calcular la salida en tiempo real, mientras que, en el segundo es necesario que previamente se haya almacenado (o transmitido) la señal de entrada.

3.2. EFECTO DEL FILTRO DE PROMEDIO

El efecto del filtro de promedio en una señal 1D consiste en suavizarla, es decir, reducir los rizos que pueda contener la señal, manteniendo su *forma*. En otras palabras, el filtro de promedio actúa como un filtro pasa-bajos.

Para ilustrar este efecto, primero crearemos una señal senoidal a la cual le adicionaremos ruido, y posteriormente la filtraremos con filtros de promedio de diferente orden.

El código en Python paso a paso es el siguiente:

```
Paso 1:  importar librerías de trabajo

import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
from scipy import signal
import math

Paso 2:  generar una señal sin ruido

step = 0.001
t = np.arange(0,2,step)
fs = 1 / step
print(fs)
frecuencia = 2 # Hz
frad = frecuencia * 2 * math.pi
x1 = np.sin(frad*t)
plt.plot(t,x1)
plt.title('señal sin ruido')
```

La señal que se obtiene es una señal senoidal de 2 segundos de duración, con $f = 2\ Hz$, $f_s = 1\ kHz$, y amplitud en el rango [-1 1] (Ver Figura 18). Recordemos que utilizamos `plt.plot` para que tenga apariencia de señal continua, aunque realmente corresponde a una señal discreta.

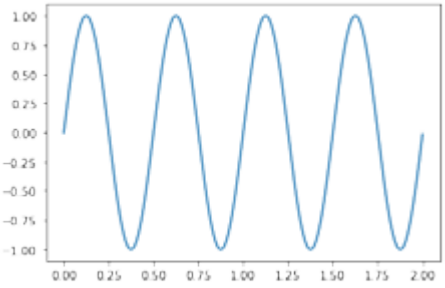


Figura 18. Señal senoidal sin ruido.

```
Paso 3:  generar ruido aleatorio

samples = len(x1)
An= 0.5
noise = An*np.random.rand(samples) - An/2
plt.plot(t,noise)
plt.title('Ruido')
```

En este paso se obtiene una señal que corresponde a ruido de 2 segundos de duración, cuya amplitud se encuentra en el rango [-0.25 0.25)]. (Ver Figura 19).

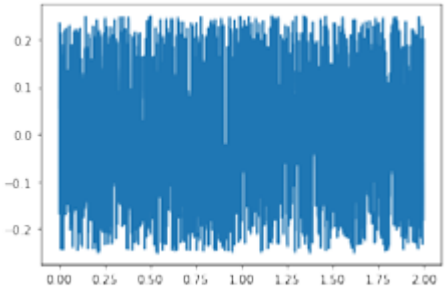


Figura 19. Ruido aleatorio.

```
Paso 4:  sumar la señal senoidal con la señal de ruido

xnoise = x1 +  noise
plt.plot(t,xnoise)
plt.title('Señal con ruido')
```

La nueva señal (Figura 20) corresponde a una señal senoidal con ruido de fondo, conservando la frecuencia fundamental de la señal de la Figura 18. No obstante, la amplitud está ahora en el rango [-1.25 1.25)].

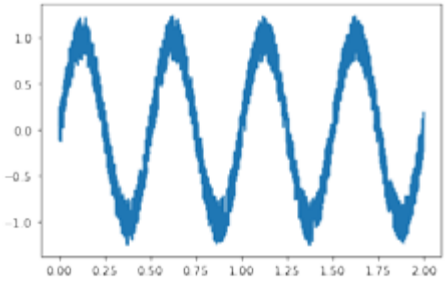


Figura 20. Señal senoidal con ruido de fondo.

```
Paso 5:  aplicación de filtro de promedio (M=7, 11, y 111)

a = 1
M = 7
b7 = np.ones([M])/M
y7 = signal.filtfilt(b7, a, xnoise)

M = 11
b11 = np.ones([M])/M
y11 = signal.filtfilt(b11, a, xnoise)

M = 111
b111 = np.ones([M])/M
y111 = signal.filtfilt(b111, a, xnoise)

# Se grafican los resultados
plt.rcParams["figure.figsize"] = (20,10)
plt.subplot(2,2,1)
plt.plot(t,xnoise)
plt.title('a)')
plt.subplot(2,2,2)
plt.plot(t,y7)
```

```
plt.title('b)')
plt.subplot(2,2,3)
plt.plot(t,y11)
plt.title('c)')
plt.subplot(2,2,4)
plt.plot(t,y111)
plt.title('d)')
plt.show()
```

Se utiliza la instrucción *filtfilt* de la librería signal para aplicar el filtro previamente diseñado (con `np.ones([M])/M`) a la señal *xnoise*. Esta instrucción permite filtrar señales 1D con filtros FIR o IRR. En el caso de filtros FIR, como corresponde al filtro de promedio, es necesario trabajar con `a = 1`.

Como resultado del código anterior se obtienen cuatro sub-gráficas, las cuales se presentan en la Figura 21.

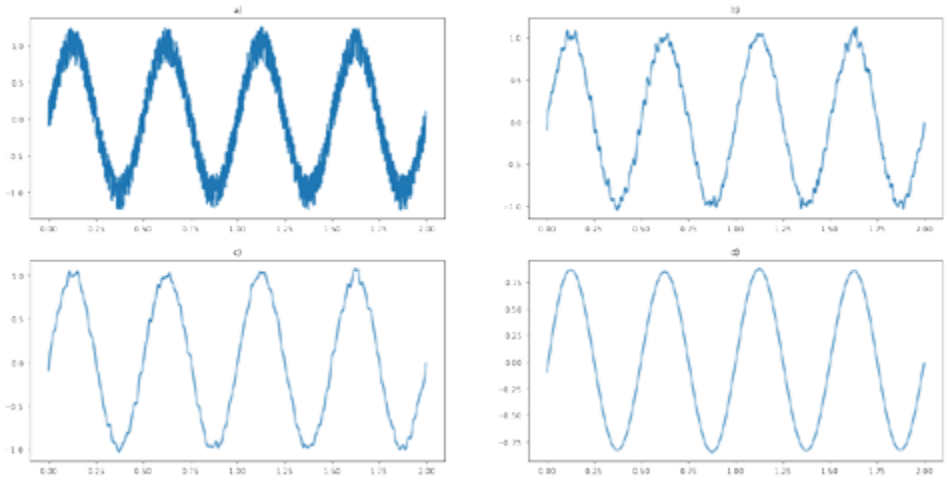


Figura 21. Resultado de filtrar una señal senoidal ruidosa con un filtro de promedio: a) señal de entrada, b) señal filtrada con M=7, c) señal filtrada con M=11, d) señal filtrada con M=111.

Al comparar los resultados obtenidos con filtros de promedio con diferentes M, se aprecia que a medida que M aumenta el efecto de suavizado es mayor, es decir, se reduce en mayor medida el rizado (ruido de fondo) de la señal. No obstante, como veremos en la siguiente sección, no se recomienda aumentar abruptamente el orden del filtro, porque se puede producir un efecto no deseado al eliminar componentes de frecuencia de la señal que son importantes. Se sugiere que el estudiante utilice un M alto (por ejemplo, M=501), y obtenga sus propias conclusiones del efecto del filtro sobre la señal.

3.3. RESPUESTA EN FRECUENCIA DEL FILTRO DE PROMEDIO

En esta sección nos centraremos en conocer y comprender el impacto que tiene el valor de M en la respuesta en frecuencia del filtro de promedio. El filtro MAF es

un filtro pasa-bajos cuya frecuencia de corte disminuye a medida que aumenta el valor de M. A diferencia de los filtros análogos, en los que la frecuencia de corte la expresamos (típicamente) en Hz, en el caso de los filtros digitales, esta frecuencia se encuentra normalizada en el rango $[0 \quad \pi)]$, por ejemplo 0.2π, con unidades $[rad/muestra]$. El rango total de la respuesta en frecuencia del filtro digital (bilateral) corresponde a $[-\pi \quad \pi)]$.

Como primer paso, vamos a reescribir la respuesta al impulso del filtro de promedio, de la forma:

$$\frac{h[n] = u[n] - u[n-M]}{M} \qquad \text{Ecuación 17}$$

donde M - 1 es el orden del filtro. Este resultado es equivalente al obtenido en la Ecuación 14.

Como segundo paso, vamos a calcular la DTFT (Transformada de Fourier de Tiempo Discreto) de $h[n]$, es decir:

$$h[n] \xrightarrow{DTFT} H(e^{j\omega}) \qquad \text{Ecuación 18}$$

Obteniendo que,

$$|H(e^{j\omega})| = \frac{1}{M}\left|\frac{sin\left(\frac{M*\omega}{2}\right)}{sin\left(\frac{\omega}{2}\right)}\right| \qquad \text{Ecuación 19}$$

Cuando graficamos la magnitud de la respuesta en frecuencia del filtro de promedio, encontramos que presenta un comportamiento especial, que lo podemos resumir como:

- Todos los filtros de promedio tienen un lóbulo principal alrededor de $\omega = 0$, y varios lóbulos secundarios que inician en $-\pi$ y terminan en π.
- La amplitud de los lóbulos secundarios disminuye a medida que se alejan de $\omega=0$. Cada lóbulo secundario es más pequeño que su antecesor (entre $[0$ $\pi]$) y existe un efecto espejo con las frecuencias negativas.
- La cantidad de lóbulos en el rango $[-\pi \quad \pi]$ es igual a M-1. Hay un lóbulo principal y M-2 lóbulos secundarios.
- Si el filtro tiene un M par, el primer "cruce por cero" y el último "cruce por cero" ocurren en las frecuencias $-\pi$ y π, respectivamente. En caso contrario, si M es impar, en esas frecuencias no existirá cruce por cero.
- En todos los casos, los cruces por cero se encuentran ubicados en $2\pi k/M$. El rango de k es $[1 \quad (M-1)/2]$ para M impar, y $[1 \quad M/2]$ para M par.

Nota: se dibuja la magnitud de la respuesta en frecuencia del filtro, por lo cual no tendrá valores negativos y formalmente no existirán los "cruces por cero". Sin embargo, sí existen valores en los cuales la amplitud ha disminuido, llega a cero, y vuelve a aumentar, los consideraremos como "cruces por cero".

Por ejemplo, para $M = 7$, el filtro de promedio tiene un lóbulo principal y cinco lóbulos secundarios, de los cuales dos lóbulos y medio (secundarios) se encuentran en las frecuencias positivas (Ver Figura 22). Dado que M es impar, el último cruce por cero en las frecuencias positivas (al igual que el primer cruce por cero en las frecuencias negativas) no ocurre en $\omega=\pi$.

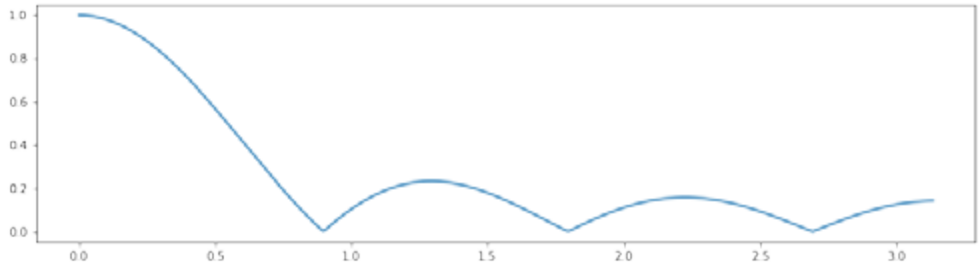


Figura 22. Magnitud de la respuesta en frecuencia de un filtro de promedio, M=7.

Para obtener la gráfica de la magnitud de la respuesta en frecuencia del filtro MAF, utilizamos el siguiente código en Python:

```
from scipy import signal
import numpy as np
from scipy import signal
import matplotlib.pyplot as plt
import math
M = 7
M7 = np.ones([M])/M
a = 1
w1, v1 = signal.freqz(M7, a)
plt.rcParams["figure.figsize"] = (14,8)
ax = plt.subplot(2, 1, 1)
plt.plot(w1, np.abs(v1))
plt.title('Respuesta en frecuencia filtro digital de promedio, M=7')
```

La instrucción `signal.freqz` de la librería de *scipy* de Python permite graficar la respuesta en frecuencia de filtros digitales, tanto FIR como IIR. Las entradas de esta instrucción corresponden a los coeficientes de los polinomios tanto del numerador como del denominador de la función de transferencia del filtro digital. En el caso del filtro MAF, por ser un filtro FIR, el denominador es una constante igual a uno, y entonces, a la entrada "`a`" de la instrucción `signal.freqz` le asignamos el valor de uno. El resultado corresponde al vector de frecuencias normalizadas, `w1`, y al vector de amplitudes, `v1`. Con la instrucción `np.abs(v1)` se calcula la magnitud de la respuesta en frecuencia del filtro digital.

Podemos obtener cada uno de los cruces por cero del filtro de promedio, con el siguiente código en lenguaje Python:

```
M=7

# k=1, entonces
k=1
wc1= 2*math.pi/M
print("frecuencia cruce por cero 1:", wc1)

# k=2, entonces
k=2
wc2= 2*math.pi*k/M
print("frecuencia cruce por cero 2:", wc2)

# k=3, entonces
k=3
wc3= 2*math.pi*k/M
print("frecuencia cruce por cero 3:", wc3)
```

y obtendríamos:

```
frecuencia cruce por cero 1: 0.8975979010256552
frecuencia cruce por cero 2: 1.7951958020513104
frecuencia cruce por cero 3: 2.6927937030769655
```

Cuando el valor de *M* es alto, se recomienda utilizar una estructura anidada (por ejemplo, ciclo for) para encontrar los cruces por cero del filtro digital, así:

```
M = 7
for k in range(1,int((M-1)/2)+1):
    wc= 2*3.14*k/M
    print("frecuencia de cruce por cero",k, ":", wc)
```

Supongamos ahora que nuestro filtro trabaja con $M = 31$, ¿cuántos lóbulos secundarios tendrá? La respuesta es 29 lóbulos secundarios, por lo que, de forma similar al caso anterior no se encontrarán cruces por cero en $-\pi$, ni en π. La gráfica se presenta en la Figura 23.

Figura 23. Magnitud de la respuesta en frecuencia de un filtro de promedio, M=31.

Independiente del orden del filtro, tendremos que en $\omega = 0$ la amplitud es igual a uno. Se aprecia que de forma similar a la gráfica de la Figura 22, el último lóbulo queda a "la mitad", es decir, no llega a cero.

Los cruces por cero los obtenemos con el siguiente código en Python:

```
M = 31
for k in range(1,int((M-1)/2)+1):
    wc= 2*3.14*k/M
    print("frecuencia de cruce por cero",k, ":", wc)
```

```
frecuencia de cruce por cero 1: 0.20258064516129032
frecuencia de cruce por cero 2: 0.40516129032258064
frecuencia de cruce por cero 3: 0.607741935483871
frecuencia de cruce por cero 4: 0.8103225806451613
frecuencia de cruce por cero 5: 1.0129032258064516
frecuencia de cruce por cero 6: 1.215483870967742
frecuencia de cruce por cero 7: 1.4180645161290324
frecuencia de cruce por cero 8: 1.6206451612903225
frecuencia de cruce por cero 9: 1.823225806451613
frecuencia de cruce por cero 10: 2.0258064516129033
frecuencia de cruce por cero 11: 2.2283870967741937
frecuencia de cruce por cero 12: 2.430967741935484
frecuencia de cruce por cero 13: 2.6335483870967744
frecuencia de cruce por cero 14: 2.836129032258065
frecuencia de cruce por cero 15: 3.0387096774193547
```

Como tercer ejemplo, utilizaremos un filtro con M par. Específicamente, si $M = 8$, obtendremos una gráfica que contiene tres lóbulos secundarios en las frecuencias positivas, y el último cruce por cero ocurre exactamente en $\omega = \pi$ (Ver Figura 24).

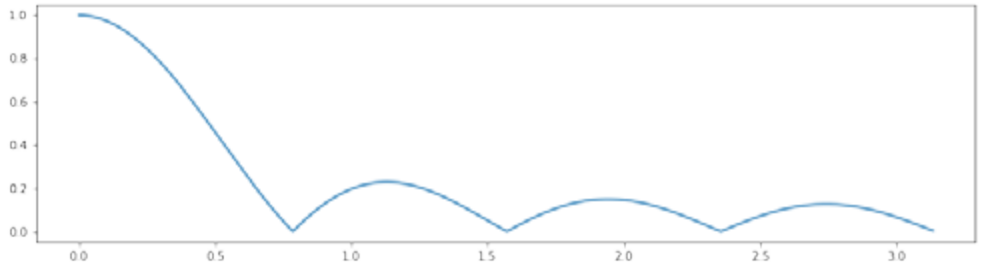


Figura 24. Magnitud de la respuesta en frecuencia de un filtro de promedio, M=8.

La respuesta en frecuencia se obtiene con el siguiente código en Python:

```
from scipy import signal
M = 8
M8 = np.ones([M])/M
a = 1
w1, v1 = signal.freqz(M8, a)
plt.rcParams["figure.figsize"] = (14,8)
ax = plt.subplot(2, 1, 1)
plt.plot(w1, np.abs(v1))
```

Y los cruces por cero, así:

```
M = 8
for k in range(1,int((M)/2)+1):
    wc= 2*3.14*k/M
    print("frecuencia de cruce por cero",k,":", wc)
```

```
frecuencia de cruce por cero 1: 0.785
frecuencia de cruce por cero 2: 1.57
frecuencia de cruce por cero 3: 2.355
frecuencia de cruce por cero 4: 3.14
```

Finalmente, si comparamos la primera frecuencia de cruce por cero de los filtros de promedio con $M = 7$, $M = 8$, y $M = 31$, podemos concluir que a medida que M aumenta, la frecuencia del primer cruce por cero disminuye (es decir, cuando utilizamos un orden de filtro alto, la frecuencia de corte es baja). No obstante, independiente del valor de M, el filtro de promedio se comporta como un filtro pasa-bajos.

3.4. FILTRO INTEGRADOR LEAKY

Este filtro tiene un comportamiento parecido al filtro de promedio (efecto pasa-bajo) cuando $M \geq 100$. La ecuación de entrada-salida se define como:

$$y[n] = \lambda y[n - 1] + (1 - \lambda)x[n] \qquad \text{Ecuación 20}$$

Cuya relación de λ y M está dada por:

$$\lambda = \frac{M - 1}{M} \qquad \text{Ecuación 21}$$

De tal forma que si $M = 100$, entonces $\lambda = 99/100 = 0.99$.

Por lo que, para este caso específico la salida es:

$$y[n] = 0.99y[n - 1] + 0.01x[n]$$

Esto significa que, para obtener la salida en el momento actual se conserva en gran parte la salida del momento anterior, y solamente una pequeñísima parte de la entrada en el momento actual. Adicionalmente, si la entrada solo existe en un momento específico (ej. $x[n] = \delta[n]$), la salida será distinta de cero a partir de ese momento en adelante.

Veamos precisamente cuál es la respuesta al impulso del filtro Leaky.

Reescribamos la ecuación 18 de la siguiente forma:

$$y[n] = \lambda y[n - 1] + (1-\lambda)\delta[n] \qquad \text{Ecuación 22}$$

Y supongamos que el sistema inicia en $n = 0$, es decir que antes de ese momento tanto la entrada como la salida eran de amplitud igual a cero.

Entonces,

- $y(0) = 0.99y(-1) + 0.01\delta[n]$, que es equivalente a $y(0) = 0.01$, dado que $y(-1) = 0$, y $x(0) = 0.01$.
- $y(1) = 0.99y(0)$, que es equivalente a $y(1) = 0.99 * 0.01$, dado que $x(1) = 0$.
- $y(2) = 0.99y(1)$, que es equivalente a $y(2) = 0.99 * 0.99 * 0.01$, dado que $x(2) = 0$.
- $y(3) = 0.99y(2)$, que es equivalente a $y(3) = 0.99 * 0.99 * 0.99 * 0.01$, dado que $x(3) = 0$.
- .... $y(k)=0.99y(k-1)$, que es equivalente a $y(k) = 0.99^k * 0.01$.

De forma general, la respuesta al impulso del Filtro Leaky se expresa como:

$$h[n] = \lambda^n (1 - \lambda) \qquad \text{para } n \geq 0 \qquad \text{Ecuación 23}$$

Examinando la ecuación 23 podemos concluir que este filtro es de respuesta al impulso infinita, dado que, a partir de $n = 0$, las amplitudes de $h[n]$ serán distintas de cero.

A continuación, dibujaremos la ecuación de entrada-salida, utilizando un diagrama de bloques:

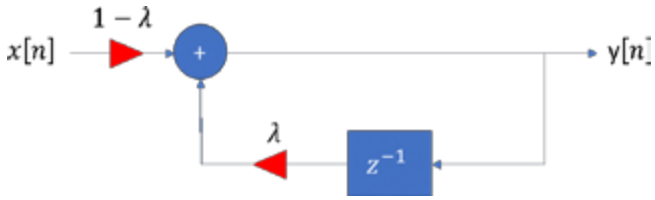


Figura 25. Diagrama de bloques filtro Leaky.

Para obtener la salida con este tipo de filtros, se necesita una unidad de retardo, un sumador y dos multiplicadores, independiente de M.

Ahora bien, dibujemos el diagrama de bloques para un filtro de promedio con $M = 100$, y comparemos el uso de recursos.

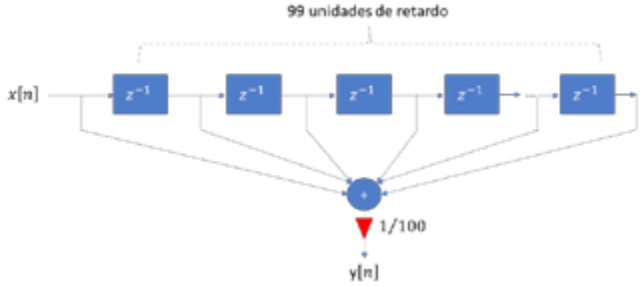


Figura 26. Diagrama de bloques filtro de promedio, M=100.

En este caso, se necesitan 99 unidades de retardo, un sumador y un multiplicador. Es claro que la cantidad de unidades de retardo es significativamente mayor que en el filtro Leaky.

Lo que significa que si se quiere implementar un filtro que *promedie* el comportamiento de la señal de entrada en las últimas 100 o 1000 muestras (por ejemplo), es más eficiente a nivel computacional utilizar una estructura como la de la Figura 25, que como de la Figura 26.

3.5. GENERALIDADES DE LOS FILTROS DIGITALES

Iniciaremos este subcapítulo de generalidades de los filtros digitales, clasificándolos en relación con su respuesta al impulso. Primero en términos de duración, y segundo en términos de cómputo. Posteriormente, revisaremos la definición de estabilidad de los filtros digitales, tomando como ejemplo el filtro de promedio y el *Leaky*.

Clasificación de los filtros digitales:

Si la respuesta al impulso del filtro es de duración finita, decimos que es FIR. En caso contrario, decimos que el filtro es IIR.

Por otro lado, un filtro puede ser causal o no causal. Un filtro es causal si su salida depende de la entrada en el mismo valor de tiempo (discreto) y/o de valores pasados de tiempo.Y es no causal, si la salida depende de valores futuros de la señal de entrada. Un filtro causal se puede ejecutar en tiempo real, es decir, que a medida que ingresa la entrada al sistema se calcula su salida. Mientras que, en filtros no causales, necesitamos conocer toda la señal de entrada para calcular la salida del sistema.

Combinando las dos clasificaciones anteriores, se pueden tener filtros FIR causales, FIR no causales, IIR causales e IIR no causales. Puedes revisar ejemplos de cada caso en el Capítulo 2.3.

Estabilidad de los filtros digitales:

Un filtro es estable si la salida del filtro es acotada para entradas acotadas. Es decir, si se cumple con la siguiente condición:

Sea $|x[n]| < M$, $|y[n]| < P$, para $M,P < \infty$. Entonces $\sum n|h[n]| < L$ para $L<\infty$.

De tal forma que,TODOS los filtros FIR son estables. Por lo que, todos los filtros de promedio son estables.

Vamos ahora a revisar la estabilidad en los filtros Leaky. Recordemos que su res-

puesta al impulso es de la forma $h[n] = \lambda^n (1 - \lambda)$ para $n \geq 0$. Entonces es necesario evaluar dos posibles escenarios, cuando $|\lambda| < 1$ y cuando $|\lambda| \geq 1$.

Escenario 1: $|\lambda| < 1$

En este caso, $\sum_{n=-\infty}^{\infty}|h[n]| = |1 - \lambda| \sum_{n=-\infty}^{\infty} |\lambda^n| = |1 - \lambda|\{\lambda^0+\lambda^1+\lambda^2+\lambda^3+\cdots+\lambda^\infty\}$ es un valor finito, dado que cada vez se suma un término más pequeño que el anterior. Por ejemplo, supongamos que $\lambda = 0.5$, entonces $|1 - \lambda| \sum_{n=-\infty}^{\infty}|\lambda^n| = |0.5|\{0.5^0+ 0.5^1 + 0.5^2 + 0.5^3+\cdots+0.5^\infty\} = \frac{0.5 + 0.25 + 0.125 + 0.0625+\cdots}{2} = \frac{1}{2}$ Entonces, el filtro IIR es estable.

Escenario 2: $|\lambda| \geq 1$

En este caso, $\sum_{n=-\infty}^{\infty}|h[n]| = |1 - \lambda| \sum_{n=-\infty}^{\infty} |\lambda^n| = |1 - \lambda|\{\lambda^0+\lambda^1+\lambda^2+\lambda^3+\cdots+\lambda^\infty\}$ es un valor infinito, dado que cada vez se suma un término más grande que el anterior. Por ejemplo, supongamos que $\lambda = 2$, entonces $|1 - \lambda| \sum_{n=-\infty}^{\infty}|\lambda^n| = |-1|\{2^0+ 2^1 + 2^2 + 2^3+\cdots+2^\infty\} =1 + 2 +4 + 8 + 16 + \cdots + \rightarrow\infty$. Entonces, el filtro IIR es inestable.

En resumen, algunos filtros IIR son estables, y otros son inestables. En el caso del filtro Leaky, es estable siempre y cuando se cumpla que $|\lambda| < 1$.

CAPÍTULO 4.
Metodos de diseño de filtros FIR

En este cuarto capítulo del libro vamos a conocer y a aplicar varios métodos o técnicas de diseño de filtros FIR. Partiremos de los filtros ideales y comprenderemos la razón por la cual no son realizables. Posteriormente, conoceremos el método de muestreo en frecuencia, y finalizaremos con el método de ventaneo. De forma simultánea abordaremos esta temática desde el punto de vista teórico, y a nivel de simulación el lenguaje de programación Python.

Al finalizar el capítulo, deberás estar en capacidad de:

1. Explicar la razón por la cual los filtros ideales no son realizables.
2. Explicar el fenómeno de Gibbs a partir del truncamiento de la respuesta al impulso de un filtro ideal.
3. Diseñar filtros (pasa-bajos, pasa-altos, pasa-banda) aplicando el método de muestreo en frecuencia, apoyándose en Python para los cálculos.
4. Diseñar filtros (pasa-bajos, pasa-altos, pasa-banda) aplicando el método de ventaneo, apoyándose en Python para los cálculos.
5. Explicar el comportamiento de los ceros en filtros FIR diseñados por los métodos de promedio y ventaneo.

4.1. FILTROS ANÁLOGOS IDEALES

Para abordar el concepto de filtros ideales, debemos primero repasar la clasificación de los filtros respecto a la respuesta en frecuencia. Los filtros se clasifican en: pasa-bajos, pasa-altos, pasa-banda y rechaza-banda.

En el caso de los filtros pasa-bajos, la banda de paso inicia en los 0 $[Hz]$ y termina en la frecuencia de corte del filtro, denominada f_c. O de forma equivalente, inicia en 0 $[rad/seg]$ y termina en W_c, para $W_c = 2\pi f_c$. A partir de la frecuencia de corte inicia la banda de rechazo, en la cual el filtro idealmente atenúa por completo esas frecuencias de la señal de entrada. Por lo tanto, en el filtro ideal la ganancia (G) en la banda de paso es constante (típicamente $G = 1$), y en la banda de rechazo es cero. En la frecuencia de corte se tiene una caída con pendiente infinita.

La respuesta en frecuencia del filtro pasa-bajo ideal se presenta en la Figura 27.

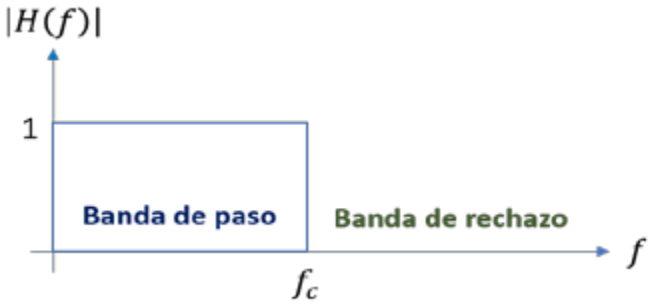


Figura 27. Respuesta en frecuencia de un filtro análogo pasa-bajo ideal.

En el caso del filtro pasa-alto ideal, la banda de rechazo inicia en 0 [Hz] y termina en la frecuencia de corte. La banda de paso corresponde a las frecuencias mayores a la f_c. Tanto el filtro pasa-alto como el filtro pasa-bajo, tienen una sola banda de paso y una sola banda de rechazo. La Figura 28 presenta la respuesta en frecuencia del filtro pasa-altos ideal.

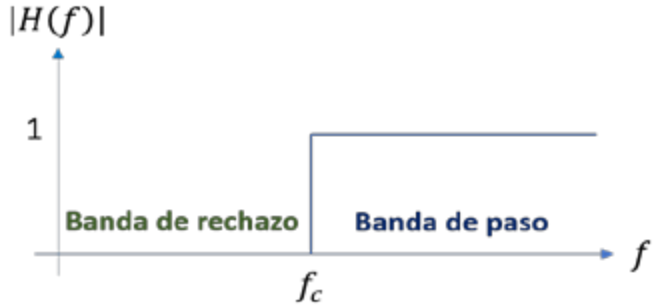


Figura 28. Respuesta en frecuencia de un filtro análogo pasa-alto ideal.

Los otros dos tipos de filtro son pasa-banda y rechaza-banda. El primero, tiene una banda de paso y dos bandas de rechazo (Figura 29). El segundo, tiene dos bandas de paso y una banda de rechazo (Figura 30). En ambos casos, se tienen dos frecuencias de corte, denominadas f_{c1} y f_{c2}.

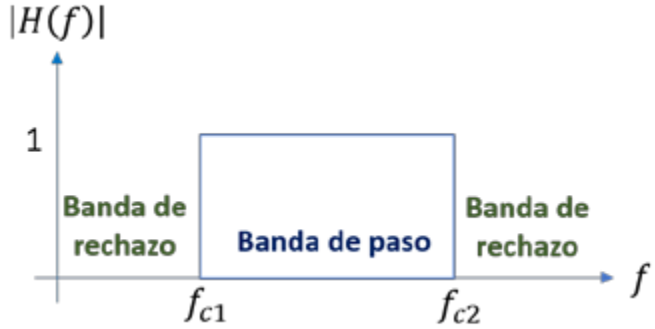


Figura 29. Respuesta en frecuencia de un filtro análogo pasa-banda ideal.

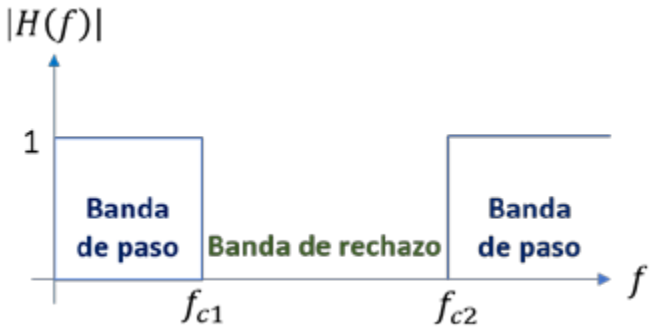


Figura 30. Respuesta en frecuencia de un filtro análogo rechaza-banda ideal.

4.2. FILTROS DIGITALES IDEALES

En el caso de los filtros digitales, la respuesta en frecuencia bilateral la expresamos en el rango [$-\pi \quad \pi$)] con unidades [rad/muestra], o en el rango [$-1 \quad 1$] con unidades [ciclo/muestra].

El filtro digital pasa-bajo ideal se presenta en la Figura 34.

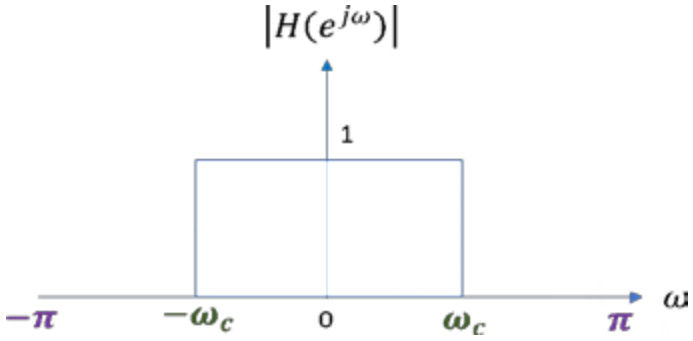


Figura 31. Respuesta en frecuencia del filtro digital pasa-bajo ideal, valores en [rad/muestra].

Matemáticamente, se define como:

$$H(e^{j\omega}) = \begin{cases} 1 & |\omega| \leq \omega c \\ 0 & e.o.c.(en\ otro\ caso) \end{cases} \quad \text{con periodicidad de } 2\pi \qquad \text{Ecuación 24}$$

Las características del filtro, son:

- Banda de paso completamente plana.
- Atenuación infinita en la banda de rechazo.
- Fase cero (sin retraso).

En el dominio del tiempo discreto, la respuesta al impulso del filtro (es decir, la Transformada de Fourier Discreta Inversa: IDTFT), es igual a:

$$h[n] = \frac{sin(\omega_c n)}{\pi n} \qquad \text{Ecuación 25}$$

La cual corresponde a una señal de duración infinita por ambos lados del eje n, conocida como señal sinc.

Revisemos ahora la estabilidad de este filtro pasa-bajos ideal. Recordando la definición de estabilidad presentada anteriormente en este libro (Capítulo 3.5), se tiene que el filtro es estable sí y solo si:

$$\sum_n |h[n]| < L \qquad para\ L<\infty$$

Entonces, el filtro pasa-bajos ideal no es estable, independiente del valor de ω_c que se seleccione, dado que la sumatoria de la magnitud de su respuesta al impulso no es finita.

A partir del concepto anterior, el *primer método* de diseño de filtros FIR corresponde al truncamiento de su respuesta al impulso. De tal forma que, partiendo de un filtro FIR ideal se selecciona un número finito de impulsos (a ambos lados del eje n) para convertirlo en un filtro estable.

4.3. TRUNCAMIENTO DE LA RESPUESTA AL IMPULSO

Este método consiste en limitar la cantidad de muestras de la respuesta al impulso del filtro. Se parte de un $h[n]$ que tiene infinitos impulsos con amplitud distinta a cero, y se llega a un $h[n]$ que tiene un número de impulsos finitos, simétrico respecto al origen.

Cuando se aplica truncamiento a $h[n]$, se hace visible el fenómeno de *Gibbs* en la respuesta en frecuencia del filtro, que consiste en la aparición de pequeñas ondulaciones tanto en la banda de paso como en la banda de rechazo del filtro. La diferencia (error) entre la máxima amplitud del rizado en relación con la amplitud plana del filtro ideal es del 9%, aproximadamente. Este error aparecerá en $H(e^{j\omega})$, independiente de la cantidad de muestras seleccionadas al truncar $h[n]$.

Por ejemplo, supongamos que la señal sinc en el dominio del tiempo discreto de duración infinita la truncamos en el rango $-5 \leq n \leq 5$, cuyo espectro se presenta en la Figura 32.

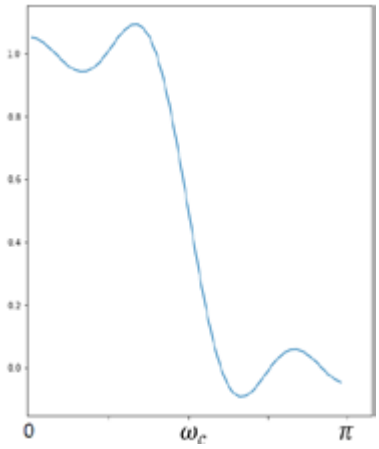


Figura 32. Espectro por truncamiento de h[n] con -5≤n≤5.

Si la misma señal *sinc* la truncamos, pero ahora en el rango $-20 \leq n \leq 20$, obtendremos el espectro de la Figura 33.

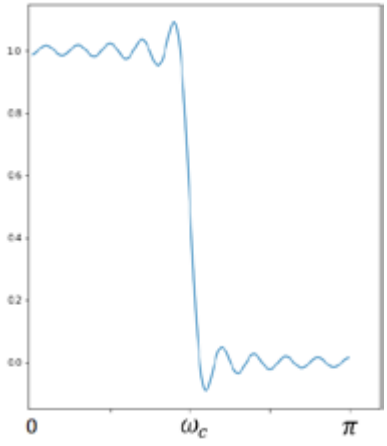


Figura 33. Espectro por truncamiento de h[n] con -20≤n≤20.

Como se aprecia en las figuras anteriores, cuando se realiza truncamiento de $h[n]$ se tiene un efecto de *"rizado"*, tanto en la banda de paso, como en la banda de rechazo. Este rizado se va "compactando" a medida que la cantidad de muestras seleccionadas de $h[n]$ aumenta, pero no desaparece.

4.4. MUESTREO EN FRECUENCIA

Este método de diseño de filtros FIR consiste en muestrear la respuesta en frecuencia de un filtro análogo ideal, y aplicar un conjunto de ecuaciones que nos permiten obtener la respuesta al impulso del filtro digital. Existen dos grupos de ecuaciones

dependiendo de si el filtro tiene una muestra en $\omega = 0$ (es decir, $\alpha = 0$) o no (es decir, $\alpha = 1)/2$). En el primer caso, se diseñan filtros con M impar, mientras que, en el segundo caso M es par.

Utilizaremos los siguientes ejemplos para ilustrar en qué consiste este método de diseño de filtros FIR. Primero, para el caso de $\alpha = 0$; y posteriormente, para $\alpha = 1)/2$.

Ejemplo 1:

Partimos de un filtro pasa-bajo ideal con $\boldsymbol{f_c = 250\ [Hz]}$. La señal de entrada la muestreamos con $\boldsymbol{f_c = 2000\ [Hz]}$ y el filtro análogo lo muestreamos con $\boldsymbol{M = 21}$ (una de sus muestras queda ubicada en la frecuencia $\boldsymbol{f = 0\ [Hz])}$. Para el diseño de este filtro, utilizaremos las ecuaciones correspondientes a $\alpha = 0$.

El valor de espaciamiento en frecuencia, Δf, entre muestras consecutivas del filtro análogo, se calcula con la siguiente ecuación:

$$\Delta f = \frac{\frac{f_s}{2}}{\frac{(M-1)}{2}} \qquad \text{Ecuación 26}$$

Que para este caso es $\boldsymbol{\Delta f = 1000/10 = 100}$, es decir que, cada 100 Hz se toma una muestra del espectro. Las muestras de amplitud distinta a cero se ubican en los siguientes valores de frecuencia $\{\boldsymbol{-200,-100,0,100,200}\}$ $[\boldsymbol{Hz}]$. Aunque la frecuencia de corte deseada está en $\boldsymbol{250\ [Hz]}$, con los valores de M y f_s seleccionados realmente se está diseñando un filtro con frecuencia de corte de $\boldsymbol{200\ [Hz]}$. El filtro muestreado se presenta en la siguiente figura.

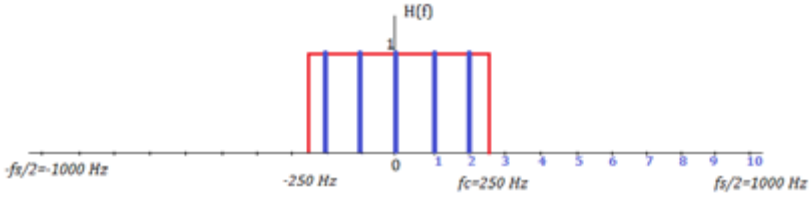


Figura 34. Muestreo en frecuencia del filtro análogo, M=21.

A partir de esta gráfica, se escribe H_r, que corresponde con el filtro muestreado:

$$H_r(k) = \begin{cases} 1 & k = 0,1,2 \\ 0 & k = 3,4,5,6,7,8,9,10 \end{cases}$$

Se debe tener en cuenta que solamente se definen los valores de k del eje de frecuencias positivo (incluido el cero), dado que los otros valores son su espejo.

A partir de H_r se obtiene $G(k)$, utilizando la siguiente ecuación:

$$G(k) = (-1)^k H_r(k) \qquad \text{Ecuación 27}$$

Realizando una alternancia en los signos de H_r así: signo positivo para los valores de k pares; signo negativo para los valores de k impar.

Entonces, para este filtro se tiene que:

$$G(k) = \begin{cases} 1 & k = 0,2 \\ -1 & k = 1 \\ 0 & k = 3,4,5,6,7,8,9,10 \end{cases}$$

Finalmente, se calcula h[n] con la ecuación (para $\alpha = 0$.):

$$h[n] = \frac{1}{M}\left\{G(0) + 2\sum_{k=1}^{U} G(k)cos\left(\left(\frac{2\pi k}{M}\right)\left(n+\frac{1}{2}\right)\right)\right\} \qquad \text{Ecuación 28}$$

La cantidad máxima de términos cosenoidales de la ecuación anterior es $U = (M-1)/2$. Sin embargo, teniendo en cuenta que a partir de k =3 se tiene que $H_r(k) = 0$, entonces solo existen los términos para $k = 1$ y $k = 2$, es decir, dos términos cosenoidales, quedando $h[n]$ expresada así:

$$h[n] = \frac{1}{21}\left\{G(0) + 2\left\{G(1)cos\left(\left(\frac{2\pi}{21}\right)\left(n+\frac{1}{2}\right)\right) + G(2)cos\left(\left(\frac{4\pi}{21}\right)\left(n+\frac{1}{2}\right)\right)\right\}\right\}$$

y al reemplazar los valores de G(k), finalmente se obtiene la siguiente ecuación de $h[n]$:

$$h[n] = \frac{1}{21}\left\{1 + 2\left\{-cos\left(\left(\frac{2\pi}{21}\right)\left(n+\frac{1}{2}\right)\right) + cos\left(\left(\frac{4\pi}{21}\right)\left(n+\frac{1}{2}\right)\right)\right\}\right\}$$

Entonces, $h[n]$ se obtiene en el rango [0 20], dado que $M = 21$.

Podemos utilizar el siguiente código en Python para obtener las 21 amplitudes de los impulsos de $h[n]$:

```
import math
import numpy as np
M=21
G0=1
G1=-1
G2=1
h= np.zeros(M)
pi = math.pi
cos = math.cos

for n in range(M):
     h[n]=1/M*(G0+2*((G1*cos(2*pi/M*(n+0.5)))+(G2*cos(4*pi/M*(n+0.5)))))
print(h)
```

Obteniendo el siguiente resultado:

```
[ 0.04445162  0.02119247 -0.01507826 -0.04761905 -0.05937998 -0.03943817
  0.01259897  0.08580656  0.16110284  0.21731539  0.23809524  0.21731539
  0.16110284  0.08580656  0.01259897 -0.03943817 -0.05937998 -0.04761905
 -0.01507826  0.02119247  0.04445162]
```

Se puede apreciar que el primer término de $h[n]$ (es decir $h[0]$) es igual al último término (es decir $h(M\text{-}1)$); el segundo término es igual al penúltimo, y así sucesivamente. De forma general, siempre que se diseñe un filtro con este método, se cumplirá que:

$$h(0) = h(M\text{-}1)$$

$$h(1) = h(M\text{-}2)$$

$$h(2) = h(M\text{-}3)$$

$$\dots$$

Como en este ejemplo M es impar, entonces el término $h((M - 1)/2)$ no tiene pareja.

Ahora, vamos a graficar la respuesta en frecuencia del filtro que hemos diseñado. Utilizaremos el siguiente código en Python:

```
from scipy import signal
import matplotlib
import matplotlib.pyplot as plt
a=1 # se hace igual a 1 porque el filtro es FIR
w1, v1 = signal.freqz(h, a)
plt.rcParams["figure.figsize"] = (14,8)
plt.plot(w1, np.abs(v1))
```

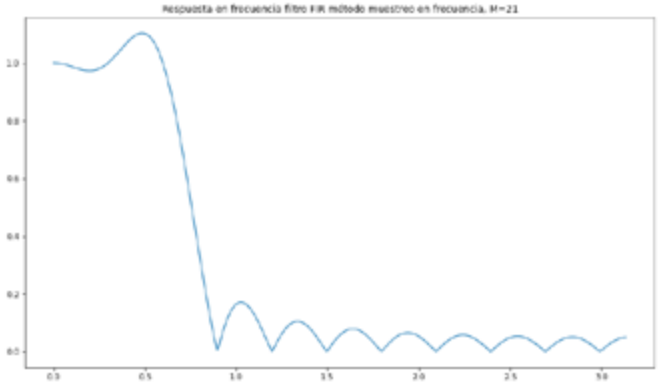


Figura 35. Magnitud de la respuesta en frecuencia método muestreo en frecuencia, M=21.

El siguiente paso consiste en encontrar a partir de la gráfica y de forma teórica la frecuencia de corte del filtro digital. Recordemos que el valor máximo es π [rad/muestra].

Para este método de diseño, la frecuencia de corte se encuentra en la amplitud en la cual se tiene una ganancia de -3 dB en escala logarítmica (o de 0.707 en escala lineal) del valor en estado estable (típicamente es 1). Entonces, de forma visual encontramos que la frecuencia de corte es de aproximadamente 0.7 $[rad/muestra]$. Podemos utilizar el siguiente código en Python para determinar su valor exacto, así:

```
x = np.where(abs(v1) > 0.707)
wcd = np.max(x)*pi/len(w1)
print(wcd)
```

```
0.6994952392758523
```

Finalmente, este valor se normaliza en el rango [0 1], de la siguiente manera:

```
fcn = wcd / pi # frecuencia de corte normalizada en el rango (0 1)
print(fcn)
```

```
0.22265625
```

Por otro lado, la frecuencia de corte normalizada teórica se calcula como:

$$f_{c_N} = \frac{2k}{M-1} \qquad \text{Ecuación 29}$$

Donde k es el máximo valor para el cual H_r es distinto de cero (o el valor mínimo para el cual H_r es distinto de cero, si el filtro es pasa-altos).

En nuestro ejemplo k=2. De tal forma que,

$$f_{c_N} = \frac{2 * 2}{20} = \frac{4}{20} = 0.2$$

El valor experimental es muy cercano el valor teórico, es decir, el filtro diseñado obtenido se aproxima en gran medida al filtro que queríamos diseñar.

Ejemplo 2:

Partimos de un filtro pasa-bajo ideal con $\boldsymbol{f_c = 450\ [Hz]}$, $\boldsymbol{f_s = 1800\ [Hz]}$, y cantidad de muestras $\boldsymbol{M = 18}$. Sin embargo, como M es par, se tiene que $\alpha = 1/2$, lo que significa que no existe muestra en $\boldsymbol{f = 0\ [Hz]}$, sino en $\boldsymbol{f = \Delta f/2\ [Hz]}$.

El valor de espaciamiento en frecuencia, Δf, entre muestras consecutivas del filtro análogo, se calcula con la siguiente ecuación:

$$\Delta f = \frac{\frac{f_s}{2}}{\frac{(M)}{2}} \qquad \text{Ecuación 30}$$

Obteniendo $\boldsymbol{\Delta f = 900/9 = 100\ [Hz]}$, cuyas muestras de valor distinto a cero se ubican en $\boldsymbol{\{-450,-350,-250,-150,-50,50,150,250,350,450\}\ [Hz]}$. El filtro muestreado se presenta en la siguiente figura.

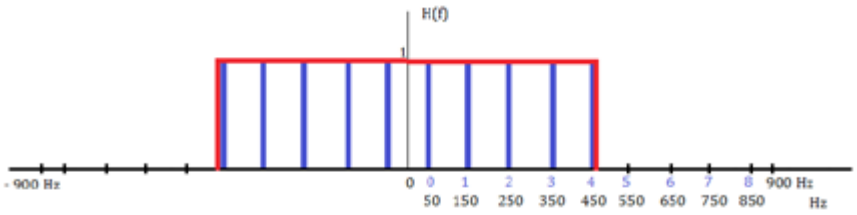


Figura 36. Muestreo en frecuencia del filtro análogo, M=18.

Como siguiente paso, escribiremos el valor de H_r, así:

$$H_r(k) = \begin{cases} 1 & k = 0,1,2,3,4 \\ 0 & k = 5,6,7,8, \end{cases}$$

Y obtenemos $G(k)$ utilizando la ecuación 27,

$$G(k) = \begin{cases} 1 & k = 0,2,4 \\ -1 & k = 1,3 \\ 0 & k = 5,6,7,8,9,10 \end{cases}$$

Y calculamos $h[n]$, a partir de $G(k)$. Se enfatiza que la ecuación cuando M es par se expresa en términos de senoidales, y no de cosenoidales como en el ejemplo anterior.

La ecuación general es:

$$h[n] = \frac{2}{M}\left\{\sum_{k=0}^{U} G(k) sen\left(\left(\frac{2\pi}{M}\right)\left(k+\frac{1}{2}\right)\left(n+\frac{1}{2}\right)\right)\right\} \qquad \textbf{Ecuación 31}$$

La cantidad máxima de términos senoidales de la ecuación anterior es $U = \frac{M}{2} - 1$, sin embargo, teniendo en cuenta que a partir de k=5 se tiene que $h_r(k)$ = 0, solamente se tendrán en este ejemplo cinco términos correspondientes a k = 0,1,2,3 y 4.

Entonces, la respuesta al impulso del filtro se define, así:

$$h[n] = \frac{2}{18}\left\{sen\left(\left(\frac{2\pi}{18}\right)\left(\frac{1}{2}\right)\left(n+\frac{1}{2}\right)\right) - sen\left(\left(\frac{2\pi}{18}\right)\left(\frac{3}{2}\right)\left(n+\frac{1}{2}\right)\right) + sen\left(\left(\frac{2\pi}{18}\right)\left(\frac{5}{2}\right)\left(n+\frac{1}{2}\right)\right) - sen\left(\left(\frac{2\pi}{18}\right)\left(\frac{7}{2}\right)\left(n+\frac{1}{2}\right)\right) + sen\left(\left(\frac{2\pi}{18}\right)\left(\frac{9}{2}\right)\left(n+\frac{1}{2}\right)\right)\right\}$$

Y se pueden obtener sus valores con el siguiente código en Python:

```
import math
import numpy as np

M=18
G0=1
G1=-1
G2=1
G3=-1
G4=1

h= np.zeros(M)
pi = math.pi
sin = math.sin

for n in range(M):
      h[n]=2/M*((G0*sin(2*pi/M*(0.5)*(n+0.5)))+(G1*sin(2*pi/M*(1+0.5)*(n+0.5)))+
               (G2*sin(2*pi/M*(2+0.5)*(n+0.5)))+(G3*sin(2*pi/M*(3+0.5)*(n+0.5)))
+
            (G4*sin(2*pi/M*(4+0.5)*(n+0.5))));
print(h)
```

Obteniendo como resultado,

```
[ 0.04272059  0.02875767 -0.057602   -0.01177696  0.07856742 -0.01681924
 -0.1235279   0.10732509  0.48829857  0.48829857  0.10732509 -0.1235279
 -0.01681924  0.07856742 -0.01177696 -0.057602    0.02875767  0.04272059]
```

De forma similar a lo obtenido en el ejemplo 1, el primer valor de $h[n]$ es igual al último valor, el segundo valor es igual al penúltimo valor, y así sucesivamente. A diferencia del caso anterior, no existe un valor que quede sin pareja, dado que M es par.

Continuaremos, dibujando la respuesta en frecuencia del filtro, con el siguiente código en Python:

```
from scipy import signal
import matplotlib
import matplotlib.pyplot as plt
a=1
w1, v1 = signal.freqz(h, a)
plt.rcParams["figure.figsize"] = (14,8)
plt.plot(w1, np.abs(v1))
```

Obteniendo,

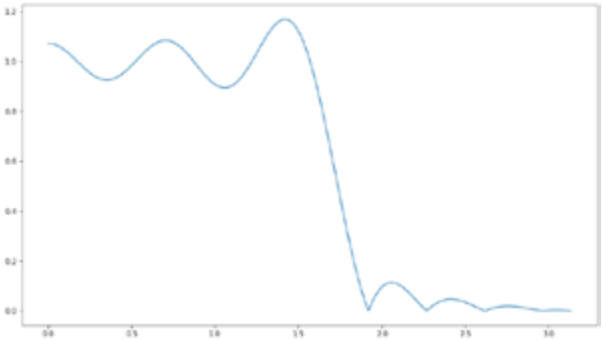


Figura 37. Magnitud de la respuesta en frecuencia método muestreo en frecuencia, M=18.

A partir de la figura anterior, se puede determinar que la frecuencia de corte del filtro digital se encuentra alrededor de 1.7 [$rad/muestra$] (evaluando la frecuencia cuya amplitud es 0.707).

Nos podemos apoyar en Python para encontrar su valor, con el siguiente código:

```
x = np.where(abs(v1) > 0.707)
wcd = np.max(x)*pi/len(w1)
print(wcd)
```

```
1.6689710972195777
```

Ahora, calculamos la frecuencia de corte normalizada del filtro digital, así:

```
fcn = wcd / pi # frecuencia de corte normalizada en el rango (0 1)
print(fcn)
```

```
0.53125
```

Y el valor teórico, por medio de la ecuación:

$$f_{c_N} = \frac{2}{M}\frac{(2k+1)}{2} \qquad \text{Ecuación 32}$$

Obteniendo en este caso,

$$f_{c_N} = \frac{2}{18} * \frac{9}{2} = 0.11 * 4.5 = 0.5$$

Como conclusión, hemos verificado que el filtro quedó diseñado correctamente.

4.5. VENTANEO

Podemos decir que este método se inspiró en el concepto de truncamiento de la respuesta al impulso. Lo que se busca, es limitar la cantidad de impulsos de la señal sinc, para que el filtro sea realizable (es decir, que no requiera de una señal en tiempo discreto de duración infinita por ambos lados del eje n), y, adicionalmente, sea estable. Sin embargo, en este caso no se descartan los coeficientes que estén por fuera del rango de la señal sinc seleccionado, sino que, se multiplica en el dominio del tiempo discreto la señal sinc por una ventana de duración finita. El efecto en el dominio de la frecuencia es el de la convolución entre el espectro de la señal sinc (que corresponde al filtro ideal) y el espectro de la ventana.

Matemáticamente, el concepto anterior lo expresamos así:

Sea $h[n]$ la respuesta al impulso del filtro ideal, y $w[n]$ la ventana discreta de duración finita. Cada una de estas señales tiene su correspondiente espectro, así:

$$h[n] \xrightarrow{DTFT} H(\omega) \qquad \text{Ecuación 33}$$

$$w[n] \xrightarrow{DTFT} W(\omega) \qquad \text{Ecuación 34}$$

Donde DTFT corresponde a la Transformada de Fourier de Tiempo Discreto (*Discrete-Time Fourier Transform*).

Entonces, se multiplica en el dominio del tiempo discreto la señal $h[n]$ de duración infinita con la señal $w[n]$ de duración finita, obteniendo una respuesta al impulso de duración finita, la cual denominaremos $\hat{h}(n)$.

$$\hat{h}[n] = h[n].w[n] \qquad \text{Ecuación 35}$$

El espectro de $\hat{h}[n]$ lo denominaremos $\hat{h}(\omega)$, el cual se obtiene de convolucionar los espectros de las señales h[n] y w[n], es decir,

$$\hat{h}[n] \xrightarrow{DTFT} \hat{H}(\omega) \qquad \text{Ecuación 36}$$

$$\hat{H}(\omega) = \frac{1}{2\pi}\{H(\omega) \circledast W(\omega)\} \qquad \text{Ecuación 37}$$

Donde $\circledast$ es el operador de convolución.

A continuación, se presenta de forma gráfica el proceso de ventaneo, en el dominio del tiempo y de la frecuencia.

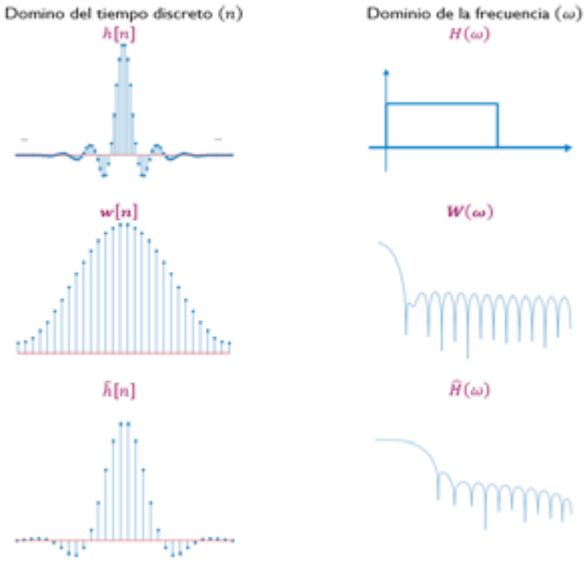


Figura 38. Diseño de filtros FIR utilizando el método de ventaneo.

Algo importante a resaltar, es que existen varios tipos de ventanas. Algunas son más suaves, otras tienen cambios bruscos de amplitud, unas son más puntiagudas, otras más anchas. Cada tipo de ventana tiene su correspondiente espectro, por lo que, el filtro resultante tendrá características diferentes. Por ejemplo, existen ventanas que atenúan de forma significativa en frecuencias distantes a la frecuencia de corte, pero que no atenúan muy bien en frecuencias cercanas a la frecuencia de corte. Otras ventanas tienen un comportamiento casi homogéneo en la zona de rechazo, pero con niveles de atenuación menores que las primeras.

En Python, la librería *scipy* tiene 23 tipos de ventanas[2] . Para diseñarlas, se puede utilizar la instrucción *signal.get_window*, o directamente con el nombre de la ventana.

A continuación, se presenta el código en Python para crear varios tipos de ventanas.

a) Ventana Boxcar (rectangular)

```
import matplotlib.pyplot as plt
from scipy import signal
M=50 # orden del filtro = M-1.
window1 = signal.boxcar(M)
from pylab import rcParams
rcParams['figure.figsize'] = 10, 6
plt.stem(window1)
```

2 https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.get_window.html#scipy.signal.get_window

b.) Ventana Hamming

```
window2 = signal.hamming(M)
from pylab import rcParams
rcParams['figure.figsize'] = 10, 6
plt.stem(window2)
```

c) Ventana Blackman

```
window3 = signal.blackman(M)
from pylab import rcParams
rcParams['figure.figsize'] = 10, 6
plt.stem(window3)
```

d) Ventana Hanning

```
window4 = signal.hann(M)
from pylab import rcParams
rcParams['figure.figsize'] = 10, 6
plt.stem(window4)
```

e) Ventana Triangular

```
window5 = signal.triang(M)
from pylab import rcParams
rcParams['figure.figsize'] = 10, 6
plt.stem(window5)
```

d) Ventana Tukey

```
window6 = signal.windows.tukey(M)
from pylab import rcParams
rcParams['figure.figsize'] = 10, 6
plt.stem(window6)
```

En la Figura 39 se presentan las seis ventanas diseñadas, todas con el mismo orden del filtro, M=50.

La primera ventana, correspondiente a boxcar, es una ventana cuyas muestras son constantes e iguales a uno, de tal forma que, es equivalente a truncar la señal *sinc* cuando se multiplica por esta ventana en el dominio del tiempo discreto. La quinta ventana, *triang*, debe su nombre precisamente a la figura geométrica que generan sus amplitudes. La ventana *tukey* se caracteriza porque tiene una zona creciente seguida de una zona constante y posteriormente una zona decreciente. Las otras tres ventanas que se seleccionaron en este ejemplo son muy similares entre sí, con un cambio de amplitud suave (sin saltos abruptos). Tanto la ventana *blackmann* como la hanning tienen su primera y última muestra de amplitud igual a cero, a diferencia de la ventana hamming que inicia y termina con una amplitud mayor a cero. Adicionalmente, de estas tres ventanas la más "angosta" es la ventana *blackman*.

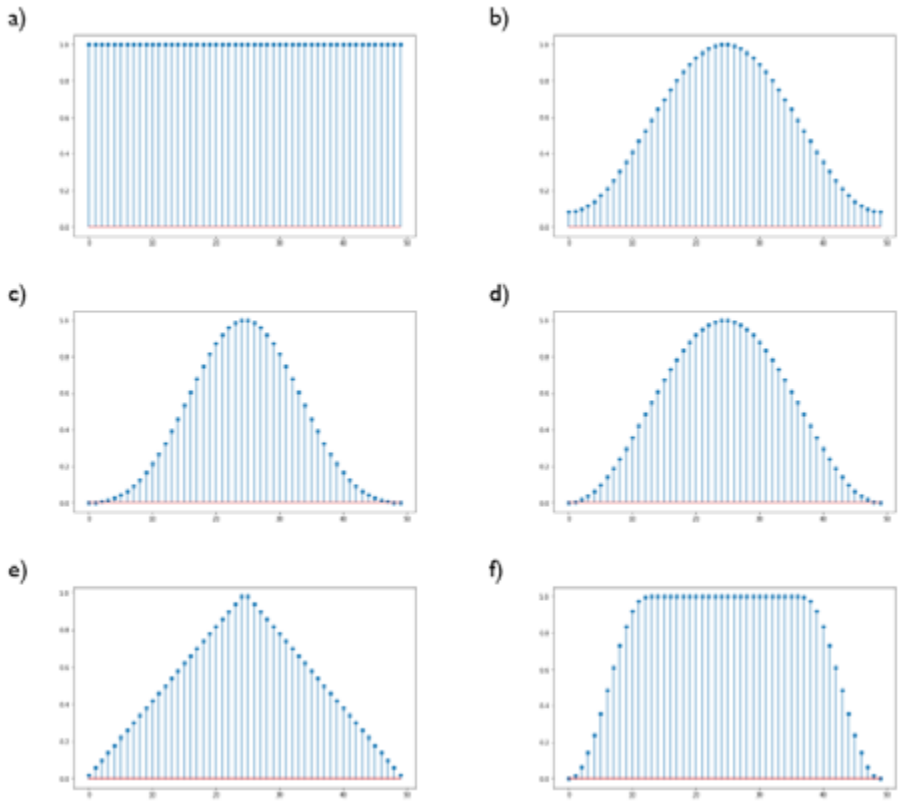


Figura 39. Ejemplos de ventanas, M=50: a) boxcar, b) hamming, c) blackman, d) hanning, e) triangular, f) tukey.

Ahora, compararemos la respuesta en frecuencia de las seis ventanas. Para ello, utilizaremos el siguiente código en Python:

```
from scipy.fft import fft, fftshift
import numpy as np
plt.figure()
window = window1 # se reemplaza para cada una de las ventanas dise-
ñadas previamente
A1 = fft(window, 2048) / (len(window)/2.0)
freq1 = np.linspace(-0.5, 0.5, len(A1))
freq1 = freq1 * 2
response1 = np.abs(fftshift(A1 / abs(A1).max()))
response1 = 20 * np.log10(np.maximum(response1, 1e-10))
from pylab import rcParams
rcParams['figure.figsize'] = 10, 6
N = len(freq1)//2
plt.plot(freq1[N+1:2*N], response1[N+1:2*N])
```

Obteniendo los siguientes espectros:

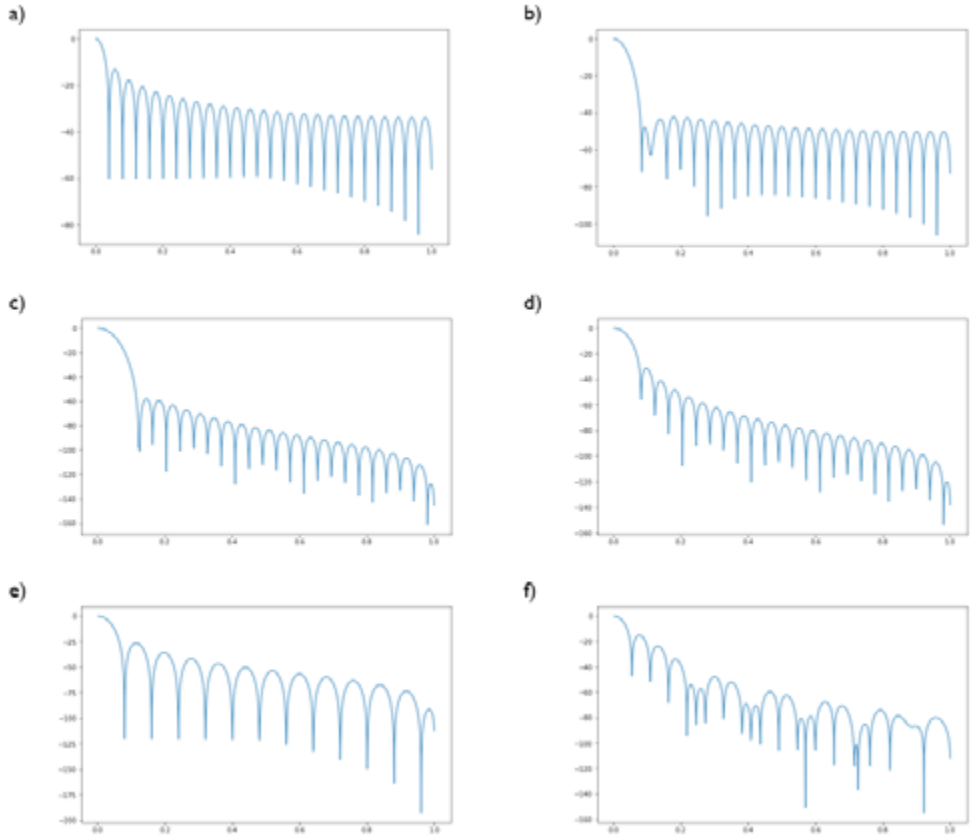


Figura 40. Respuesta en frecuencia para M=50 de las ventanas: a) boxcar, b) hamming, c) blackman, d) hanning, e) triangular, f) tukey.

Como era de esperarse, los espectros obtenidos de las seis ventanas diseñadas difieren entre sí. Empezaremos comentando el espectro de *boxcar*, el cual presenta la menor atenuación en la banda de rechazo, oscilando su ganancia entre -30 dB a -60dB, mientras que otras ventanas como la *blackman* llegan a tener hasta -160 dB de ganancia. En el caso de la ventana *hamming* la ganancia oscila entre -50dB y -90dB.

Como paso final, diseñaremos el filtro FIR con el método de ventaneo. Para ello, debemos seleccionar la ventana por la cual multiplicaremos en el dominio del tiempo la señal *sinc;* mientras que, en el dominio de la frecuencia se realizará la convolución de los dos espectros. En Python utilizamos la instrucción `signal.firwin` de la librería *scipy* para el diseño del filtro FIR por el método de ventaneo.

Los filtros que diseñaremos a continuación son pasa-bajos. Utilizaremos $f_s = 8000\ [Hz]$, y entonces por Nyquist la máxima frecuencia de corte es $fc_{max} = f_s/2 = 4000[Hz]$. Seleccionaremos como frecuencia de corte $fc = 2000[Hz]$, obteniendo que $fc = fc_{max}/2$.

Importe de librerías:

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

Parámetros de diseño (M, frecuencia y tipo de filtro):

```
M=50 # el filtro es de orden M-1
f=2000
pass_zero=True # True corresponde a un filtro pasabajo.
```

Diseño del filtro FIR con la ventana *boxcar* y visualización de la respuesta en frecuencia:

```
h1= signal.firwin(M, f, window='boxcar', fs=8000, pass_zero=pass_
zero)
w1, v1 = signal.freqz(h1,1)
```

Diseño del filtro FIR con la ventana *hamming* y visualización de la respuesta en frecuencia:

```
h2= signal.firwin(M, f, window='hamming', fs=8000, pass_zero=pass_
zero)
w2, v2 = signal.freqz(h2, 1)
```

Diseño del filtro FIR con la ventana *blackman* y visualización de la respuesta en frecuencia:

```
h3= signal.firwin(M, f, window='blackman', fs=8000, pass_zero=pass_
zero)
w3, v3 = signal.freqz(h3, 1)
```

Diseño del filtro FIR con la ventana *hanning* y visualización de la respuesta en frecuencia:

```
h4= signal.firwin(M, f, window='hann', fs=8000, pass_zero=pass_zero)
w4, v4 = signal.freqz(h4, 1)
```

Diseño del filtro FIR con la ventana triangular y visualización de la respuesta en frecuencia:

```
h5= signal.firwin(M, f, window='triang', fs=8000, pass_zero=pass_
zero)
w5, v5 = signal.freqz(h5, 1)
```

Diseño del filtro FIR con la ventana tukey y visualización de la respuesta en frecuencia:

```
h6= signal.firwin(M, f, 200, window='tukey', fs=8000)
w6, v6 = signal.freqz(h6, 1)
```

Primero dibujaremos la respuesta al impulso de los filtros, $\hat{h}(n)$, (resultado de multiplicar en el dominio del tiempo la señal sinc por la ventana), y posteriormente, la respuesta en frecuencia del filtro diseñado, $\hat{H}(\omega)$, (resultado de convolucionar en el dominio de la frecuencia la respuesta en frecuencia de la ventana con la del filtro ideal).

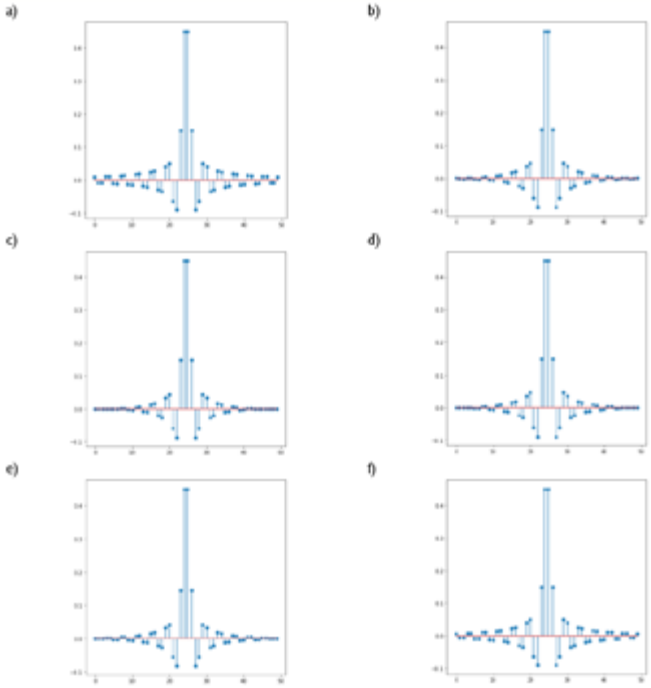


Figura 41. Respuesta al impulso, $\hat{h}(n)$, método de ventaneo, M=50: a) boxcar, b) hamming, c) blackman, d) hanning, e) triangular, f) tukey.

Se utiliza el siguiente código para dibujar las seis respuestas al impulso de los filtros:

```
plt.stem(h) # h= h1, h2, … h6.
```

Al comparar las gráficas, se aprecia que las diferencias se ven más marcadas en los primeros y últimos impulsos de $\hat{h}(n)$, es decir, en las amplitudes más pequeñas de la señal sinc.

Posteriormente, dibujaremos la respuesta en frecuencia de los seis filtros FIR, utilizando escala logarítmica:

```
plt.plot(w, 20*np.log10(np.abs(v))) # w= w1, w2, … w6.  v= v1, v2, … v6.
```

Y obtenemos los siguientes espectros:

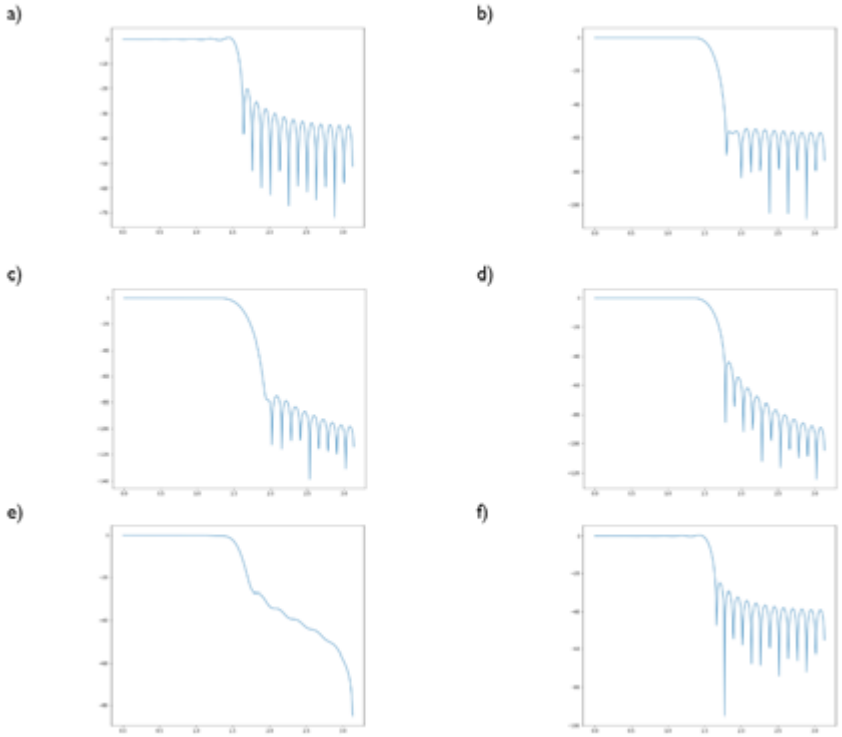


Figura 42. Respuesta en frecuencia de filtros FIR diseñados con ventanas (escala logarítmica), M=50: a) boxcar, b) hamming, c) blackman, d) hanning, e) triangular, f) tukey.

Como se observa, el comportamiento de la respuesta en frecuencia del filtro en la banda de paso cambia de forma significativa entre las ventanas seleccionadas para su diseño. Las mayores atenuaciones (ganancias alrededor de -100dB) se obtienen con las ventanas hamming, blackman, y hanning.

Finalmente, visualizaremos la respuesta en frecuencia de los filtros, pero ahora en escala lineal. El objetivo es poder determinar de forma gráfica la frecuencia de corte del filtro digital obtenido.

Para ello, utilizaremos la siguiente instrucción para cada filtro:

```
plt.plot(w, (np.abs(v))) # w= w1, w2, … w6.  v= v1, v2, … v6.
```

Y obtenemos las siguientes gráficas:

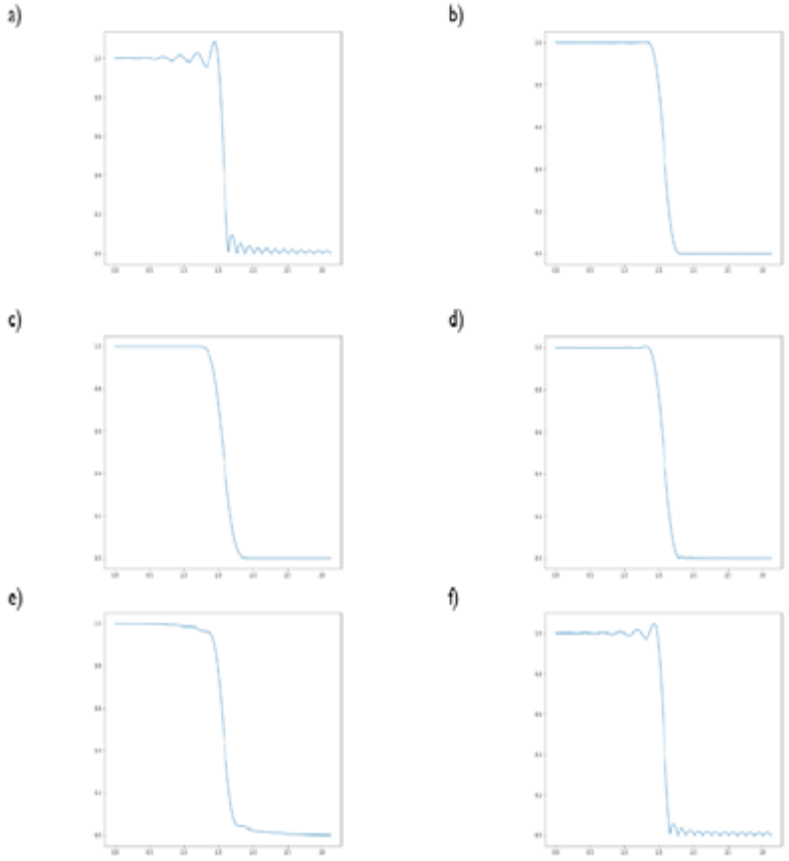


Figura 43. Respuesta en frecuencia de filtros FIR diseñados con ventanas (escala lineal), M=50: a) boxcar, b) hamming, c) blackman, d) hanning, e) triangular, f) tukey.

Podemos observar que tanto en la banda de paso como en la de rechazo, el filtro que presenta mayores ondulaciones es *boxcar* (debido al fenómeno de *Gibbs* que vimos previamente). En segundo lugar, se encuentra el filtro diseñado con la ventana tukey. Los filtros con "mejor" comportamiento, de los evaluados en esta sección, son *hamming, blackman y hanning.* En el caso del filtro diseñado con la ventana triang, su respuesta no es tan *"plana"* en las bandas de paso y de rechazo.

Como paso final, calcularemos la frecuencia de corte del filtro digital y la compararemos con la frecuencia de corte teórica. Como se mencionó en el Capítulo 2, se debe encontrar entre $[0 \quad \pi]$ (en unidades rad/muestra). Como este ejemplo utilizó $fc = fc_{max}/2$, entonces la frecuencia de corte teórica del filtro digital es de $\pi/2$.

Con el siguiente código en Python encontramos la frecuencia de corte experimental (ω_{cd}) de los seis filtros FIR diseñados con las ventanas *boxcar, hamming, blackman, hanning, triangular, y tukey.*

```
pi = math.pi
x = np.where(abs(v) > 0.707) # Para v= v1, v2, … v6.
wcd = np.max(x)*pi/len(w) # Para w= w1, w2, … w6.
print(wcd)
```

Y se obtienen los siguientes resultados:

Ventana	*boxcar*	*hamming*	*blackman*	*hanning*	*triangular*	*tukey*
ω_{cd}	1.5401	1.5155	1.5033	1.5094	1.5155	1.5401

Los cuales son cercanos al valor teórico, correspondiente a 1.5707 [*rad/muestra*].

4.6. CEROS EN FILTROS FIR

Partiendo de la función de transferencia del filtro digital, H(z), que estudiamos en el Capítulo 2 de este libro, tenemos que un filtro FIR se expresa de la siguiente forma:

$$H(z) = \sum_{k=0}^{M-1} a_k z^{-k}$$

Donde M -1 es el orden del filtro. Entonces, la cantidad de términos de $H(z)$ diferentes de cero es M. Esta función de transferencia también se puede escribir como una multiplicatoria (en lugar de una sumatoria) de M -1 términos, a partir de la factorización del polinomio de z, así:

$$H(z) = \prod_{k=1}^{M-1}(z - c_k) \qquad \text{Ecuación 38}$$

Por ejemplo, vamos a suponer que la función de transferencia del filtro FIR es $H(z) = 1 - 2z^{-1} + z^{-2}$, entonces factorizamos el polinomio de segundo orden obteniendo dos términos, así, $H(z) = (1 - z^{-1})(1 - z^{-1})$. Cada uno de los términos representa las raíces del numerador de la función de transferencia, y se conocen como los ceros del filtro digital. Es decir, cada término se iguala a cero y se despeja z para obtener los ceros del filtro.

Para este ejercicio, se tienen dos ceros en la misma posición, ubicados en:

$$(1 - z^{-1}) = 0 \quad \therefore \quad 1 = z^{-1} \quad \therefore \quad z = 1$$

Es decir, c_1=1, c_2 = 1.

> Se resalta que en el caso de los filtros FIR, solamente se tienen raíces en el numerador, es decir, los filtros FIR son sistemas solo-ceros.

En el capítulo 5 se generalizará este concepto a filtros IIR.

A medida que avancemos en el libro conoceremos el *"significado"* de los ceros de un filtro digital. Por ahora, graficaremos su posición en el plano z, apoyándonos en lenguaje de programación Python.

Partiremos con el filtro de promedio que estudiamos en el Capítulo 2 y seguiremos con el método de ventaneo.

Gráfica polos y ceros filtro de promedio:

Lo primero que vamos a realizar es definir el vector de amplitudes del filtro de promedio utilizando `np.ones`. Posteriormente, calculamos los ceros (z), polos (p) y ganancia (k), de la función de transferencia del filtro, por medio de `signal.tf2zpk`. Finalmente, dibujamos el círculo unitario en el plano z con `plt.plot(np.cos(theta),np.sin(theta))`, y ubicamos los ceros con `plt.scatter(np.real(z1),np.imag(z1))`. Se resalta que el filtro de promedio no tiene polos, dado que es un filtro FIR. Este concepto se explica con mayor detalle en el próximo capítulo.

El código completo en Python se presenta a continuación:

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

M = 2 # M = 2, 3, 4, 5.
b = np.ones(M)/(M)
z, p, k = signal.tf2zpk(b,1)
print(len(z))
theta = np.linspace(-math.pi,math.pi,201)
plt.rcParams["figure.figsize"] = (7,7)
plt.plot(np.cos(theta),np.sin(theta))
plt.scatter(np.real(z),np.imag(z))
plt.show()
```

Y se obtienen las siguientes gráficas:

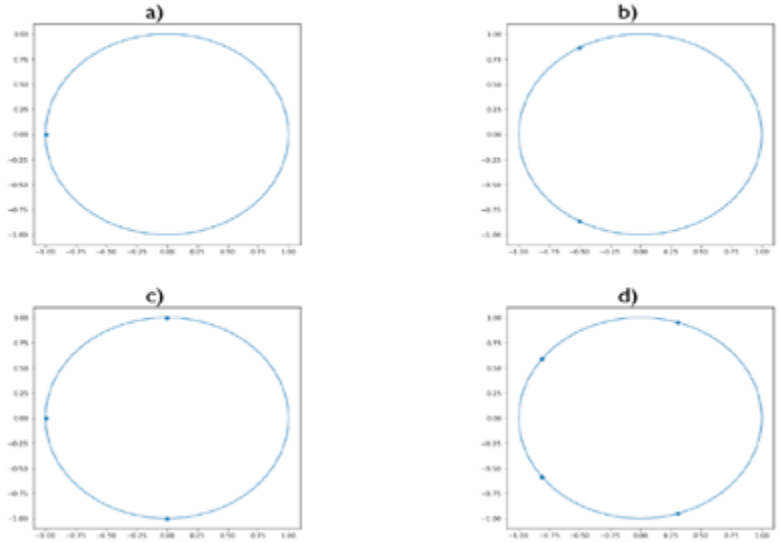


Figura 44. Gráfica de polos y ceros filtro de promedio, para: a) M=2, b) M=3, c) M=4, d) M=5.

De la figura anterior, se pueden enumerar las siguientes conclusiones:

- Todos los ceros de un filtro de promedio se ubican sobre el círculo unitario.
- La cantidad de ceros es igual a M - 1. Es decir, se tienen tantos ceros como el orden del filtro.
- Cuando el valor de M - 1. es par, cada cero tiene su conjugado, es decir, comparten el mismo valor de la parte real y con signo contrario en la parte imaginaria.
- Cuando el valor de M - 1. es impar, se tiene un cero en z = -1.
- Los ceros se concentran en la parte izquierda de la gráfica, como "alejándose" de z = -1

Gráfica polos y ceros filtro diseñado por el método de ventaneo:

Para este método de diseño, la gráfica de polos y ceros es distinta a la obtenida con el filtro de promedio. Se sugiere utilizar M par en filtros pasa-bajos, y M impar en filtros pasa-altos.

Para filtro pasa-bajos y ventana *hamming*, utilizamos el siguiente código en Python:

```
from scipy import signal
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import math
M=2 # hacer M = 2, 4, 6, 8
f1=2000
pass_zero=True # Si es True corresponde a un filtro pasa-bajo.
h1= signal.firwin(M, f1, window='hamming', fs=8000, pass_zero=pass_zero)
z1, p1, k1 = signal.tf2zpk(h1,1)
theta = np.linspace(-math.pi,math.pi,201)
plt.rcParams["figure.figsize"] = (7,7)
plt.plot(np.cos(theta),np.sin(theta))
plt.scatter(np.real(z1),np.imag(z1))
plt.show()
```

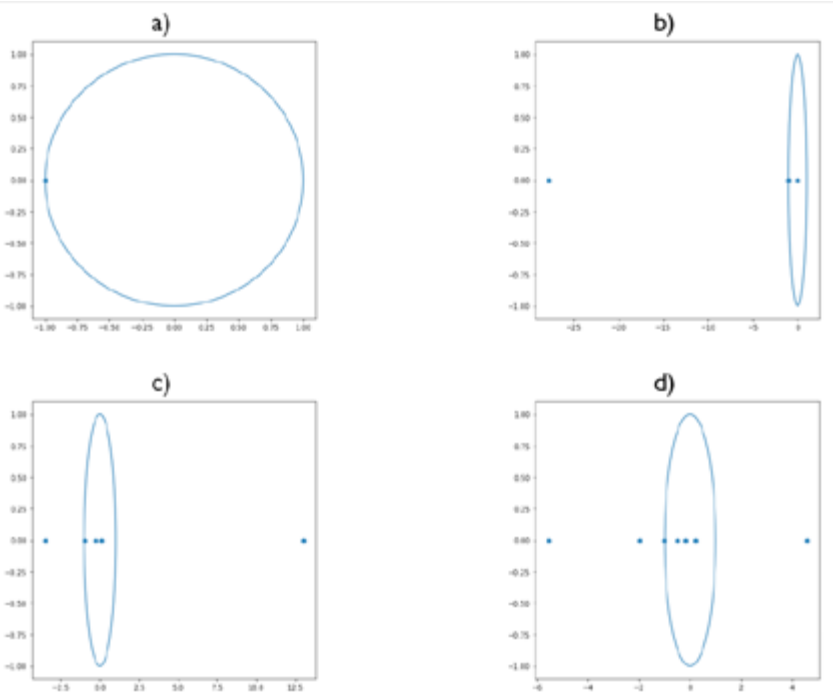


Figura 45. Gráfica de polos y ceros, filtro pasa-bajos diseñado con la ventana hamming: a) M=2, b) M=4, c) M=6, d) M=8.

En este caso, se obtienen ceros por fuera del círculo unitario, principalmente en valores negativos de z, pero eventualmente también en valores positivos. No obstante, se aprecia que uno de los ceros se encuentra en $z = -1$, dado que se diseñó un filtro pasa-bajos.

A continuación, vamos a graficar los polos y ceros, pero ahora de un filtro pasa-altos.

```
M=4
f1=2000
pass_zero=False # Si es False a un filtro pasa alto
h1= signal.firwin(M, f1, window='hamming', fs=8000, pass_zero=pass_zero)
z1, p1, k1 = signal.tf2zpk(h1,1)
theta = np.linspace(-math.pi,math.pi,201)
plt.rcParams["figure.figsize"] = (7,7)
plt.plot(np.cos(theta),np.sin(theta))
plt.scatter(np.real(z1),np.imag(z1))
plt.show()
```

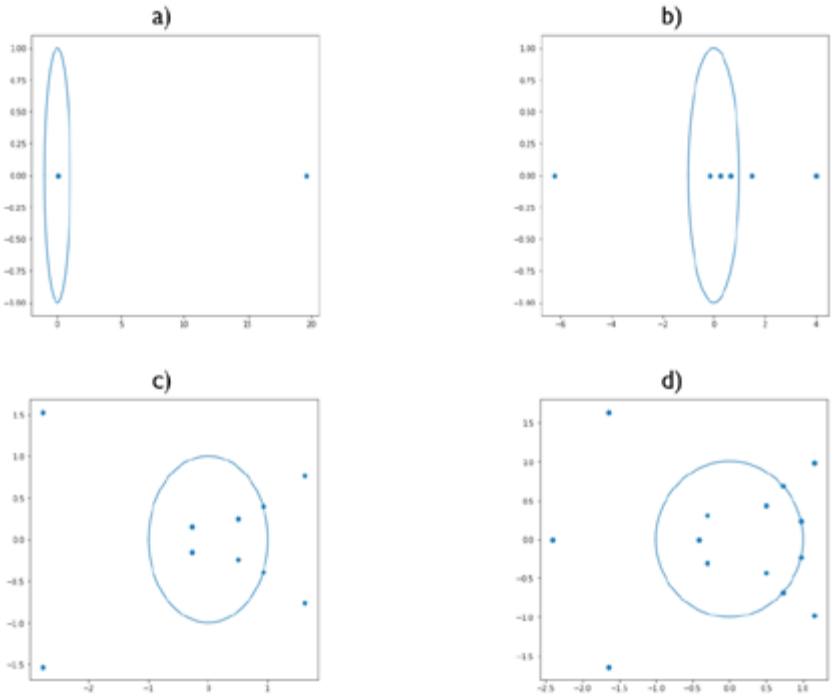


Figura 46. Gráfica de polos y ceros, filtro pasa-altos diseñado con la ventana hamming: a) M=3, b) M=7, c) M=11, d) M=15.

La principal diferencia en el comportamiento de la gráfica de polos y ceros entre filtros pasa-bajos y pasa-altos, es que en los últimos los ceros se concentran alrededor de $z = 1$ (ver Figura 46), mientras que en los primeros se concentran alrededor de $z = -1$. Esta observación es válida independiente del método de diseño del filtro y/o de la ventana seleccionada.

CAPÍTULO 5.
Metodos de diseño de filtros IIR

En este quinto capítulo del libro abordaremos el diseño de filtros de respuesta al impulso infinita (IIR), a partir del diseño de filtros análogos y aplicando mapeo entre el dominio Laplaciano y el dominio Z.

Al finalizar el capítulo, deberás estar en capacidad de:

1. Encontrar la TZ de señales de duración finita e infinita, así como su región de convergencia.
2. Diseñar filtros digitales aplicando Transformada Bilineal, con ayuda de Python.
3. Diseñar filtros Butterworth digitales, con ayuda de Python.
4. Encontrar la relación entre la frecuencia de corte del filtro análogo con la frecuencia de corte del filtro digital (o frecuencia de resonancia, en el caso de filtros pasa-banda de banda angosta).
5. Explicar el comportamiento de polos y ceros en filtros IIR.
6. Filtrar señales 1D con filtros IIR.

Una diferencia importante en el diseño de filtros digitales FIR con los IIR radica en que los segundos se diseñan a partir de un mapeo entre el dominio *Laplaciano* y el dominio Z. Teniendo en cuenta que la función de transferencia de los filtros análogos contiene un polinomio en el denominador dependiente de s, entonces, los filtros digitales obtenidos por el mapeo entre estos dos dominios contendrán en su función de transferencia un polinomio en el denominador dependiente de z. Por lo tanto, con esta técnica de diseño, siempre se obtendrán filtros IIR.

En este capítulo repasaremos algunos conceptos básicos de la TZ y posteriormente abordaremos dos ecuaciones de mapeo entre los dominios *Laplaciano* y z, una correspondiente a la aproximación en derivadas, y la otra, a la Transformada Bilineal. Aunque en la práctica la aproximación en derivadas es un método que no se utiliza por las limitaciones que tiene, permite entender el concepto de mapeo entre ambos dominios y facilita comprender en qué consiste la Transformada Bilineal. Por esa razón, la incluiremos en este capítulo.

5.1. CONCEPTOS BÁSICOS DE LA TZ

En el Capítulo 2.1 se presentó una breve introducción a la Transformada Z. En este Capítulo abordaremos el concepto de Región de Convergencia (ROC) de la TZ de la señal discreta, así:

> *"La ROC es el conjunto de todos los valores de z para los cuales la TZ de $x[n]$ es finita, es decir, que $X(z)$ converge a un valor".*

Si no se logra satisfacer la condición anterior con ningún valor de z, entonces se dice que la TZ de la señal no existe.

A continuación, aplicaremos el concepto de ROC a varios casos. Inicialmente para señales de duración finita y posteriormente para señales de duración infinita.

Caso 1: señal de duración finita causal.

Supongamos que $x[n] = \delta[n] + 2\delta[n - 1] + \delta[n - 2]$. Entonces, la TZ de la señal es $X(z) = z^0 + 2z^{-1} + z^{-2}$, ó de forma equivalente, $X(z) = 1 + \frac{2}{z} + \frac{1}{z^2}$.

Ahora bien, ¿existe algún valor o un conjunto de valores de z para los cuales $X(z)$ no sea finita? Específicamente, si $z = 0$ entonces $X(z) = 1 + \frac{2}{0} + \frac{1}{0^2} \cdot \rightarrow \infty$, es decir, $X(z)$ no converge, y entonces ese valor queda por fuera de la ROC.

Por lo cual, la ROC de la señal se expresa así:

$$ROC = todo\ plano\ z - \{z = 0\}.$$

Para $z \neq 0$ se tiene que $X(z)$ es finita.

Caso 2: señal de duración finita anti-causal

Utilizaremos la señal $x[n] = \delta[n + 2] + 5\delta[n + 1]$, cuya TZ es $X(z) = z^2 + 5z^1$. Para este caso, cuando $z = 0$ se tiene que $X(z)$ es finita, y entonces hace parte de su ROC. Analicemos entonces si para otro valor de z se tendría que $X(z)$ no es finita. Específicamente, si $z = \infty$ se tiene que $X(z) = \infty^2 + 5\infty^5 \rightarrow \infty$, por lo cual se debe excluir este valor de la ROC, quedando expresada de la siguiente manera:

$$ROC = todo\ plano\ z - \{z = \infty\}.$$

Caso 3: señal de duración infinita causal

Partiremos de la señal

$$x[n] = \alpha^n\, u[n]$$

Para conocer su comportamiento, dibujaremos la señal para algunos valores

de n teniendo en cuenta que esta señal inicia en $n = 0$ y termina en $n = \infty$. En la Figura 47 se presenta un ejemplo de la señal para algunos valores de n y $\alpha = 1/2$.

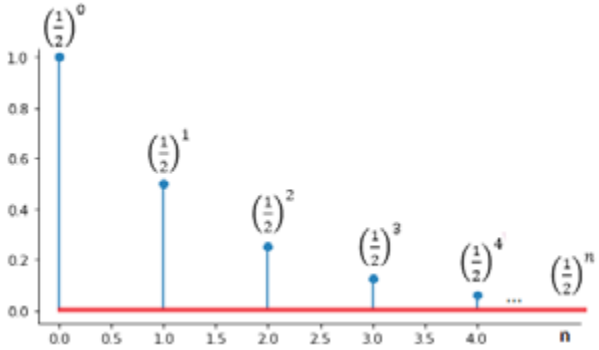


Figura 47. Gráfica de la señal $(1/2)^n\, u[n]$.

De forma general, si $0 < \alpha < 1$ la señal es decreciente, pero si $\alpha > 0$ es entonces creciente. Si α es negativa, tendrá un comportamiento oscilatorio (valores positivos y negativos alternados).

El código en Python para dibujar n muestras de la señal es:

```
import numpy as np
import matplotlib.pyplot as plt
a=0.5
n=5
n= np.linspace(0, n-1, n)
x = a **n
plt.stem(n, x)
```

La TZ de esta señal es:

$$X(z) = \alpha^0 z^0 + \alpha^1 z^{-1} + \alpha^2 z^{-2} + \alpha^3 z^{-3} + \cdots + \alpha^\infty z^{-\infty}$$

Teniendo en cuenta que la cantidad de términos de la expresión anterior es infinita, nos apoyamos en la siguiente serie matemática:

$$1 + A + A^2 + A^3 + \cdots + A^\infty = \frac{1}{1-A} \quad \leftrightarrow \quad |A| < 1$$

De tal forma que, al comparar las dos ecuaciones anteriores encontramos una similitud entre ellas cuando $A=\alpha/z$. La ROC quedaría entonces como $|\alpha/z| < 1$, o de forma equivalente $|z| > |\alpha|$.

De tal forma que, podemos reescribir la TZ de la señal, así:

$$X(z) = \frac{1}{1-\left(\frac{\alpha}{z}\right)} = \frac{1}{1-\alpha z^{-1}} \quad \leftrightarrow \quad ROC: |z| > |\alpha|$$

Para el caso específico de $\alpha = 0.5$, se tiene que su TZ es:

$$X(z) = \frac{1}{1-0.5z^{-1}} \quad \leftrightarrow \quad ROC: |z| > 0.5$$

Por lo cual, la ROC de esta señal causal de duración infinita es el exterior de un círculo de radio α.

Caso 4: señal de duración infinita anti-causal

Partiremos de la señal

$$x[n] = -\beta^n u[-n-1]$$

Esta señal existe desde n= -∞ hasta n = -1. Para los demás valores de n, su amplitud es cero. La siguiente figura presenta su comportamiento para algunos valores de n, y con β = 2.

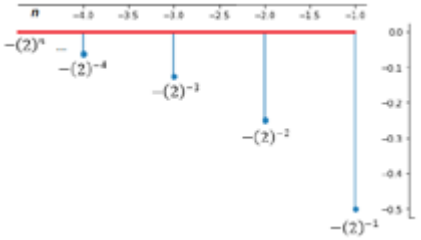


Figura 48. Gráfica de la señal -(2)^n u[-n-1].

Se debe tener en cuenta que el signo negativo está por fuera de la potencia n, de tal forma que toda la amplitud de la señal se invierte. Ahora bien, si $\beta > 1$, entonces se tiene una señal que disminuye en amplitud a medida que se aleja del origen en valores negativos de n.

El código en Python para dibujar n muestras de la señal es:

```
import numpy as np
import matplotlib.pyplot as plt
a=2
n=4
n= np.linspace(-n, -1, n)
x = -(a **n)
plt.stem(n, x)
```

La TZ de esta señal es:

$$X(z) = -\{\beta^{-1}z^1 + \beta^{-2}z^2 + \beta^{-3}z^3 + \cdots + \beta^{-\infty}z^{\infty}\}$$

De forma similar al caso anterior, nos apoyamos en la siguiente serie matemática:

$$1 + A + A^2 + A^3 + \cdots + A^{\infty} = \frac{1}{1-A} \quad \leftrightarrow \quad |A|<1$$

Pasando el valor de 1 al lado derecho de la ecuación, tenemos que:

$$A + A^2 + A^3 + \cdots + A^{\infty} = \frac{1}{1-A} - 1 \quad \leftrightarrow \quad |A|<1$$

Y resolviendo,

$$A + A^2 + A^3 + \cdots + A^{\infty} = \frac{A}{1-A} \qquad \leftrightarrow \qquad |A|<1$$

Finalmente, multiplicamos a ambos lados de la ecuación por -1, obteniendo que:

$$-\{A + A^2 + A^3 + \cdots + A^{\infty}\} = \frac{A}{A-1} \qquad \leftrightarrow \qquad |A|<1$$

La TZ de la señal se parece a la serie anterior cuando $A = z/\beta = \beta^{-1}z$. Y la ROC quedaría como $|\beta^{-1}z| < 1$, o de forma equivalente, $|z| < |\beta|$.

Entonces, podemos reescribir la TZ de la señal, así:

$$X(z) = \frac{\beta^{-1}z}{\beta^{-1}z-1}$$

O de forma equivalente,

$$\frac{1}{1-\beta z-1} \leftrightarrow \quad ROC: |z|<|\beta|$$

Para el caso específico de $\beta = 2$, se tiene que su TZ es:

$$X(z) = \frac{1}{1-2z^{-1}} \leftrightarrow \quad ROC: |z| < 2$$

Entonces, la ROC de esta señal anti-causal de duración infinita es el interior de un círculo de radio β.

Caso 5: señal de duración infinita por ambos lados de *n*

Partiremos de la señal

$$x[n] = \alpha^n u[n] - \beta^n u[-n-1]$$

Para $x_1[n] = \alpha^n u[n]$, y $x_2[n] = -\beta^n u[-n-1]$, es decir que, $x[n] = x_1[n] + x^2[n]$.

Teniendo en cuenta lo presentado en el Caso 3 y Caso 4 de este Capítulo, se tiene que:

$$X(z) = \frac{1}{1-\alpha z^{-1}} + \frac{1}{1-\beta z^{-1}}$$

Con ROC: $|z| > |\alpha| \cap |z| < |\beta|$

De tal forma que, la TZ $\exists \leftrightarrow \beta > \alpha$. En caso contrario, $\nexists$.

5.2. APROXIMACIÓN EN DERIVADAS

El concepto que vamos a aplicar en esta subsección y la siguiente es el de mapeo. Pero ¿qué significa exactamente mapear dos dominios? Según Britannica[3], la defini-

3 Disponible en: https://www.britannica.com/science/mapping

ción de mapeo es *"cualquier forma prescrita de asignar a cada objeto en un conjunto a un objeto en particular en otro (o el mismo) conjunto"*. Entonces, para nuestro caso, el mapeo permite relacionar el dominio Laplaciano con el dominio z a través de una función.

En el caso del método de aproximación en derivadas, se mapea la función de transferencia $H(s)$ con la función de transferencia $H(z)$, correspondiente a la derivada. En el dominio Laplaciano la función de transferencia de la derivada es $H(s) = s$, mientras que en el dominio z es $H(z) = (1 - z^{-1})/T_s$, donde T_s corresponde al periodo de muestreo del sistema (es decir, el espaciamiento entre muestras consecutivas, sabiendo que $T_s = 1/f_s$).

La Figura 49 nos permite ilustrar el concepto de derivada. Supongamos que queremos calcular la derivada de una señal discreta en un tiempo específico n, denominada $m(n)$, la cual se define como el incremento en amplitud de la señal dividido en el periodo de muestreo, T_s, de la forma:

$$m(n) = \frac{x(n) - x(n-1)}{T_s} \qquad \text{Ecuación 39}$$

Por ejemplo, para $n = 7$, tendremos que su derivada es $m(7) = \{x(7) - x(6)\}/T_s$.

Entonces, si la salida del sistema es la derivada de la señal de entrada, para todo valor de n tendremos la siguiente ecuación de entrada-salida:

$$y[n] = \frac{x[n] - x[n-1]}{T_s} \qquad \text{Ecuación 40}$$

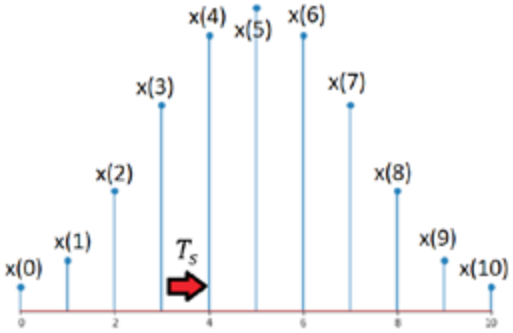


Figura 49. Señal discreta: concepto de derivada.

Aplicando la TZ a cada uno de los términos de la ecuación anterior y la propiedad de desplazamiento de la TZ, tendremos que:

$$Y(z) = \frac{X(z) - z^{-1}X(z)}{T_s} = \frac{X(z)\{1 - z^{-1}\}}{T_s} \qquad \text{Ecuación 41}$$

De tal forma que la función de transferencia nos queda así:

$$\frac{Y(z)}{X(z)} = H(z) = \frac{\{1-z-1\}}{T_s} \qquad \text{Ecuación 42}$$

Relacionando las dos funciones de transferencia $H(s)$ con la de $H(z)$, obtenemos el mapeo entre el dominio s y el dominio z:

$$s = \frac{(1-z^{-1})}{T_s} \qquad \text{Ecuación 43}$$

Despejando z de la ecuación anterior, obtenemos:

$$z = \frac{1}{1-(sT_s)} \qquad \text{Ecuación 44}$$

Finalmente, al reemplazar $s = j\Omega$, obtendremos:

$$z = \frac{1}{1-j\Omega T_s} = \frac{1}{1-j\Omega T_s} * \frac{1+j\Omega T_s}{1+j\Omega T_s} = \frac{1+j\Omega T_s}{1-(j\Omega T_s)^2} = \frac{1+j\Omega T_s}{1-j^2\Omega^2 {T_s}^2}$$

Entonces,

$$z = \frac{1}{1+\Omega^2 {T_s}^2} + \frac{j\Omega T_s}{1+\Omega^2 {T_s}^2} \qquad \text{Ecuación 45}$$

Donde Ω corresponde a la frecuencia de corte del filtro análogo. Al variar Ω en el rango {-∞, ∞} se obtiene una correspondencia en el plano z de un círculo de $r = 0.5$ y centro en $z = 0.5$. De tal forma que un filtro análogo estable (el cual tiene sus polos en el semiplano izquierdo), se transforma en un filtro digital estable (el cual tiene sus polos dentro del círculo unitario). La principal desventaja de este método de diseño de filtros IIR consiste en que la ubicación de los polos en ese pequeño círculo corresponde a frecuencias bajas. De tal forma que solamente se pueden diseñar filtros con valores de ΩT_s pequeños.

5.3. TRANSFORMADA BILINEAL

La Transformada Bilineal es una mejora del método de aproximación en derivadas, dado que se mapea todo el semi-plano izquierdo del dominio Laplaciano, aprovechando todo el interior del círculo unitario. Como consecuencia, se pueden diseñar filtros de cualquier frecuencia de corte, superando la limitación que tenía el método de aproximación en derivadas.

La ecuación que nos permite mapear ambos dominios es:

$$s = \frac{2}{T_s}\left(\frac{1-z^{-1}}{1+z^{-1}}\right) \qquad \text{Ecuación 46}$$

Con esta función de mapeo, todo el semiplano izquierdo en el dominio s se corresponde con el interior del círculo unitario en el dominio z.

En el dominio Laplaciano, un filtro pasa bajo tiene un cero en $s = \infty$. Cuando se aplica la ecuación 46, el cero del filtro digital queda ubicado en en $z = -1$.

Adicionalmente, la correspondencia entre la frecuencia de corte del filtro análogo

(Ω_a) con la del filtro digital (ω_d) no es lineal, sino una relación de tipo tangencial, dado por la ecuación:

$$\Omega_a \cong \frac{2}{T_s} tg\left(\frac{\omega_d}{2}\right) \qquad \text{Ecuación 47}$$

A partir de la ecuación anterior, podemos encontrar la frecuencia de corte (o de resonancia) del filtro digital a partir de la del filtro análogo, así:

$$\omega_d \cong 2tg^{-1}\left(\frac{\Omega_a * T_s}{2}\right) \qquad \text{Ecuación 48}$$

Como siguiente paso, necesitamos recordar las funciones de transferencia de filtros análogos. Trabajaremos con filtros de segundo orden.

Tipo de filtro	Función de transferencia	Variables de diseño
Pasa-bajo	$H(s) = \frac{G{\Omega_c}^2}{s^2 + (2\varsigma\Omega_c)s + {\Omega_c}^2}$	G: ganancia del filtro Ω_c: frecuencia de corte del filtro análogo ς: factor de amortiguamiento
Pasa-alto	$H(s) = \frac{Gs^2}{s^2 + (2\varsigma\Omega_c)s + {\Omega_c}^2}$	G: ganancia del filtro Ω_c: frecuencia de corte del filtro análogo ς: factor de amortiguamiento
Pasa-banda	$H(s) = \frac{G\left(\frac{\Omega_r}{Q}\right)s}{s^2 + \left(\frac{\Omega_r}{Q}\right)s + {\Omega_r}^2}$	G: ganancia del filtro Ω_r: frecuencia de resonancia del filtro análogo Q: factor de calidad

Se aclara que las frecuencias de los filtros análogos de las funciones de transferencia de la tabla anterior están en unidades de $[rad/seg]$.

Con el siguiente ejemplo se ilustra el método de diseño de filtro IIR con la Transformada Bilineal, apoyado en Python.

Ejemplo 1: filtro pasa-altos

Se quiere diseñar un filtro digital utilizando Transformada Bilineal, a partir de un filtro análogo pasa-alto, con $\Omega_c = 100[Hz]$, $\varsigma = 1$, y $G = 1$. La frecuencia de muestreo, f_s, es 10 veces la frecuencia de corte del filtro análogo.

El primer paso consiste en convertir la frecuencia de corte que inicialmente está en $[Hz]$ en unidades $[rad/seg]$. Posteriormente, escribir la función de transferencia en el dominio análogo, teniendo en cuenta el tipo de filtro, así:

$$H(s) = \frac{1*s^2}{s^2 + (2 * 1 * 100 * 2 * \pi)s + (100 * 2 * \pi)^2}$$

A partir de $H(s)$ se escribe el siguiente código en Python:

i

```
mport numpy as np
from scipy import signal
import matplotlib.pyplot as plt
f = 100
frad = 2*3.14*f
amort = 1
G = 1
nums=np.array([G, 0, 0])
dens=np.array([1, 2*amort*frad, frad*frad])
fs=10 *f
ws, hs = signal.freqs(nums, dens)
```

Del código anterior, f es la frecuencia de corte del filtro análogo en unidades $[Hz]$, *frad* es la frecuencia de corte del filtro análogo en unidades $[rad/seg]$, *amort* es el factor de amortiguamiento, y G es la ganancia del filtro. Adicionalmente, *nums* es el vector del polinomio del numerador de $H(s)$, *dens* es el vector del polinomio del denominador de $H(s)$, y fs es la frecuencia de muestreo del sistema. Teniendo en cuenta que ambos polinomios (numerador y denominador) son de segundo orden, entonces cada vector contiene tres valores, el primero asociado a s^2, el segundo a s^1 y el tercero a s^0.

Con la instrucción *signal.freqs* se calcula la respuesta en frecuencia del filtro análogo. La salida ws corresponde al vector de frecuencias; mientras que, hs es el vector de amplitudes de $H(s)$.

Para graficar la respuesta en frecuencia, escribimos el siguiente código:

```
plt.plot(ws, (np.abs(hs)), label=r'$|H(s)|$')
plt.legend()
plt.xlabel('Frecuencia [rad/seg]')
plt.ylabel('Magnitud')
plt.title('Respuesta en frecuencia del filtro análogo')
plt.grid()
```

Obteniendo la siguiente gráfica:

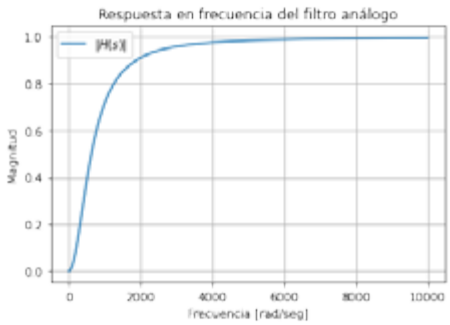


Figura 50. Respuesta en frecuencia filtro análogo pasa-alto, Ωc=200π $[rad\diagup seg]$.

Como siguiente paso, convertimos $H(s)$ en $H(z)$, aplicando Transformada Bilineal. Para ello, utilizamos la instrucción **signal.bilinear* y posteriormente creamos el sistema LTI con la instrucción *signal.dlti.* A continuación, con *signal.freqz* calculamos

la respuesta en frecuencia del filtro digital (a partir de los vectores del numerador y denominador de $H(z)$), y la graficamos.

```
filtz = signal.dlti(*signal.bilinear(nums, dens, fs))
wz, hz = signal.freqz(filtz.num, filtz.den)
plt.plot(wz, (np.abs(hz)), label=r'$|H(z)|$')
plt.legend()
plt.xlabel('Frecuencia normalizada [rad/muestra]')
plt.ylabel('Magnitud')
plt.title('Respuesta en frecuencia del filtro digital')
plt.grid()
```

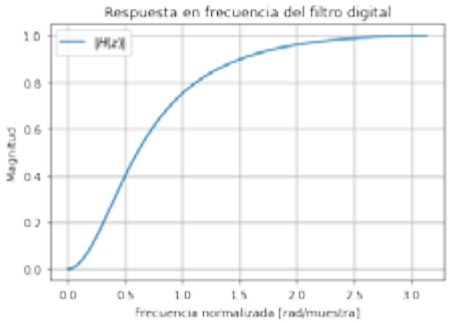


Figura 51. Respuesta en frecuencia filtro digital pasa-alto, $\omega_d = 0.6$ $[rad/muestra]$ $\varsigma = 1$.

Teniendo en cuenta que $\varsigma = 1$, entonces la frecuencia de corte corresponde a la ganancia de 0.5. Al revisar la Figura anterior, este valor se encuentra en 0.6 $[rad/muestra]$, aproximadamente.

A nivel teórico, calculamos la frecuencia de corte con el siguiente código:

```
wd =2*np.arctan(frad/(fs*2))
wd
```

Obteniendo

```
0.608501664475969
```

Coincidiendo el valor teórico con el encontrado a partir de la gráfica del filtro digital.

Por otro lado, podemos escribir la función de transferencia del filtro digital, a partir de los vectores `filtz.num` y `filtz.den`.

Sabiendo que,

```
filtz.num
```

```
array([ 0.57917428, -1.15834857, 0.57917428])
```

```
filtz.den
```

```
array([ 1. , -1.04414003, 0.2725571 ])
```

Entonces, escribimos $H(z)$, partiendo de z^0 en el primer término del polinomio, tanto del numerador como del denominador, obteniendo que:

$$H(Z) = \frac{0.57917428 - 1.15834857z^{-1} + 0.57917428z^{-2}}{1 - 1.04414003z^{-1} + 0.2725571z^{-2}}$$

Como siguiente paso, se calculan los ceros, polos y ganancia del filtro digital, utilizando `signal.tf2zpk`, así:

```
z, p, k = signal.tf2zpk(filtz.num,filtz.den)
print(z)
print(p)
print(k)
```

Obteniendo:

```
[1. 1.]
[0.52207002 0.52207002]
0.5791742828084856
```

Finalmente, se grafican los polos y ceros del filtro digital:

```
theta = np.linspace(-np.pi,np.pi,201)
plt.rcParams["figure.figsize"] = (5,5)

plt.plot(np.cos(theta),np.sin(theta))
plt.scatter(np.real(z),np.imag(z), marker='o')
plt.scatter(np.real(p),np.imag(p), marker='x')
plt.title('Gráfica polos y ceros filtro digital')
```

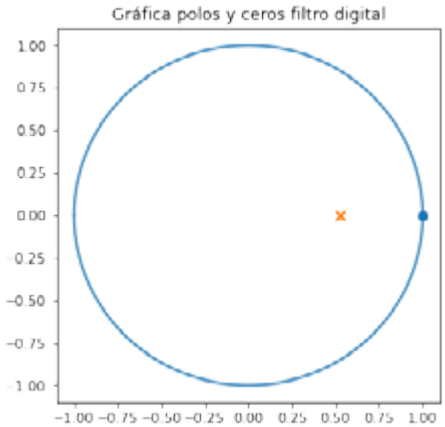


Figura 52. Gráfica de polos y ceros del filtro digital pasa-alto, ω_d=0.6 $[rad/muestra]$.

A partir de la gráfica anterior, se pueden extraer las siguientes conclusiones:

1.Si el filtro es pasa-alto, los dos ceros se ubican en z=1.

2.Los polos están relacionados con la frecuencia de corte del filtro digital. En este caso se ubican en el semicírculo derecho del plano z, dado que $(\omega_d < \pi)/2$.

Ejemplo 2: filtro pasa-bajos

Se quiere diseñar un filtro digital utilizando Transformada Bilineal, a partir de un filtro análogo pasa-bajo, con $\Omega_c = 100\ [Hz]$, ς=0.707, y G=1. La frecuencia de muestreo es 4 veces la frecuencia de corte del filtro análogo.

El primer paso consiste en convertir la frecuencia de corte que inicialmente está en $[Hz]$, en unidades $[rad/seg]$. Posteriormente, escribir la función de transferencia en el dominio análogo, teniendo en cuenta el tipo de filtro, así:

$$H(s) = \frac{1 * (100 * 2 * \pi)^2}{s^2 + (2 * 0.707 * 100 * 2 * \pi)s + (100 * 2 * \pi)^2}$$

A partir de $H(s)$ se escribe el siguiente código en Python:

```
import numpy as np
from scipy import signal
import matplotlib.pyplot as plt
f = 100
frad = 2*3.14*f
amort = 0.707
G = 1
nums=np.array([0, 0, G*frad*frad])
dens=np.array([1, 2*amort*frad, frad*frad])
ws, hs = signal.freqs(nums, dens)
plt.plot(ws, (np.abs(hs)), label=r'$|H(s)|$')
plt.legend()
plt.xlabel('Frecuencia [rad/seg]')
plt.ylabel('Magnitud')
plt.title('Respuesta en frecuencia del filtro análogo')
plt.grid()
```

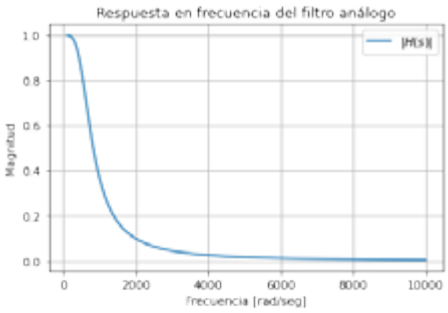


Figura 53. Respuesta en frecuencia filtro análogo pasa-bajo, $\Omega_c = 200\pi$ [$[rad/seg]$.].

Posteriormente, se utiliza la Transformada Bilineal para encontrar la función de transferencia del filtro digital, con el siguiente código:

```
fs=4 *f
filtz = signal.dlti(*signal.bilinear(nums, dens, fs))
wz, hz = signal.freqz(filtz.num, filtz.den)
plt.plot(wz, (np.abs(hz)), label=r'$|H(z)|$')
plt.legend()
plt.xlabel('Frecuencia normalizada [rad/muestra] ')
plt.ylabel('Magnitud')
plt.title('Respuesta en frecuencia del filtro digital')
plt.grid()
```

Obteniendo,

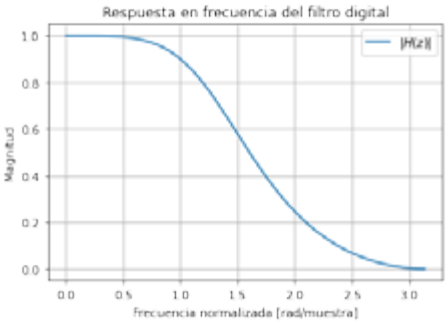


Figura 54. Respuesta en frecuencia filtro digital pasa-bajo, $\omega_d = 0.133$ [*rad/muestra*], $\varsigma = 0.707$.

La ganancia en la frecuencia de corte es 0.707, dado que $\varsigma = 0.707$. Entonces la frecuencia de corte del filtro digital a partir de la gráfica es:

```
x = np.where(abs(hz) > 0.707)
wcd = np.max(x)*3.14/len(wz)
print(wcd)
```

```
1.3246875
```

A nivel teórico, la frecuencia de corte la encontramos con la siguiente ecuación:

```
wd =2*np.arctan(frad/(fs*2))
wd
```

```
1.3310548874510058
```

Los dos valores anteriores son muy cercanos, entonces el filtro quedó bien diseñado.

Como siguiente paso, encontramos las constantes de los polinomios del numerador y denominador de $H(z)$, así:

```
filtz.num
```

```
array([0.22603683, 0.45207366, 0.22603683])
```

```
filtz.den
```

```
array([ 1. , -0.28154419, 0.18569152])
```

Y escribimos la función de transferencia del filtro digital:

$$H(z) = \frac{0.22603683 + 0.45207366z^{-1} + 0.22603683z^{-2}}{1 - 0.28154419z^{-1} + 0.18569152z^{-2}}$$

Los ceros, polos y ganancia de $H(z)$, la encontramos con el siguiente código:

```
z, p, k = signal.tf2zpk(filtz.num,filtz.den)
print(z)
print(p)
print(k)
```

Obteniendo:

```
[-1. -1.]
[0.1407721+0.40727722j 0.1407721-0.40727722j]
0.22603683128439978
```

Es decir, el filtro tiene dos ceros en z = -1, y dos polos muy cercanos al eje vertical del plano z.

La gráfica de polos y ceros del filtro se obtiene con el siguiente código:

```
theta = np.linspace(-np.pi,np.pi,201)
plt.rcParams["figure.figsize"] = (5,5)
plt.plot(np.cos(theta),np.sin(theta))
plt.scatter(np.real(z),np.imag(z), marker='o')
plt.scatter(np.real(p),np.imag(p), marker='x')
plt.title('Gráfica polos y ceros filtro digital')
```

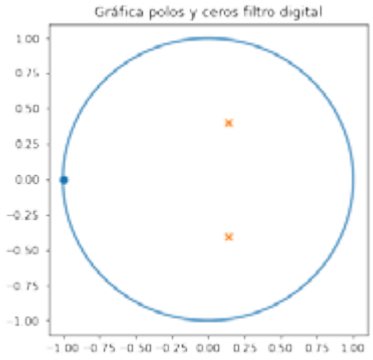


Figura 55. Gráfica de polos y ceros del filtro digital pasa-bajo, ω_d=1.33 [*rad/muestra*].

A partir de la gráfica anterior, se pueden extraer las siguientes conclusiones:

1. Si el filtro es pasa-bajo, los dos ceros se ubican en z = -1.
2. Los dos polos se ubican en el semicírculo derecho del plano z (muy cerca del eje vertical), dado que la frecuencia de corte, ω_d, es ligeramen te menor a $\pi/2$.

Ejemplo 3: filtro pasa-banda de banda angosta

Se quiere diseñar un filtro digital utilizando Transformada Bilineal, a partir de un filtro análogo pasa-banda, con $\Omega_r = 100\ [Hz]$, $Q = 2$, y $G = 1$. La frecuencia de muestreo es 3 veces la frecuencia de corte del filtro análogo.

El primer paso consiste en convertir la frecuencia de corte que inicialmente está en $H(z)$ en unidades $[rad/seg]$. Posteriormente, escribir la función de transferencia en el dominio análogo, teniendo en cuenta el tipo de filtro, así:

$$H(s) = \frac{1 * \frac{100 * 2 * \pi}{2}}{s^2 + \left(\frac{100 * 2 * \pi}{2}\right) s + (100 * 2 * \pi)^2}$$

A partir de $H(s)$ se escribe el siguiente código en Python:

```
import numpy as np
from scipy import signal
import matplotlib.pyplot as plt
f = 100
frad = 2*3.14*f
Q = 2
G = 1
nums=np.array([0, G*frad/Q, 0])
dens=np.array([1, frad/Q, frad*frad])

ws, hs = signal.freqs(nums, dens)
plt.plot(ws, (np.abs(hs)), label=r'$|H(s)|$')
plt.legend()
plt.xlabel('Frecuencia [rad/seg]')
plt.ylabel('Magnitud')
plt.title('Respuesta en frecuencia del filtro análogo')
plt.grid()
```

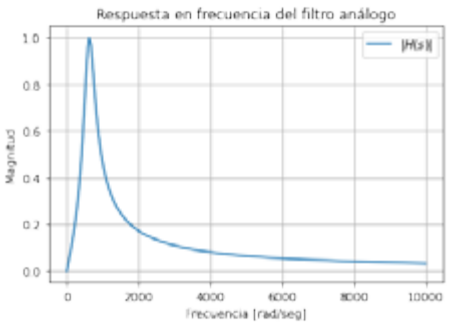


Figura 56. Respuesta en frecuencia filtro análogo pasa-banda, $\Omega_r = 200\pi\ [rad/seg]$.

Y se convierte el filtro análogo en digital con la Transformada Bilineal, así:

```
fs=3 *f
filtz = signal.dlti(*signal.bilinear(nums, dens, fs))
wz, hz = signal.freqz(filtz.num, filtz.den)
plt.plot(wz, (np.abs(hz)), label=r'$|H(z)|$')
plt.legend()
plt.xlabel('Frecuencia normalizada [rad/muestra]')
plt.ylabel('Magnitud')
plt.title('Respuesta en frecuencia del filtro digital')
plt.grid()
```

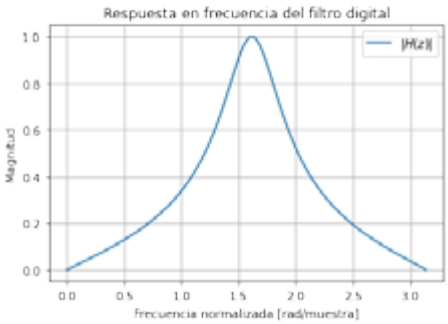


Figura 57. Respuesta en frecuencia filtro digital pasa-banda, $\omega_d = 1.61$ $[rad/muestra]$, $Q = 2$.

La frecuencia de resonancia la encontramos a partir de la gráfica anterior, con el siguiente código en Python:

```
x = np.where(abs(hz) > 0.999)
wcd = np.max(x)*3.14/len(wz)
print(wcd)
```

```
1.6251953125
```

A nivel teórico, la frecuencia de resonancia del filtro digital es:

```
wd =2*np.arctan(frad/(fs*2))
wd
```

```
1.6163910321996993
```

Los valores anteriores son muy similares entre sí, entonces hemos verificado que el filtro quedó bien diseñado.

Las constantes de los polinomios del numerador y denominador de la función de transferencia del filtro digital son:

```
filtz.num
```

```
array([ 0.19983368, 0. , -0.19983368])
```

```
filtz.den
```

```
array([1. , 0.07294142, 0.60033263])
```

De tal forma que H(z) es:

H(z)=(0.19983368+⟦-0.19983368z⟧^(-2))/(1-0.07294142z^(-1)+⟦0.60033263z⟧^(-2))

$$H(z) = \frac{0.19983368 + 0.19983368z^{-2}}{1 - 0.07294142z^{-1} + 0.60033263z^{-2}}$$

Los ceros, polos y ganancia del filtro digital se calculan con el siguiente código en Python:

```
z, p, k = signal.tf2zpk(filtz.num,filtz.den)
print(z)
print(p)
print(k)
```

```
[-1.  1.]
[-0.03647071+0.77395253j -0.03647071-0.77395253j]
0.1998336840676125
```

Y se grafican con el siguiente código:

```
theta = np.linspace(-np.pi,np.pi,201)
plt.rcParams["figure.figsize"] = (5,5)

plt.plot(np.cos(theta),np.sin(theta))
plt.scatter(np.real(z),np.imag(z), marker='o')
plt.scatter(np.real(p),np.imag(p), marker='x')
plt.title('Grafica polos y ceros filtro digital')
```

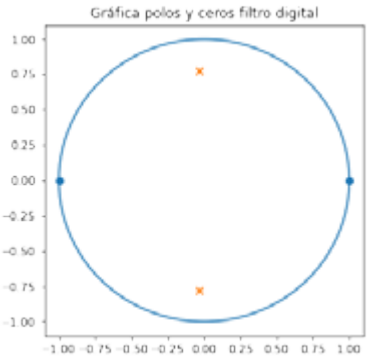


Figura 58. Gráfica de polos y ceros del filtro digital pasa-banda, ω_d=1.61 [*rad/muestra*].

A partir de la gráfica anterior, se pueden extraer las siguientes conclusiones:

1. En este caso, existe un cero en z = -1 y otro en z = 1, dado que el filtro es pasa-banda.
2. Los dos polos se ubican en el semicírculo izquierdo del plano z, dado que la frecuencia de resonancia, ω_d, es mayor a $\pi/2$.

5.4. FILTRO BUTTERWORTH

En esta última sección de diseño de filtros IIR, trabajaremos con los filtros Butterworth, los cuales se caracterizan por:

- Respuesta plana en la banda de paso.
- En la frecuencia de corte tiene una ganancia de -3 dB en escala logarítmica, o de $\sqrt{2}/2$ en escala lineal, respecto a la amplitud en la banda de paso.
- $H(s)$ solamente posee polos.

Apoyándonos en Python tenemos dos estrategias para diseñar los filtros Butterworth, las cuales son:

Diseñar un filtro análogo Butterworth y aplicar Transformada Bilineal.

Diseñar directamente el filtro digital Butterworth.

A continuación, exploraremos las dos estrategias de diseño, a partir de ejemplos.

Ejemplo 1: filtro Butterworth análogo y Transformada Bilineal

Se quiere diseñar un filtro Butterworth pasa-bajo, a partir de un filtro análogo y aplicando Transformada Bilineal, para diferentes valores de orden del filtro (específicamente, $N = 2,4,6,8,10$). La frecuencia de corte del filtro análogo es $\Omega_c = 100\ [Hz]$.

- Graficar la respuesta en frecuencia del filtro análogo, para $N = 2,4,6,8,10$.
- Escribir $H(s)$ cuando $N = 2$.
- Calcular el filtro digital a partir del filtro análogo aplicando Transformada Bilineal, con $f_s = 10 * \Omega_c$. Graficar la respuesta en frecuencia del filtro digital Butterworth, para $N = 2,4,6,8,10$
- Escribir $H(z)$ cuando $N = 2$.
- Obtener los polos y ceros del filtro digital Butterworth, para $N = 2,4,6,8,10$ Graficar los polos y ceros del filtro digital Butterworth, para $N = 2,4,6,8,10$.

Respuesta en frecuencia del filtro análogo, $N = 2,4,6,8,10$:

```
import numpy as np
from scipy import signal
import matplotlib.pyplot as plt
f = 100
wn = f * 2 * np.pi
```

```
N = 2
b2,a2 = signal.iirfilter(N, wn, btype='lowpass', analog=True,
ftype='butter')
ws2, hs2 = signal.freqs(b2, a2)
wsHz2=ws2/(2*np.pi)
plt.rcParams["figure.figsize"] = (14,8)
plt.plot(wsHz2, (np.abs(hs2)), label=r'$|H(s)| con N=2$')
plt.legend()
plt.xlabel('Frecuencia [Hz]')
plt.ylabel('Magnitud')
plt.title('Respuesta en frecuencia filtro pasa bajo')
plt.grid()
```

```
N = 4
b4,a4 = signal.iirfilter(N, wn, btype='lowpass', analog=True,
ftype='butter')
ws4, hs4 = signal.freqs(b4, a4)
wsHz4=ws4/(2*np.pi)
plt.plot(wsHz4, (np.abs(hs4)), label=r'$|H(s)| con N=4$')
plt.legend()
```

```
N = 6
b6,a6 = signal.iirfilter(N, wn, btype='lowpass', analog=True,
ftype='butter')
ws6, hs6 = signal.freqs(b6, a6)
wsHz6=ws6/(2*np.pi)
plt.plot(wsHz6, (np.abs(hs6)), label=r'$|H(s)| con N=6$')
plt.legend()
```

```
N = 8
b8,a8 = signal.iirfilter(N, wn, btype='lowpass', analog=True,
ftype='butter')
ws8, hs8 = signal.freqs(b8, a8)
wsHz8=ws8/(2*np.pi)
plt.plot(wsHz8, (np.abs(hs8)), label=r'$|H(s)| con N=8$')
plt.legend()
```

```
N = 10
b10,a10 = signal.iirfilter(N, wn, btype='lowpass', analog=True,
ftype='butter')
ws10, hs10 = signal.freqs(b10, a10)
wsHz10=ws10/(2*np.pi)
plt.plot(wsHz10, (np.abs(hs10)), label=r'$|H(s)| con N=10$')
plt.legend()
plt.show()
```

Nota: Tener en cuenta que el vector de frecuencias, *ws*, se divide entre 2π, para que la gráfica quede en [*Hz*].

Obteniendo las siguientes respuestas en frecuencia del filtro análogo:

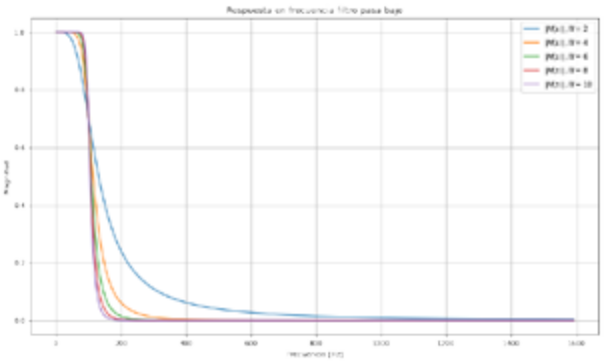


Figura 59. Respuesta en frecuencia filtro análogo Butterworth, $\Omega_c = 100[Hz]$ y $N = 2,4,6,8,10$.

De la figura anterior se puede identificar que independiente del orden del filtro, la ganancia en la frecuencia de corte es la misma, correspondiente a 0.707. Es decir, todas las curvas cruzan por el mismo valor de ganancia cuando $\Omega_c = 100[Hz]$. Adicionalmente, a medida que el valor de N aumenta, entonces la caída entre la banda de paso y la banda de rechazo se hace más pronunciada, es decir, mayor atenuación en las frecuencias cercanas a la de corte (se aproxima en mayor medida al filtro ideal).

Función de transferencia del filtro análogo, para $N = 2$:

Previamente se han encontrado las constantes de los polinomios tanto del numerador como del denominador del filtro análogo, en las variables *"b"* y *"a"*. Para el caso de $N = 2$, se utilizan $b2$ y $a2$.

```
b2
```

```
array([394784.17604357])
```

```
a2
```

```
array([1.00000000e+00, 8.88576588e+02, 3.94784176e+05])
```

A partir de los resultados anteriores, se tiene que:

$$H(s) \cong \frac{39.47*10^4}{s^2 + 8.88 * 10^2\, s + 3.94 * 10^5}$$

Nota: por simplicidad se expresó $H(s)$ solamente con dos cifras decimales.

Cálculo de $H(z)$ y respuesta en frecuencia del filtro digital:

Se utiliza `*signal.bilinear` para realizar el mapeo entre el filtro análogo y el filtro digital, `y signal.freqz` para la respuesta en frecuencia del filtro digital.

```
fs= 10*f # frecuencia de muestreo en Hz
filtz2 = signal.dlti(*signal.bilinear(b2, a2, fs))
wz2, hz2 = signal.freqz(filtz2.num, filtz2.den)
plt.plot(wz2, (np.abs(hz2)), label=r'$|H(z)|$') # se repite para los
demás valores de N
plt.legend()
plt.xlabel('Frecuencia normalizada [rad/muestra]')
plt.ylabel('Magnitud')
plt.title('Respuesta en frecuencia del filtro digital')
plt.grid()
```

Obteniendo las siguientes respuestas en frecuencia del filtro digital:

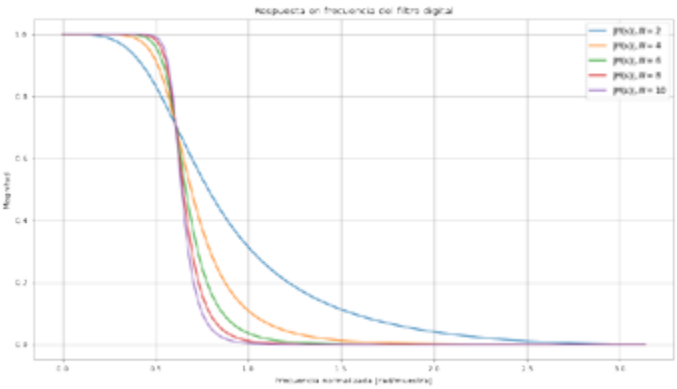


Figura 60. Respuesta en frecuencia filtro digital Butterworth, $\omega_d = 0.6$ [*rad/muestra*] y $N = 2,4,6,8,10$.

Teniendo en cuenta que se aplicó Transformada Bilineal para el diseño del filtro digital, entonces se utiliza la ecuación que relaciona la frecuencia del filtro análogo con la del filtro digital que se presentó en la sección 5.3, así:

```
fcdigital= 2 * np.arctan(wn/(2*fs))
print(fcdigital)
```

```
0.6087915947292302
```

Función de transferencia del filtro digital, para *N*=2:

A partir de los vectores filtz.num y filtz.den se encuentran las constantes de los polinomios del numerador y denominador de $H(z)$, respectivamente. Específicamente para $N = 2$, se utiliza `filtz2.num` y `filtz2.den`.

```
filtz2.num
```

```
array([0.06396438, 0.12792877, 0.06396438])
```

```
filtz2.den
```

```
array([ 1. , -1.16826067, 0.42411821])
```

Y entonces,

$$H(z) \cong \frac{0.0639 + 0.1279z^{-1} + 0.0639z^{-2}}{1 - 1.1683z^{-1} + 0.4241z^{-2}}$$

Nota: por simplicidad se expresó $H(z)$ solamente con cuatro cifras decimales.

Cálculo y gráfica de los polos y ceros del filtro digital, para N = 2,4,6,8,10:

```
z2, p2, k2 = signal.tf2zpk(filtz2.num,filtz2.den)
z4, p4, k4 = signal.tf2zpk(filtz4.num,filtz4.den)
z6, p6, k6 = signal.tf2zpk(filtz6.num,filtz6.den)
z8, p8, k8 = signal.tf2zpk(filtz8.num,filtz8.den)
z10, p10, k10 = signal.tf2zpk(filtz10.num,filtz10.den)
```

```
theta = np.linspace(-np.pi,np.pi,201)
plt.rcParams["figure.figsize"] = (5,5)

plt.plot(np.cos(theta),np.sin(theta))
plt.scatter(np.real(z2),np.imag(z2), marker='o')
plt.scatter(np.real(p2),np.imag(p2), marker='x')
plt.title('Gráfica polos y ceros filtro digital, N=2')
```

…

```
plt.plot(np.cos(theta),np.sin(theta))
plt.scatter(np.real(z10),np.imag(z10), marker='o')
plt.scatter(np.real(p10),np.imag(p10), marker='x')
plt.title('Gráfica polos y ceros filtro digital, N=10 ')
```

Y se obtienen las gráficas que se presentan en la Figura 61. Se puede apreciar que independiente del orden del filtro todos los ceros se ubican en z = -1 (por ser un filtro pasa-bajos), y que todos los polos se ubican en el semicírculo derecho (dado que $\omega_d < \pi/2$).

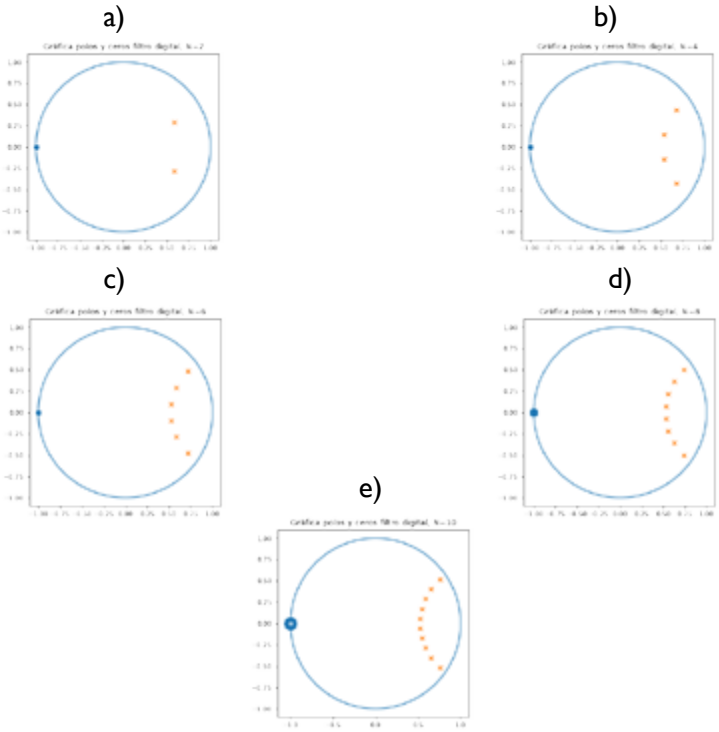


Figura 61. Gráfica de polos y ceros del filtro pasa pasa-bajo Buttherworth digital, ω_d = 0.6 [*rad/muestra*] y N = 2,4,6,8,10. Estrategia de diseño # 1.

Ejemplo 2: filtro Butterworh digital

Se quiere diseñar directamente un filtro Butterworth digital, correspondiente con un filtro análogo con $\Omega_c = 100\ [Hz]$, $f_s = 10 * \Omega_c$, y diferentes valores de orden del filtro, específicamente $N = 2,4,6,8,10$.

- Calcular Hz y graficar el filtro digital Butterworth, para $N = 2,4,6,8,10$.
- Escribir Hz cuando $N = 2$.
- Obtener los polos y ceros cuando $N = 2,4,6,8,10$. Graficar los polos y ceros cuando $N = 2,4,6,8,10$.

Como primer paso, debemos encontrar la frecuencia normalizada del filtro digital, la cual la podemos expresar como:

$$w_n = \frac{\Omega_c}{f_s/2} \qquad \text{Ecuación 49}$$

Que, en este caso es:

$$w_n = \frac{100}{\frac{1000}{2}} = 0.2$$

Nota: tener en cuenta que $0 < w_n < 1$.

Posteriormente, utilizamos la instrucción `signal.iirfilter`, haciendo `analog=False`.

Cálculo de $H(z)$ y gráfica de la respuesta en frecuencia del filtro digital:

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

N = 2
f = 100
fs = 10*f
wn = f/(fs/2)
b2, a2 = signal.iirfilter(N, wn, btype='lowpass', analog=False,
ftype='butter')
wz2, hz2 = signal.freqz(b2, a2, fs)
plt.rcParams["figure.figsize"] = (14,8)
plt.plot(wz2, (np.abs(hz2)), label=r'$|H(z)|, N=2$')
plt.legend()
plt.xlabel('Frecuencia normalizada [rad/muestra]')
plt.ylabel('Magnitud')
plt.title('Rta frecuencia filtro digital Butterworth')
plt.grid()
```

```
N = 4
b4, a4 = signal.iirfilter(N, wn, btype='lowpass', analog=False,
ftype='butter')
wz4, hz4 = signal.freqz(b4, a4, 4000)
plt.plot(wz4, (np.abs(hz4)), label=r'$|H(z)|, N=4$')
plt.legend()
```

```
N = 6
b6, a6 = signal.iirfilter(N, wn, btype='lowpass', analog=False,
ftype='butter')
wz6, hz6 = signal.freqz(b6, a6, 4000)
plt.plot(wz6, (np.abs(hz6)), label=r'$|H(z)|, N=6$')
plt.legend()
```

```
N = 8
b8, a8 = signal.iirfilter(N, wn, btype='lowpass', analog=False,
ftype='butter')
wz8, hz8 = signal.freqz(b8, a8, 4000)
plt.plot(wz8, (np.abs(hz8)), label=r'$|H(z)|, N=8$')
plt.legend()
```

```
N = 10
b10, a10 = signal.iirfilter(N, wn, btype='lowpass', analog=False,
ftype='butter')
wz10, hz10 = signal.freqz(b10, a10, 4000)
plt.plot(wz10, (np.abs(hz10)), label=r'$|H(z)|, N=10$')
plt.legend()
plt.show()
```

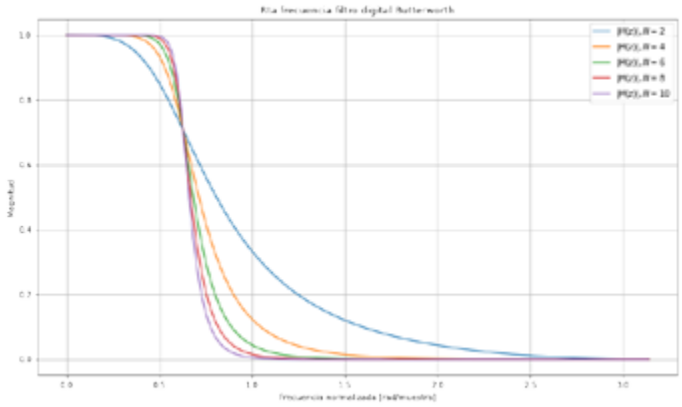


Figura 62. Respuesta en frecuencia filtro Butterworth digital pasa pasa-bajo, $\omega_N = 0.2$ y $N = 2,4,6,8,10$. Estrategia de diseño #2.

Función de transferencia del filtro digital, para $N = 2$:

A partir de los vectores b y a se encuentran las constantes de los polinomios del numerador y denominador de $H(z)$, respectivamente. Específicamente para $N = 2$, se utiliza `b2` y `a2`.

```
b2
```

```
array([0.06745527, 0.13491055, 0.06745527])
```

```
a2
```

```
array([ 1. , -1.1429805, 0.4128016])
```

Y entonces,

$$H(z) \cong \frac{0.0674 + 0.1349z^{-1} + 0.0674z^{-2}}{1 - 1.1429z^{-1} + 0.4128z^{-2}}$$

Nota 1: por simplicidad se expresó $H(z)$ solamente con cuatro cifras decimales.

Nota 2: se puede comparar este valor de $H(z)$ con el obtenido en el ejemplo #1 de esta sección.

Cálculo y gráfica de los polos y ceros del filtro digital, para N=2,4,6,8,10:

```
z2, p2, k2 = signal.tf2zpk(b2, a2)
z4, p4, k4 = signal.tf2zpk(b4, a4)
z6, p6, k6 = signal.tf2zpk(b6, a6)
z8, p8, k8 = signal.tf2zpk(b8, a8)
z10, p10, k10 = signal.tf2zpk(b10, a10)
```

Se utiliza el mismo código que del ejemplo # 1 de esta sección para graficar los polos y ceros de los filtros digitales. Las gráficas se presentan en la Figura 63.

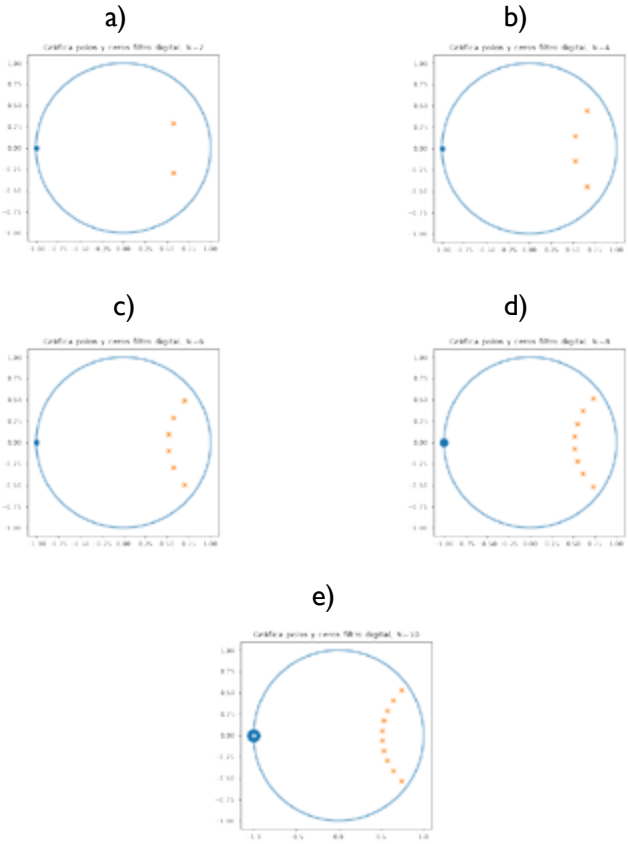


Figura 63. Gráfica de polos y ceros del filtro pasa pasa-bajo, $\omega_N = 0.2$ y N = 2,4,6,8,10. Estrategia de diseño # 2.

Al comparar la Figura 63 con la Figura 61, se aprecia que la ubicación de los polos y ceros es muy similar, por lo que las dos estrategias de diseño de esta sección permiten llegar al "mismo resultado".

5.5. FILTRADO DE SEÑALES CON FILTROS IIR

Para finalizar esta sección de filtros IIR, vamos a filtrar una señal con un filtro IIR diseñado con el método de Transformada Bilineal, a partir de un filtro análogo.

Para ello, utilizaremos la siguiente señal:

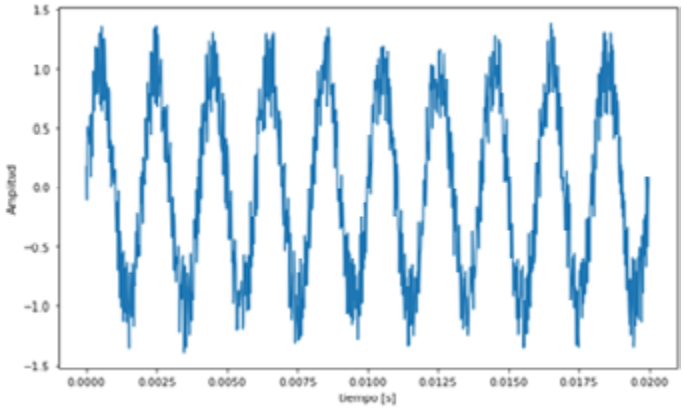


Figura 64. Señal en el dominio del tiempo, xnoise[n].

Esta señal se ha generado con el siguiente código en Python,

```
#Paso 1:  importar librerías de trabajo
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
from scipy import signal
import math

#Paso 2:  generar una señal sin ruido
f = 500 # Hz
fs = 100 * f
step = 1/fs
frad = f * 2 * math.pi
t = np.arange(0,10/f,step)
x = np.sin(frad*t)

#Paso 3:  generar ruido aleatorio
samples = len(x)
An= 0.8
noise = An*np.random.rand(samples) - An/2

# Paso 4:  sumar la señal senoidal con la señal de ruido
xnoise = x +  noise
plt.plot(t,xnoise)
plt.xlabel('tiempo [s]')
plt.ylabel('Amplitud')
```

Visualizando $xnoise[n]$, ésta contiene dos señales, una correspondiente a $x[n]$, y otra a $noise[n]$. Específicamente, $x[n]$ es una señal senoidal; mientras que, $noise[n]$, es un ruido de fondo. Sin embargo, para poder tener información más puntual del comportamiento en frecuencia tanto de $x[n]$ como de $noise[n]$, es necesario realizar un análisis espectral de la señal $xnoise[n]$.

Entonces, utilizamos el código en Python que vimos en el Capítulo 1, para el cálculo y gráfica de la Transformada de Fourier de la señal.

```
import scipy.fftpack as  fourier
L=len(xnoise)
transformada = fourier.fft(xnoise)
magnitud = abs(transformada)
magnitud_lateral = magnitud[0:L//2]
fase = np.angle(transformada)
frecuencias = fs*np.arange(0, L//2)/L
plt.plot(frecuencias, magnitud_lateral)
plt.xlabel('Frecuencia (Hz)', fontsize='10')
plt.ylabel('|FFT|', fontsize='10')
plt.show()
```

Obteniendo el siguiente espectro de $xnoise[n]$

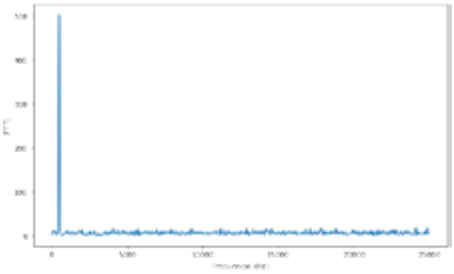


Figura 65. Espectro de $xnoise[n]$.

Observamos que existe un tono (señal de frecuencia pura) correspondiente a la señal senoidal, y que el ruido se encuentra en todos los valores de frecuencia, hasta $f_s/2$ (es decir $25K$ $[Hz]$).

Para determinar la frecuencia exacta correspondiente a la señal senoidal, vamos a apoyarnos en el siguiente código en Python:

```
np.max(magnitud_lateral)
```

```
496.67405522769
```

```
x = np.where(abs(magnitud_lateral) == np.max(magnitud_lateral))
f_tono = np.min(x)*(fs/2)/len(magnitud_lateral)
print(f_tono)
500.0
```

De tal forma que, el tono se encuentra ubicado en los 500 $[Hz]$, de amplitud 496.67.

Teniendo en cuenta que queremos filtrar el ruido que abarca todas las frecuencias, y que la señal de interés se encuentra únicamente en la frecuencia de 500 [Hz], lo

más conveniente en este caso es diseñar un filtro pasa-banda de banda angosta, con $Q > 0.5$ (ej. $Q = 1$), y $\Omega_r = 500\ [Hz]$.

Entonces, la función de transferencia del filtro análogo queda de la siguiente forma:

$$H(s) = \frac{1 * \frac{500 * 2 * \pi}{1}}{s^2 + \left(\frac{500 * 2 * \pi}{1}\right) s + (500 * 2 * \pi)^2}$$

A partir de H(s) se escribe el siguiente código en Python:

```
import numpy as np
from scipy import signal
import matplotlib.pyplot as plt
f = 500
frad = 2*3.14*f
Q = 1
G = 1
nums=np.array([0, G*frad/Q, 0])
dens=np.array([1, frad/Q, frad*frad])

ws, hs = signal.freqs(nums, dens)
plt.plot(ws, (np.abs(hs)), label=r'$|H(s)|$')
plt.legend()
plt.xlabel('Frecuencia [rad/seg]')
plt.ylabel('Magnitud')
plt.title('Respuesta en frecuencia del filtro análogo')
plt.grid()
```

Y obtenemos la siguiente respuesta en frecuencia del filtro análogo,

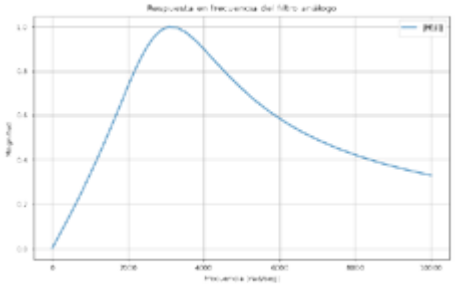


Figura 66. Respuesta en frecuencia filtro análogo pasa-banda, $\Omega_r = 1000\pi\ [rad/seg]$.

Y aplicamos Transformada Bilineal, para obtener $H(z)$, así:

```
filtz = signal.dlti(*signal.bilinear(nums, dens, fs))
wz, hz = signal.freqz(filtz.num, filtz.den)
plt.plot(wz, (np.abs(hz)), label=r'$|H(z)|$')
plt.legend()
plt.xlabel('Frecuencia normalizada [rad/muestra]')
plt.ylabel('Magnitud')
plt.title('Respuesta en frecuencia del filtro digital')
plt.grid()
```

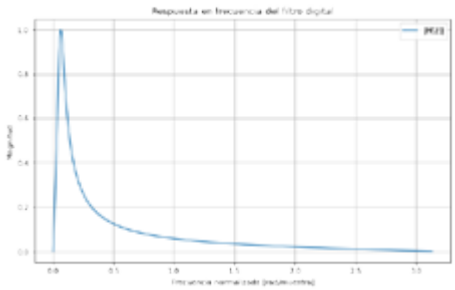


Figura 67. Respuesta en frecuencia filtro digital pasa-banda, $\omega_d = 0.061$ $[rad/muestra]$.

La frecuencia de corte del filtro digital la obtenemos tanto a partir de la gráfica de la respuesta en frecuencia del filtro, como de la ecuación teórica que relaciona la frecuencia del filtro análogo con la frecuencia del filtro digital.

Utilizamos el siguiente código en Python:

```
np.max(abs(hz))
```

```
0.9958317963050908
```

```
x = np.where(abs(hz) == np.max(abs(hz)))
wcd = np.max(x)*3.14/len(wz)
print(wcd)
```

```
0.061328125000000004
```

```
wd =2*np.arctan(frad/(fs*2))
wd
```

```
0.06277937277186546
```

Verificamos que los valores son similares, entonces el filtro quedó bien diseñado.

Posteriormente, encontramos las constantes de los polinomios del numerador y denominador de $H(z)$, así:

```
filtz.num
```

```
array([ 0.01544233, 0. , -0.01544233])
```

```
filtz.den
```

```
array([ 1. , -1.96523623, 0.96911534])
```

Y escribimos $H(z)$ de la siguiente manera,

$$H(z) = \frac{0.015 - 0.015z^{-2}}{1 - 1.965z^{-1} + 0.969z^{-2}}$$

Nota: por simplicidad se han utilizado solamente tres cifras decimales en $H(z)$.

A partir de $H(z)$ filtramos la señal con `signal.filtfilt`, con el siguiente código:

```
filtrada = signal.filtfilt(filtz.num, filtz.den, xnoise)
plt.rcParams["figure.figsize"] = (10,6)
plt.plot(t,filtrada)
```

Nota: a diferencia del caso de filtros FIR, ahora el parámetro "a" de la instrucción `signal.filtfilt` no es una constante de valor igual a uno, sino que también es un vector. Específicamente, con el nombre de las variables que hemos utilizado, corresponde a `filtz.den`.

Obteniendo como resultado:

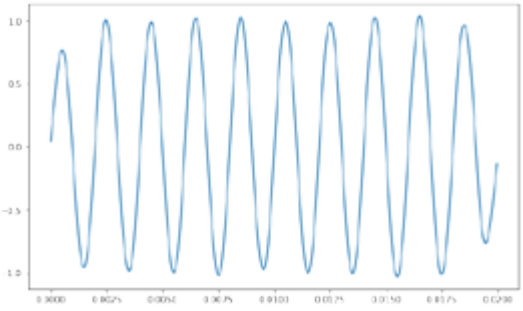


Figura 68. Señal filtrada en el dominio del tiempo.

Adicionalmente, podemos verificar que el espectro de la señal filtrada no contiene el ruido de fondo, con el siguiente código en Python:

```
transformada2 = fourier.fft(filtrada)
magnitud2 = abs(transformada2)
magnitud_lateral2 = magnitud2[0:L//2]
fase2 = np.angle(transformada2)
frecuencias2 = fs*np.arange(0, L//2)/L
plt.rcParams["figure.figsize"] = (10,6)
plt.plot(frecuencias2, magnitud_lateral2)
plt.xlabel('Frecuencia (Hz)', fontsize='10')
plt.ylabel('|FFT|', fontsize='10')
plt.show()
```

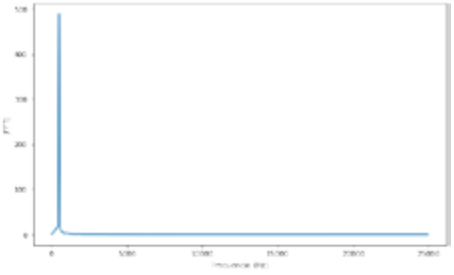


Figura 69. Espectro de la señal filtrada.

CAPÍTULO 6.
Procesamiento de Imágenes

Como capítulo final de este libro, trabajaremos con señales en dos dimensiones, específicamente con imágenes. Esto te permitirá obtener las bases conceptuales y de programación para abordar cursos más avanzados en procesamiento de imagen, por ejemplo, de visión por computador.

Al finalizar el capítulo, deberás estar en capacidad de:

1. Explicar las diferencias entre imágenes blanco-negro, escala de grises, e imágenes a color.
2. Explicar en qué consisten los modelos de color RGB y HSV, así como realizar conversiones utilizando la librería de OpenCV de Python.
3. Realizar ecualización de imagen utilizando la librería de OpenCV de Python.
4. Distinguir diferentes tipos de ruido en imágenes.
5. Reconocer qué tipo de filtro es adecuado para reducir un tipo de ruido específico en la imagen.
6. Explicar el concepto de convolución en imágenes.
7. Realizar detección de bordes a partir de diferentes tipos de kernels.
8. Explicar las diferencias entre DCT y DFT en imágenes.
9. Aplicar la DCT o la DFT en imágenes utilizando la librería de OpenCV de Python.
10. Explicar el concepto de compresión de imágenes.

6.1. CONCEPTOS BÁSICOS DE IMÁGENES

En las primeras secciones del libro hemos trabajado con señales uni-dimensionales, y gran parte de los ejemplos se han enfocado en audio. En este capítulo, nos enfocaremos en imágenes, que corresponden a señales en 2D, cuyos ejes corresponden a filas y columnas.

Lo primero que debemos saber es que no todas las imágenes tienen las mismas características. Por ejemplo, pueden variar entre ellas el tamaño y el color utilizado.

En relación con el tamaño, la unidad de medida es el píxel, y la resolución de la imagen está dada por la cantidad de filas y columnas. Entonces, una imagen de 100 x 100 tendrá 10,000 píxeles de resolución, mientras que, una imagen de 1,000 x 1,000 tendrá 1M píxeles de resolución. En las cámaras digitales actuales es típico encontrar resoluciones de varios mega píxeles. Entonces, hemos identificado la primera diferencia entre las señales 1D correspondientes a audio y las imágenes, en el primer caso hablábamos de muestras de la señal, y ahora hablaremos de píxeles de la imagen.

La segunda característica de la imagen corresponde a su color. Podemos encontrar imágenes a blanco-negro (BW: *black and white*), a escala de grises y a color de tres bandas (aunque también existen imágenes con mayor número de canales, las cuales no abordaremos en este libro).

Las primeras, BW, tienen solamente un bit asociado a cada píxel de la imagen, de tal forma que, si la imagen tiene 1M píxeles, entonces tendrá 1M bits. El valor de "1" corresponde al blanco, mientras que, el valor de "0" corresponde al negro. Las segundas, imágenes a escala de grises, tienen 8 bits por cada píxel, y el rango de color va del negro ("00000000") al blanco ("11111111") pasando por distintas tonalidades de gris, para un total de 256 colores. Entonces, una imagen de 1M píxeles tendrá 8Mbits, o de forma equivalente 1MB. Finalmente, tenemos las imágenes a color, que típicamente se denominan RGB (Red, Green, Blue), aunque realmente ese es uno de los espacios de color que existen. En este caso, por cada píxel de la imagen tenemos 8 bits asociados a cada uno de los tres canales de color, para un total de 24 bits por píxel. Retomando el mismo ejemplo, para la imagen de 1M píxeles, tendremos 3MB.

Para ilustrar de mejor forma las diferencias en términos de color de las imágenes BW, a escala de grises y a color, se presenta en la Figura 70 una imagen del repositorio personal de los autores del libro.

Figura 70. Ejemplo de imagen: a) BW, b) Escala de grises, c) Color. Fuente: repositorio personal de los autores.

6.2. ESPACIOS DE COLOR

El espacio de color más ampliamente conocido se denomina RGB, donde la imagen se representa en tres canales o "bandas" de color, una correspondiente al rojo, otra al verde, y la última al azul. Cada color tiene 256 tonalidades distintas (2^8), y en total se tiene una paleta de16,776,216 colores (es decir, 2^{24}). Retomando el ejemplo de la sección anterior, se presentan las tres bandas de color en la Figura 71.

(a) (b) (c)

Figura 71. Ejemplo de imagen RGB: a) banda R, b) banda G, c) banda B. Fuente: repositorio personal de los autores.

(a) (b) (c)

Figura 72. Ejemplo de imagen HSB: a) banda H, b) banda S, c) banda B. Fuente: repositorio personal de los autores.

Otro espacio de color corresponde a HSV (*Hue, Saturation,Value*) o también conocido como HSB (*Hue, Saturation, Brightness*). En este espacio de color, la primera banda corresponde al tono de la imagen, la segunda a la saturación de la imagen, y la tercera al *brillo de la imagen*. Para nuestra foto de la playa, las tres bandas se presentan en la Figura 72. En este espacio de color, la banda de brillo (Figura 72b) es muy similar a la imagen a escala de grises (que presentamos en la Figura 70b).

6.3. INTRODUCCIÓN A LA LIBRERÍA OPENCV

Bueno, en este punto te preguntarás como se puede leer la imagen en lenguaje Python, convertir una imagen a color en una imagen a escala de grises y/o BW, así como transformar una imagen de un espacio a color a otro. Para ello, vamos a utilizar la librería OpenCV de Python, la cual es especializada en procesamiento de imágenes[4].

4 https://opencv24-python-tutorials.readthedocs.io/en/latest/py_tutorials/py_tutorials.html

Figura 73. Logo de OpenCV.

Entonces, manos a la obra con el código en Python.

Paso 1: importar la librería de OpenCV, leer la imagen que previamente hemos subido a nuestro entorno de trabajo en Colaboratory, y conocer el tamaño de la imagen.

```
import cv2
img = cv2.imread("/content/Fig74.jpg")
img.shape
```

Para la imagen de prueba, el resultado es:

```
(300, 400, 3)
```

Paso 2: visualización de la imagen. Para ello se debe importar un patch en Colaboratory.

```
from google.colab.patches import cv2_imshow
cv2_imshow(img)
```

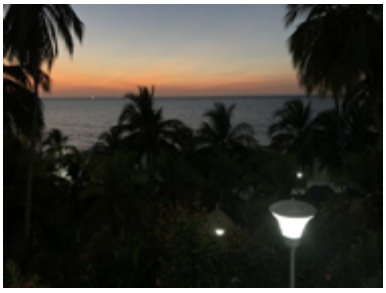


Figura 74. Imagen a color – foto playa.

Nota: si trabajas en Jupyter Notebook no es necesario que importes el patch, y puedes utilizar cv2.imshow.

Paso 3: conversión de imagen RGB a escala de grises

```
img_gray=cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
cv2_imshow(img_gray)
```

Figura 75. Imagen a escala de grises – foto playa.

Paso 4: conversión de imagen a escala de grises en BW

```
r,img_bw = cv2.threshold(img_gray,45,255,cv2.THRESH_BINARY)
cv2_imshow(img_bw)
```

Figura 76. Imagen a blanco y negro – foto playa.

Lo que hemos realizado en este paso 4 se conoce como *umbralización* de la imagen (o *thresholding*, en inglés), proceso en el cual a los píxeles que superan el umbral se les asigna el color blanco, y a los que no superan el umbral se les asigna el color negro. Si modificamos el valor del umbral, la imagen va a lucir más clara (umbral bajo) o más oscura (umbral alto). Podemos apreciar que las palmeras tienen el color negro, mientras que el mar y el cielo el color blanco, dado que, en la imagen a escala de grises la tonalidad de gris tanto del cielo como del mar es mucho más clara que la de las palmeras.

La instrucción `cv2.threshold`[5] requiere de dos valores numéricos, el primero corresponde al umbral, y el segundo al valor que se asigna en caso de que el píxel supere el umbral. En el ejemplo, el umbral es 45 y el valor asignado a los píxeles que superen el umbral es 255.

5 https://opencv24-python-tutorials.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_thresholding/py_thresholding.html

Paso 5: guardar las imágenes en tu entorno de trabajo

```
cv2.imwrite('image_color.jpg',img)
cv2.imwrite('image_gray.jpg',img_gray)
cv2.imwrite('image_bw.jpg',img_bw)
```

Paso 6: conversión de RGB a HSV

```
H, S, V = cv2.split(cv2.cvtColor(img, cv2.COLOR_RGB2HSV))
cv2_imshow(H)
```

Figura 77. Imagen canal H – foto playa.

```
cv2_imshow(S)
```

Figura 78. Imagen canal S – foto playa.

```
cv2_imshow(V)
```

Figura 79. Imagen canal V – foto playa.

Hasta este punto, ya sabemos cómo leer imágenes con la librería OpenCV, convertir una imagen a color en una imagen a escala de grises y BW, y convertir del espacio de color RGB a HSV. Puedes ampliar la información de conversión de espacios de color en la documentación de OpenCV de `cv2.cvtColor`[6].

6.4. ECUALIZACIÓN DE IMÁGENES

¿Alguna vez te ha pasado que tomas una foto con poca luz y la imagen te quedó muy oscura? ¿Sabes cómo funcionan los ajustes de brillo en los celulares, por ejemplo, para aclarar fotos oscuras? ¡Eso lo aprenderás en esta sección y adiós a borrar fotos porque quedaron muy oscuras!

Lo primero que debemos conocer es el concepto de histograma de una imagen y como calcularlo y graficarlo en lenguaje Python. Pues bueno, la definición general del histograma es que es una representación gráfica de la ocurrencia de los datos. En el caso de imágenes, el histograma muestra cuántos píxeles de la imagen tienen color 0, cuántos tienen color 1, y así sucesivamente hasta cuantos píxeles tienen color 255 (en imágenes a escala de grises). En el caso de imágenes a color, presentará la cantidad de píxeles para cada uno de los 256 niveles de color por banda, es decir, es necesario dibujar tres histogramas, uno para la banda R, otro para la banda G, y otro para la banda B. Si la imagen es BW, entonces el histograma solamente tendrá dos niveles de color, el 0 correspondiente al negro, y el 1 correspondiente al blanco.

A continuación, se presentan los pasos.

Paso 1: Lectura de la imagen a color

```
import cv2
img = cv2.imread("/content/Fig80.jpg")
```

Y obtengo esta hermosa imagen. Si, ya sé que está un poco oscura, pero más adelante aprenderemos a aclararla.

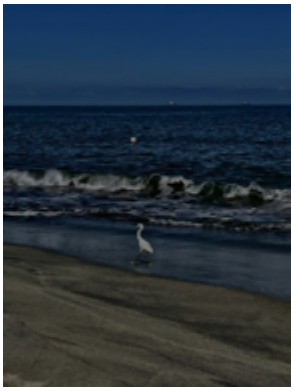


Figura 80. Imagen a color – foto mar. Fuente: repositorio personal de los autores.

6 https://opencv24-python-tutorials.readthedocs.io/en/stable/py_tutorials/py_imgproc/py_colorspaces/py_colorspaces.html

Por ahora, vamos a conocer el tamaño de la imagen con `img.shape`. El resultado es (4032, 3024, 3. Es decir que, nuestra imagen tiene 4032 filas, 3024 columnas y 3 bandas de color (por defecto en el espacio BGR). El total de píxeles de la imagen es 4032 x 3024, que es igual a 12,192,768. En términos de bytes, el total se calcula así: 4032 x 3024 x 3, que es igual a 36,578,304, dado que en cada banda de color un píxel tiene 1B, y la imagen tiene tres bandas de color. Ahora, te preguntarás si ese tamaño que acabamos de encontrar es el mismo que te aparece en tu PC en relación con esa imagen, y si revisas te darás cuenta de que solo pesa 1,54 KB. La diferencia entre el cálculo que acabamos de realizar y el peso real de la imagen radica en su tipo de formato, (en este caso es *.jpg), el cual es un formato de compresión de imágenes que reduce su peso, pero conserva su resolución espacial. Si la imagen estuviese en formato bmp de 24 bits, el espacio en disco sería el calculado previamente (alrededor de 36 MB).

Para facilitar la visualización de la imagen en Colaboratory, vamos a realizar un proceso de redimensionamiento de la imagen, para que quede de tamaño 400 filas y 300 columnas, para ello utilizaremos el siguiente código:

Paso 2: Redimensionamiento de la imagen

```
img=cv2.resize(img, (300, 400), interpolation = cv2.INTER_AREA)
```

Ten en cuenta que primero incluimos la cantidad de columnas que deseamos que la imagen tenga, y después la cantidad de filas. Entonces, la cantidad de píxeles por banda es ahora de 400 x 300, que es igual a 120,000. El siguiente paso, es convertir la imagen a escala de grises.

Paso 3: Conversión imagen a color en escala de grises

```
img_gray=cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
cv2_imshow(img_gray)
```

Obteniendo esta imagen:

Figura 81. Imagen a escala de grises – foto mar.

Bueno, ahora sí, vamos a dibujar nuestro histograma y a mejorar la apariencia de la imagen.

Paso 4: Histograma de la imagen a escala de grises

```
import matplotlib.pyplot as plt
pixels=img_gray.shape[0]*img_gray.shape[1]
print('la cantidad de píxeles de la imagen es:', pixels)
hist = cv2.calcHist([img_gray],[0],None,[256],[0,256])
plt.plot(hist)
plt.show()
```

Obteniendo el siguiente resultado:

la cantidad de píxeles de la imagen es: 120000

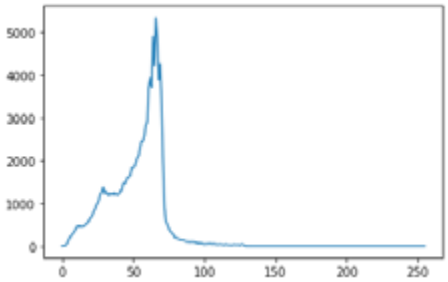


Figura 82. Histograma de la imagen a escala de grises – foto mar.

A partir del histograma se identifica que la imagen está altamente concentrada en intensidades de píxel alrededor de 60 (en escala 0 a 255), y que existen muy pocos píxeles con intensidades superiores a 128 (mitad de escala). Esto es coherente con la "apariencia oscura" de la imagen.

A continuación, mejoraremos la apariencia de la imagen a escala de grises.

Paso 5: Ecualización del histograma de la imagen a escala de grises

```
img_gray_eq = cv2.equalizeHist(img_gray)
cv2_imshow(img_gray_eq)
```

Como resultamos, obtenemos:

Figura 83. Imagen ecualizada a escala de grises – foto mar.

Si comparas esta imagen con la imagen a escala de grises original (*Paso 3*), notarás una gran diferencia. Es más clara. ¿Cómo crees entonces que es el histograma de la imagen ecualizada?

Paso 6: Histograma de la imagen a escala de grises ecualizada

```
hist2 = cv2.calcHist([img_gray_eq],[0],None,[256],[0,256])
plt.plot(hist2)
```

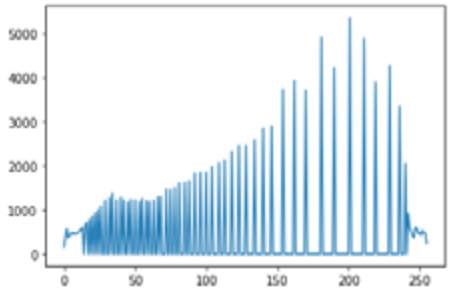


Figura 84. Histograma de la imagen ecualizada a escala de grises – foto mar.

Este histograma es significativamente diferente al obtenido en el *Paso 4*. Ahora, una gran parte de los píxeles de la imagen tienen niveles de color mayores a 128, y, por lo tanto, la imagen tiene una apariencia clara. Por otro lado, es típico en los histogramas ecualizados que se tengan numerosos picos de ocurrencia, y que no se tengan curvas suavizadas como en los histogramas de imágenes naturales (sin ecualizar).

A continuación, dibujaremos el histograma por banda de color y ecualizaremos la imagen a color.

Paso 7: Histograma de la imagen a color (histograma por cada banda de color)

```
img_RGB=cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
color = ('r','g','b')
for i,col in enumerate(color):
    histr = cv2.calcHist([img_RGB],[i],None,[256],[0,256])
    plt.plot(histr,color = col)
    plt.xlim([0,256])
plt.show()
```

En este punto es pertinente explicar que cuando leemos imágenes con OpenCV, las bandas de color quedan en orden contrario al del espacio RGB. Es decir, primero la banda B (azul), después la banda G (verde), y finalmente, la banda R (roja). Por ello, se hace necesario convertir de BGR a RGB, y posteriormente dibujar el histograma de cada una de las bandas (se puede realizar en gráficas independientes, o en la misma gráfica como con este código).

El histograma que obtenemos es:

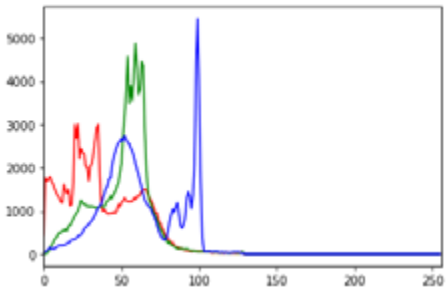


Figura 85. Histograma por banda de la imagen a color – foto mar.

Cada histograma se ha dibujado con el color correspondiente a su banda. El histograma de la banda roja tiene la mayor parte de sus píxeles por debajo del color 50. El histograma de la banda verde tiene la mayor parte de sus píxeles con color cercano a 50. Mientras que, el histograma de la banda azul tiene dos zonas de color que sobresalen, alrededor de 50 y alrededor de 100, esta última con mayor cantidad de píxeles. Aunque los histogramas son diferentes entre sí, tienen en común que en los tres casos la cantidad de píxeles por encima del color 128 es prácticamente cero.

Paso 8: Ecualización del histograma de la imagen a color

```
H, S, V = cv2.split(cv2.cvtColor(img, cv2.COLOR_RGB2HSV))
V_equ = cv2.equalizeHist(V)
img_equ = cv2.cvtColor(cv2.merge([H, S, V_equ]), cv2.COLOR_HSV2RGB)
cv2_imshow(img_equ)
```

El proceso de ecualización de la imagen lo realizaremos en la banda V del espacio de color HSV. Para lo cual, primero convertiremos la imagen de RGB a HSV, posteriormente realizaremos un split en las tres bandas, para así ecualizar únicamente la banda V. Finalmente, volvemos a unir las tres bandas del espacio HSV (con la banda V ecualizada), y convertimos de HSV a RGB. La imagen a color ecualizada es:

Figura 86. Imagen ecualizada a color – foto mar.

Mejoró con relación a la imagen del Paso 1, ¿cierto? Bueno, ya has aprendido un concepto de procesamiento de imágenes que tiene una aplicación práctica. Cuando vuelvas a cambiar el brillo de una imagen, recuerda que lo que estás haciendo es un proceso de ecualización de su histograma.

Espero que te haya gustado esta temática. Si quieres ampliar la información de histogramas en OpenCV, te invito a consultar la documentación de la librería[7].

6.5. RUIDO EN IMÁGENES

En esta sección aprenderemos a reconocer tres tipos diferentes de ruido presentes en imágenes: guassiano (gaussian), uniforme (uniform), y sal y pimienta (salt and pepper).

6.5.1. Ruido gaussiano:

Este ruido se caracteriza porque su distribución (histograma) tiene la forma de una campana de gauss, en la que existe un valor central (con gran parte de los píxeles del ruido), y pocos píxeles en los colores extremos. La forma y comportamiento está definida por el promedio y la varianza. Si la varianza es baja, la campana de gauss es angosta; mientras que, si la varianza es alta, la campana de gauss es ancha. El promedio es el valor central de la campana.

Vamos ahora a generar este tipo de ruido para adicionarlo a una imagen a color y visualizar su efecto. Para ello utilizaremos el siguiente código en Python:

```
import cv2
import numpy as np
from google.colab.patches import cv2_imshow
img = cv2.imread("/content/Fig89.jpg")
noise = np.zeros((img.shape[0], img.shape[1]),dtype=np.uint8)
gaussian_noise = np.zeros((img.shape[0], img.shape[1], img.sha-
pe[2]),dtype=np.uint8)
gaussian_noise[:,:,0]=cv2.randn(noise, 128, 30)
gaussian_noise[:,:,1]=cv2.randn(noise, 128, 30)
gaussian_noise[:,:,2]=cv2.randn(noise, 128, 30)
cv2_imshow(gaussian_noise)
```

Lo primero que hacemos es crear una matriz de ceros del mismo tamaño de la imagen a la cual le adicionaremos el ruido. Posteriormente, con la instrucción `cv2.randn`[8] vamos a generar ruido gaussiano. Debemos seleccionar el valor central de la distribución gaussiana (μ), y la desviación estándar (σ); para nuestro caso $\mu = 128$, y $\sigma = 30$. Este ruido gaussiano lo creamos para cada una de las bandas a color (banda 0, banda 1 y banda 2, de gaussian_noise). El resultado se presenta a continuación:

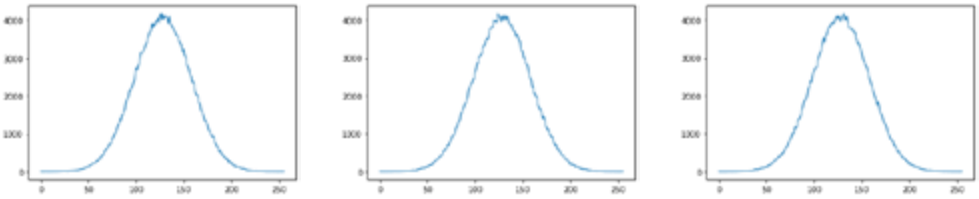
7 https://opencv24-python-tutorials.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_histograms/py_table_of_contents_histograms/py_table_of_contents_histograms.html#table-of-content-histograms

8 https://docs.opencv.org/4.5.3/d2/de8/group__core__array.html#gaeff1f61e972d133a04ce3a5f81cf6808

Figura 87. Imagen a color – ruido gaussiano.

Para verificar que el ruido obtenido es de tipo gaussiano, utilizamos el siguiente código:

```
import matplotlib.pyplot as plt
# repetir este paso por canal
hist = cv2.calcHist([gaussian_noise],[0],None,[256],[0,256])
plt.plot(hist)
plt.show()
```



(a) (b) (c)

Figura 88. Histograma por banda de la imagen a color – ruido gaussiano.

Se verifica que los histogramas de la Figura 88 de cada uno de los canales, efectivamente tienen forma de campana de gauss.

Como siguiente paso, leemos una imagen en Colaboratory[9]:

```
img = cv2.imread("/content/Fig89.jpg")
img.shape
img=cv2.resize(img, (640, 480), interpolation = cv2.INTER_AREA)
cv2_imshow(img)
```

Figura 89. Imagen a color – villa de leyva. Fuente: repositorio personal de los autores.

9 Esta imagen hace parte del repositorio personal del autor de este libro

Y adicionamos el ruido que previamente hemos creado, así:

```
noisy_img_gn = cv2.add(img, (gaussian_noise*0.5).astype(np.uint8))
cv2_imshow(noisy_img_gn)
```

El ruido gaussiano se multiplica por 0.5 para no saturar a la imagen, y se convierte en formato entero de 8 bits con `astype(np.uint8)`. Posteriormente, se adiciona a la imagen a color con la instrucción `cv2.add`, obteniendo el siguiente resultado:

Figura 90. Imagen a color con ruido gaussiano – villa de leyva.

¿Cuál es el efecto de este tipo de ruido en la imagen?

Rta: La foto luce "envejecida".

6.5.2. Ruido uniforme:

Otro ruido típico en imágenes es el ruido uniforme. A diferencia del ruido anterior, este tiene una distribución *uniforme* de sus colores, es decir que no existe un color central, sino que todos los colores (o tonos) tienen la misma cantidad de píxeles (o aproximadamente la misma cantidad).

El procedimiento para crear este tipo de ruido es similar al caso anterior. Debemos crear una matriz de ceros del mismo tamaño de la imagen, y posteriormente para cada una de las bandas de color creamos el ruido. Sólo que en este caso utilizamos la instrucción `cv2.randu`, en lugar de `cv2.randn`. Podemos utilizar el siguiente código en Python:

```
noise = np.zeros((img.shape[0], img.shape[1]),dtype=np.uint8)
uniform_noise = np.zeros((img.shape[0], img.shape[1], img.sha-
pe[2]),dtype=np.uint8)
uniform_noise[:,:,0]=cv2.randu(noise, 0, 256)
uniform_noise[:,:,1]=cv2.randu(noise, 0, 256)
uniform_noise[:,:,2]=cv2.randu(noise, 0, 256)
cv2_imshow(uniform_noise)
```

Obteniendo el siguiente resultado:

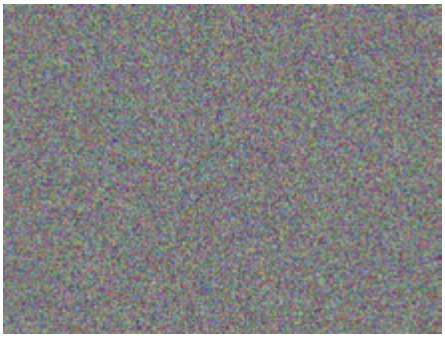
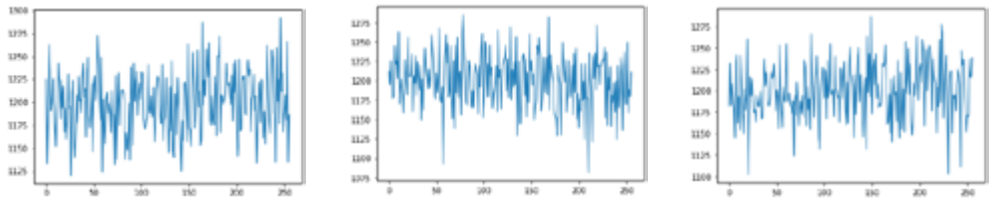


Figura 91. Imagen a color – ruido uniforme.

Como siguiente paso dibujamos el histograma, banda a banda, así:

```
# repetir este paso por canal
hist = cv2.calcHist([uniform_noise],[0],None,[256],[0,256])
plt.plot(hist)
plt.show()
```

Obteniendo:



(a) (b) (c)

Figura 92. Histograma por banda de la imagen a color – ruido uniforme.

Aunque el histograma que obtenemos no tiene una *perfecta distribución uniforme*, la cantidad de píxeles para cada uno de los tonos (0 a 255) es muy similar entre sí. Para el tamaño de imagen que hemos utilizado en este ejemplo, la cantidad de píxeles por canal es 480 x 640 = 307.200. Esta cantidad de píxeles dividido en los 256 tonos es igual a 1.200. Si se revisa con detalle los histogramas, precisamente las ocurrencias oscilan alrededor de ese valor.

Finalmente, adicionamos el ruido a la imagen a color, con el siguiente código en Python:

```
noisy_img_un = cv2.add(img, (uniform_noise*0.5).astype(np.uint8))
cv2_imshow(noisy_img_un)
```

Cuyo resultado es:

Figura 93. Imagen a color con ruido uniforme – villa de leyva.

¿En qué se diferencia esta imagen de la obtenida al adicionar el ruido gaussiano?

Rta: aparte de envejecida, tienen pequeños "gránulos" o "puntos de arena" en toda la imagen. El efecto es notorio en zonas amplias y de pocos detalles, como el cielo o las nubes.

6.5.3. Ruido sal y pimienta:

Este tipo de ruido tiene dos tonos, uno correspondiente a la sal y el otro a la pimienta. Para crear ruido de este tipo, lo primero que debemos hacer es crear ruido uniforme para cada una de las bandas de color, y posteriormente aplicar un proceso de umbralización (similar al que utilizamos cuando convertimos una imagen a escala de grises en una imagen BW). Dependiendo del valor del umbral seleccionado, tendremos más o menos píxeles correspondientes a sal y a pimienta.

Para el siguiente código en Python el umbral seleccionado es 10, y a los píxeles que superen el umbral se les asigna el color 255 (máxima escala).

```
sp_noise=np.zeros((img.shape[0],  img.shape[1],  img.shape[2]),d-
type=np.uint8)
ret,impulse_noise0=cv2.threshold(uniform_noise[:,:,0],10,255,cv2.
THRESH_BINARY)
ret,impulse_noise1=cv2.threshold(uniform_noise[:,:,1],10,255,cv2.
THRESH_BINARY)
ret,impulse_noise2=cv2.threshold(uniform_noise[:,:,2],10,255,cv2.
THRESH_BINARY)
sp_noise[:,:,0]=impulse_noise0
sp_noise[:,:,1]=impulse_noise1
sp_noise[:,:,2]=impulse_noise2
cv2_imshow(sp_noise[:,:,0])
```

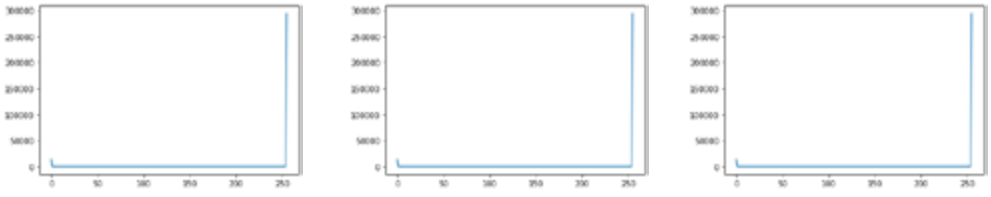
¡En este caso obtendremos poca pimienta y mucha sal!

Figura 94. Imagen a color con ruido sal y pimienta, con $th = 10$.

Y sus histogramas por banda, son:

```
# repetir este paso por canal
hist = cv2.calcHist([sp_noise],[2],None,[256],[0,256])
plt.plot(hist)
plt.show()
```

Obteniendo:



(a) (b) (c)

Figura 95. Histograma por banda de la imagen a color – ruido sal y pimienta con $th = 10$.

Ahora, vamos a fijar un umbral alto, por ejemplo, de 200:

```
sp_noise=np.zeros((img.shape[0],  img.shape[1],  img.shape[2]),d-
type=np.uint8)
ret,impulse_noise0=cv2.threshold(uniform_noise[:,:,0],200,255,-
cv2.THRESH_BINARY)
ret,impulse_noise1=cv2.threshold(uniform_noise[:,:,1],200,255,-
cv2.THRESH_BINARY)
ret,impulse_noise2=cv2.threshold(uniform_noise[:,:,2],200,255,-
cv2.THRESH_BINARY)
sp_noise[:,:,0]=impulse_noise0
sp_noise[:,:,1]=impulse_noise1
sp_noise[:,:,2]=impulse_noise2
cv2_imshow(sp_noise[:,:,0])
```

¡En este caso obtendremos poca sal y mucha pimienta!

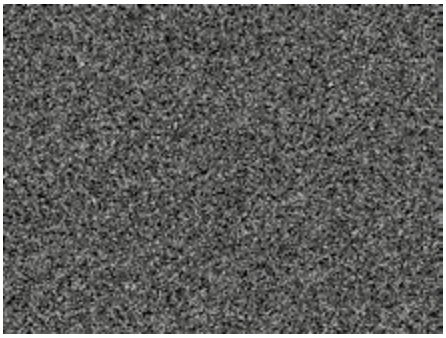
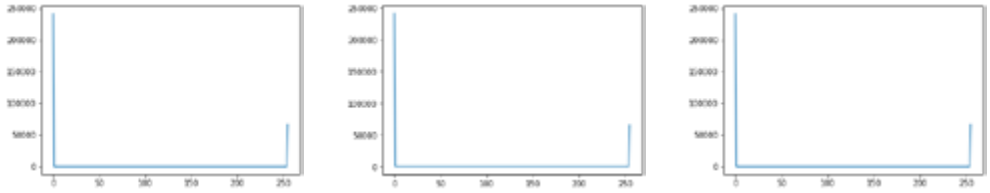


Figura 96. Imagen a color con ruido sal y pimienta, con $th = 200$.

Y sus histogramas, son:



(a) (b) (c)

Figura 97. Histograma por banda de la imagen a color – ruido sal y pimienta con $th = 200$.

Y finalmente adicionamos este ruido a la imagen, así:

```
noisy_img_sp = cv2.add(img, (sp_noise*0.5).astype(np.uint8))
cv2_imshow(noisy_img_sp)
```

Figura 98. Imagen a color con ruido sal y pimienta, $th = 200$ – villa de leyva.

¿En qué se diferencia esta imagen de la obtenida al adicionar el ruido uniforme?

Rta: es mucho más notorio el efecto granular que en la imagen con ruido uniforme.

6.6. FILTROS ESPACIALES

En esta sección entenderemos y aplicaremos el concepto de filtro espacial. Matemáticamente lo abordaremos en la Sección 6.7, pero por ahora, de forma conceptual y práctica realizaremos el filtrado de ruido en imágenes.

Lo primero que vamos a realizar es comparar el efecto que tiene en una imagen los tres diferentes tipos de ruido que se explicaron en la Sección 6.5. La Figura 99 presenta un ejemplo.

(a) (b) (c)

Figura 99. Imagen de playa con tres tipos distintos de ruido: (a) sal y pimienta, (b) guassiano, (c) uniforme. Fuente: repositorio personal de los autores.

Pero, ¿cómo las diferenciamos?

- Empecemos con la imagen central, el efecto que tiene es de "envejecida", entonces esa imagen contiene ruido gaussiano.
- Ahora, comparemos las imágenes de los extremos, ambas tienen un efecto "granular". La imagen de la izquierda tiene ruido sal y pimienta, por ser más evidente el efecto granular; mientras que, la de la derecha tiene ruido uniforme.

A continuación, por medio de ejemplos se ilustrará el efecto que tienen diferentes filtros espaciales en imágenes con diferentes tipos de ruido.

Empezaremos con la imagen que tienen ruido sal y pimienta, a la cual le aplicamos un filtro tipo promedio. Este filtro 2D es similar al filtro de promedio 1D que conocimos al inicio de este libro, pero en este caso es una matriz con todos sus valores iguales a uno dividido en su tamaño (igual a filas x columnas). Por ejemplo, si el tamaño del filtro es (5×5), entonces cada posición del filtro tendrá el peso de $1/25$, como se presenta en la siguiente Figura.

$$\frac{1}{25}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Figura 100. Filtro de promedio (5×5).

Una vez hemos leído la imagen en Python, aplicamos el filtro con el siguiente código:

```
f1_sp = cv2.blur(noisy_img_sp, (5,5), 0)
cv2_imshow(f1_sp)
```

Este filtro lo aplicamos a la imagen denominada *noisy_img_sp*, cuyo resultado es la *imagen f1_sp*, la cual se presenta a continuación.

Figura 101. Imagen filtrada con filtro de promedio – ruido sal y pimienta.

El segundo tipo de filtro que vamos a evaluar es el filtro gausisano. En este caso, los valores de la matriz varían entre sí, teniendo mayor peso la posición central del filtro, y de menor peso las posiciones de los extremos. En este tipo de filtro los pesos decrecen de forma gaussiana a medida que se alejan de la posición central, como se presenta en la siguiente figura:

$$\frac{1}{273}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

Figura 102. Filtro gaussiano (5×5). Se ha encerrado en un recuadro rojo la posición central del filtro.

Para el filtro gausisano se utiliza el siguiente código en Python:

```
f2_sp = cv2.GaussianBlur(noisy_img_sp, (5,5), 0)
cv2_imshow(f2_sp)
```

Cuyo tamaño del filtro es también (5×5), y la salida en este caso se denomina *f2_sp*. La imagen filtrada se presenta a continuación:

Figura 103. Imagen filtrada con filtro de gaussiano – ruido sal y pimienta.

Finalmente, filtraremos la imagen con un filtro de mediana (no confundir con el filtro de promedio). Este tipo de filtro difiere a los dos anteriores en que no existe una matriz de pesos del filtro. Se realiza un proceso de ordenamiento de los valores de los píxeles de la imagen de una región de igual tamaño al del filtro, y se selecciona el valor correspondiente a la posición central de los píxeles ordenados.

El código en Python es:

```
f3_sp = cv2.medianBlur(noisy_img_sp, 5)
cv2_imshow(f3_sp)
```

La imagen filtrada corresponde a *f3_sp,* como se presenta en la siguiente figura.

Figura 104. Imagen filtrada con filtro de mediana – ruido sal y pimienta.

La forma en que cada uno de estos filtros opera sobre la imagen, se explicará en detalle en la Sección 6.7.

Por ahora, quiero que respondas la siguiente pregunta.

¿Cuál imagen filtrada consideras que presenta mejor calidad?, es decir, ¿qué filtro seleccionarías para eliminar ruido tipo sal y pimienta?
Rta: Para este tipo de ruido, el filtro de mediana es la mejor opción.

En la segunda parte de esta sección, buscaremos un filtro para una imagen que contiene ruido tipo gaussiano. Partiremos con el filtro de promedio (Figura 72), y seguiremos con otro tipo de filtro denominado filtro bilateral (Figura 73).

Figura 105. Imagen filtrada con filtro de promedio – ruido gaussiano.

El filtro bilateral tiene en cuenta tres parámetros para calcular el valor de salida: diámetro de la vecindad (d), varianza a nivel de color (σ_{color}), y varianza a nivel de ubicación espacial (σ_{space}).

- d es el diámetro de cada vecindad de píxeles. Si es negativo, se calcula a partir de σ_{space}.
- Cuando σ_{color} es alto, entonces, los colores más alejados dentro de la vecindad se mezclan, obteniendo largas áreas de color casi-homogéneo.
- Cuando σ_{space} es alto, entonces, los píxeles más alejados entre sí se mezclan (espacialmente hablando).

Este tipo de filtro es similar al filtro gaussiano, en términos de la cercanía en ubicación espacial, pero incluye el concepto de cercanía de color también.

El siguiente es el código en Python para el filtro bilateral con $d = 9$, $\sigma_{color} = 10$, $\sigma_{space} = 10$

```
blur1 = cv2.bilateralFilter(noisy_img_sp,15,50,100)
cv2_imshow(blur1)
```

Figura 106. Imagen filtrada con filtro bilateral – ruido gaussiano.

¿Cuál imagen filtrada consideras que presenta mejor calidad?, es decir, ¿Qué filtro seleccionarías para eliminar ruido tipo gaussinoa?
Rta: Para este tipo de ruido, el filtro bilateral es la mejor opción.

Puedes complementar la información de los filtros espaciales de esta sección en https://docs.opencv.org/4.x/d4/d13/tutorial_py_filtering.html

6.7. CONVOLUCIÓN

En esta sección comprenderemos el concepto de "*convolución*" en imágenes. Estrictamente hablando, realmente realizaremos una operación de *correlación* entre un filtro y una imagen, pero, teniendo en cuenta que, en la comunidad académica de visión por computador y de aprendizaje profundo el nombre utilizado para esa operación es el de *convolución*, utilizaremos ese nombre en este libro.

Lo primero a tener en cuenta es que la *convolución* es una operación que requiere dos matrices, una de las cuales es la imagen, y la otra es el filtro. Típicamente, los filtros tienen la misma cantidad de filas que de columnas, por ejemplo, de 3 x 3, pero se podrían diseñar filtros con dimensiones que no sean iguales entre sí. Cada una de las posiciones del filtro se denominan "pesos". Conceptualmente, el filtro debe tener una dimensión menor a la de la imagen para poder realizar un proceso de "barrido" sobre ella.

Con un ejemplo ilustraremos el proceso:

Figura 107. Imagen y filtro para operación de convolución.

El primero paso consiste en adicionarle un borde a la imagen con valores de ceros, ampliando su dimensión en 2 filas y dos columnas. Es decir, para nuestra Imagen de ejemplo, la cual es de (5 x 5), al incluirle el borde quedará de (7 x 7).

0	0	0	0	0	0	0
0	10	10	30	30	10	0
0	10	10	30	30	10	0
0	10	10	30	30	10	0
0	10	10	30	30	10	0
0	10	10	30	30	10	0
0	0	0	0	0	0	0

Figura 108. Imagen de entrada con borde.

El propósito de adicionarle el borde a la Imagen de entrada es que el resultado de la convolución (Imagen filtrada) contenga la misma cantidad de filas y de columnas que de la Imagen de entrada. Cuando el tamaño del filtro es de 3 x 3, el borde es de 2 filas (una superior y una inferior) y dos columnas (una a la izquierda y una a la derecha); cuando el filtro es de tamaño 5x5, el borde es de 4 filas (dos superiores y dos inferiores) y 4 columnas (dos a la derecha y dos a la izquierda), y así sucesivamente.

Como segundo paso, el filtro se superpone sobre la Imagen de entrada, ubicándolo en el extremo superior izquierdo. Posteriormente, se realiza la multiplicación de los píxeles de la Imagen con los pesos del filtro. Si el filtro es de tamaño 3 x 3, entonces se realizan 9 multiplicaciones. Finalmente, se suma el resultado de las multiplicaciones, y el valor obtenido se asigna al primer pixel de la imagen (primera fila, primera columna). Hay que tener en cuenta que, si el resultado de la operación anterior es negativo, se escribe un cero en el pixel de salida correspondiente. Por otro lado, si el resultado es superior a 255, se escribe 255.

El proceso se presenta a continuación:

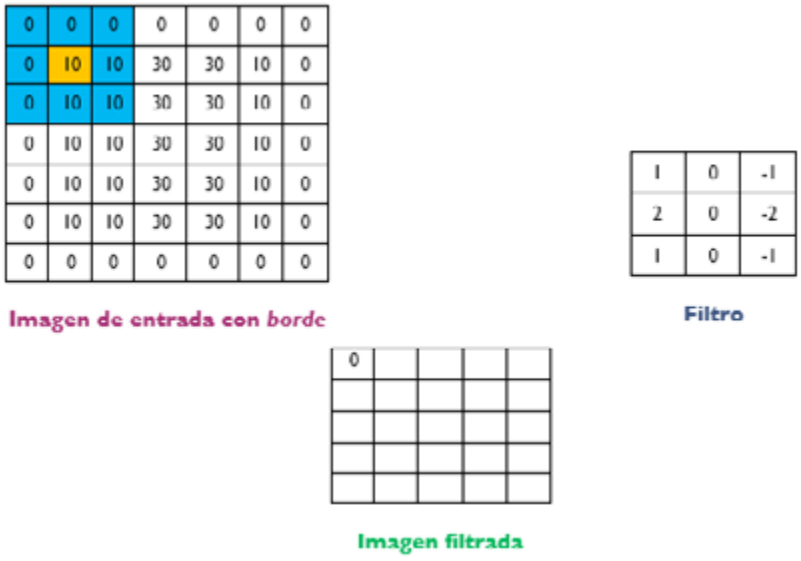


Figura 109. Proceso de convolución: Paso 2. Se sombrea en amarillo el píxel central de la imagen, para el paso correspondiente.

Como tercer paso, el filtro se desplaza una posición a la derecha, y se repite de nuevo el proceso de realizar las multiplicaciones, sumar su resultado y asignar al pixel correspondiente de la imagen de salida (primera fila, segunda columna). El proceso se presenta en la siguiente figura.

Figura 110. Proceso de convolución: paso 3. Se sombrea en amarillo el píxel central de la imagen, para el paso correspondiente.

Como cuarto paso, y así sucesivamente, se desplaza de nuevo el filtro una posición a la derecha, se realizan las correspondientes multiplicaciones, se suman sus valores y se asigna al píxel de la imagen de salida que corresponda. Una vez el filtro se desplaza y llega al borde de la imagen, debe desplazarse de nuevo, empezando por la segunda fila de la imagen, primera columna. El proceso de desplazamiento se realiza de forma iterativa, hasta que se recorra por completo a la imagen de entrada. La ubicación del píxel central para cada uno de los pasos del proceso de convolución y la dirección del desplazamiento se presentan a continuación:

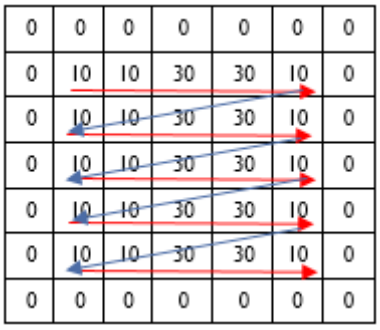


Figura 111. Pixel central en el proceso de convolución: barrido de la imagen de izquierda a derecha, y de arriba abajo.

Para el presente ejemplo, el resultado de la convolución es:

0	0	0	60	90
0	0	0	80	120
0	0	0	80	120
0	0	0	80	120
0	0	0	60	90

Figura 112. Imagen filtrada.

Para saber cuál es el tamaño del borde a adicionarle a la imagen, utilizaremos las siguientes ecuaciones:

$$W_0 = W_1 - W_k + 1 \qquad \text{Ecuación 50}$$

$$H_0 = H_1 - H_k + 1 \qquad \text{Ecuación 51}$$

Donde W_1, W_K, W_0, corresponden a la cantidad de columnas de la imagen de entrada con borde, del filtro y de la imagen filtrada (output), respectivamente. Mientras que, H_1, H_k, H_0, corresponden a la cantidad de filas de la imagen de entrada con borde, del filtro y de la imagen filtrada, respectivamente.

Entonces, si queremos que la imagen de salida tenga 5 x 5 y estamos utilizando un filtro de 3 x 3, necesitamos que la imagen de entrada con borde sea de 7 x 7, teniendo en cuenta que al reemplazar los valores anteriores en la ecuación 50 o ecuación 51, tenemos que $5 = 7 - 3 + 1$. A la imagen de entrada sin borde (cuyas dimensiones son iguales a la de la imagen de salida), debemos adicionarle 2 filas y 2 columnas, o, en otras palabras, un borde de 1 rodeando a la imagen.

6.8. DETECCIÓN DE BORDES

En esta subsección abordaremos el tema de detección de bordes en imágenes. Lo primero, es saber que, así como existen filtros cuyo propósito consiste en reducir el ruido de una imagen (como los vistos en el Capítulo 6.6.), también tenemos filtros cuyo objetivo es detectar el borde de una imagen. Mientras los primeros cumplen que la sumatoria de sus *pesos* es igual a 1, en los segundos (detección de bordes) se cumple que la sumatoria de sus *pesos* es igual a 0.

Adicionalmente, se pueden detectar bordes en una sola dirección o multi-dirección. Dentro de los filtros más conocidos en la literatura tenemos *Prewitt, Sobel y Laplaciano*. Y como algoritmo de detección de bordes (que incluye etapa de pre-procesamiento, filtrado y pos-procesamiento), tenemos el algoritmo Canny[10].

Empecemos con el filtro *Prewitt*. Es una clase de detector de bordes aplicando la diferencia entre píxeles de primer orden. Puede detectar bordes en el eje horizon-

10 Öztürk, Ş., & Akdemir, B. (2015). Comparison of edge detection algorithms for texture analysis on glass production. Procedia-Social and Behavioral Sciences, 195, 2675-2682.

tal o en el eje vertical. Este filtro utiliza un tamaño de 3x3, donde la fila o columna central son de valor 0, y las filas o columnas de los extremos son de valor 1 y -1. A continuación, se presenta el filtro *Prewitt* para cada dirección de detección de borde.

Figura 113. Filtro Prewitt (3 x 3).

En el caso del filtro *Sobel*, también se detectan bordes en la dirección vertical y horizontal, pero en este caso, se realiza un énfasis en el pixel central de las filas o columnas cuyos pesos son distintos de cero, realizando una detección más fuerte de los cambios de la imagen utilizando la primera derivada. En la siguiente figura se presenta el filtro *Sobel*.

Figura 114. Filtro Sobel (3 x 3).

Por otro lado, el filtro *Laplaciano* se basa en la segunda derivada de la imagen (o diferencia de segundo orden)[11]. Existen dos versiones del filtro *Laplaciano*, en la primera, se computa la diferencia entre el pixel central y el promedio de sus vecinos directos (arriba, abajo, izquierda, derecha), y en la segunda, se computa la diferencia entre el pixel central y el promedio de todos sus vecinos (incluidas las esquinas). La versión básica y la alternativa se presentan a continuación[12].

Figura 115. Filtro Laplaciano (3 x 3).

Finalmente, tenemos el algoritmo o filtro Canny, el cual realiza varias etapas, las cuales se resumen a continuación[13y14]:

11 https://www.sciencedirect.com/topics/engineering/laplacian-filter.
12 Nixon, M. S., & Aguado, A. S. (2008). Low-level feature extraction (including edge detection). Feature Extraction and Image Processing. 3rd edi. Linacre House/Jordan Hill/Oxford: Elsevier, 115-79.
13 https://docs.opencv.org/4.x/da/d22/tutorial_py_canny.html
14 https://docs.opencv.org/3.4/da/d5c/tutorial_canny_detector.html

a. Reducción de ruido: es una etapa de pre-procesamiento que consiste en reducir el ruido presente en la imagen, por medio de un filtro Gaussiano de tamaño 5 x 5.

b. Identificación del gradiente de intensidad de la imagen: se filtra la imagen obtenida en el paso anterior tanto con un filtro Sobel de detección de bordes horizontales, como de detección de bordes verticales, obteniendo G_x y G_y, respectivamente. A partir de las dos imágenes resultantes (una por cada filtro Sobel), se calcula la imagen gradiente, tanto en magnitud como en fase, aplicando las siguientes ecuaciones:

$$|G| = \sqrt{G_x + G_y} \qquad \textbf{Ecuación 52}$$

$$\measuredangle = \mathbf{tg}^{-1}\left({}^{G_y}/_{G_x} \right) \qquad \textbf{Ecuación 53}$$

La dirección del gradiente siempre es perpendicular a los bordes. Se aproxima a uno de los cuatro posibles ángulos: horizontal, vertical, diagonal derecha, diagonal izquierda.

c. Supresión de los no máximos: esta etapa y la siguiente hacen parte del pos-procesamiento. Consiste en remover los píxeles no deseados, que no correspondan con el borde de la imagen. Si existen varios píxeles vecinos en la dirección del gradiente que son potenciales bordes, se identifica cuál de ellos es un máximo local, y ese es el píxel que se conversa para la siguiente etapa del algoritmo.

d. Umbralización con histéresis: en esta última fase se eliminan falsos bordes, a partir de un proceso de histéresis con dos umbrales. Se define un umbral alto y un umbral bajo. Si el potencial borde supera al umbral alto, entonces se considera un borde real. Si, por el contrario, es menor que el umbral bajo, se descarta. Para los potenciales bordes cuya intensidad se encuentra entre el umbral bajo y el umbral alto, la decisión de incluirse como un verdadero borde o de eliminarse depende de sus píxeles vecinos. Si éstos son bordes, se considera también como borde; en caso contrario, se descarta.

Una de las ventajas del *algoritmo Canny* es que detecta de forma simultánea bordes en cuatro direcciones (vertical, horizontal, diagonal derecha y diagonal izquierda). Adicionalmente, el borde detectado es delgado, gracias a sus etapas de pos-procesamiento posteriores al filtrado (supresión de los no-máximos y umbralización con histéresis).

A continuación, aplicaremos los filtros anteriores a una imagen, para comparar las diferencias de forma visual entre los bordes detectados en cada caso.

```
import cv2
import numpy as np
from google.colab.patches import cv2_imshow
img = cv2.imread('coctel.jpg')
cv2_imshow(img)
```

```
prewitt_x  = np.array([[1, 1, 1],
                       [0,  0,  0],
                       [-1,  -1,  -1]], dtype=np.float32)
print(prewitt_x )

fig1= cv2.filter2D(img, -1, prewitt_x, borderType=0)
cv2_imshow(fig1)

fig1g = cv2.cvtColor(fig1, cv2.COLOR_BGR2GRAY )
ret, fig1bw = cv2.threshold(fig1g,50,255,cv2.THRESH_BINARY)
cv2_imshow(255-fig1bw) # imagen filtrada con Prewitt_x
```

```
prewitt_y  = np.array([[1,  0,  -1],
                       [1,  0,  -1],
                       [1,  0,  -1]], dtype=np.float32)
print(prewitt_y )

fig2= cv2.filter2D(img, -1, prewitt_y, borderType=0)
cv2_imshow(fig2)

fig2g = cv2.cvtColor(fig2, cv2.COLOR_BGR2GRAY )
ret, fig2bw = cv2.threshold(fig2g,50,255,cv2.THRESH_BINARY)
cv2_imshow(255-fig2bw) # imagen filtrada con Prewitt_y
```

```
fig3bw  = fig1bw  + fig2bw
cv2_imshow(255-fig3bw) # imagen filtrada con Prewitt_x + Prewitt_y
```

```
sobel_x  = np.array([[1,  2,  1],
                     [0,  0,  0],
                     [-1,  -2,  -1]], dtype=np.float32)
print(sobel_x )

fig4= cv2.filter2D(img, -1, sobel_x, borderType=0)
cv2_imshow(fig4)

fig4g = cv2.cvtColor(fig4, cv2.COLOR_BGR2GRAY )
ret, fig4bw = cv2.threshold(fig4g,50,255,cv2.THRESH_BINARY) # imagen
filtrada con Sobel_x
cv2_imshow(255-fig4bw)
```

```
sobel_y  = np.array([[1,  0,  -1],
                     [2,  0,  -2],
                     [1,  0,  -1]], dtype=np.float32)
print(sobel_y )

fig5= cv2.filter2D(img, -1, sobel_y, borderType=0)
cv2_imshow(fig5)

fig5g = cv2.cvtColor(fig5, cv2.COLOR_BGR2GRAY )
ret, fig5bw = cv2.threshold(fig5g,50,255,cv2.THRESH_BINARY) # imagen
filtrada con Prewitt_y
cv2_imshow(255-fig5bw)
```

```
fig6bw  = fig4bw  + fig5bw
cv2_imshow(255-fig6bw) # imagen filtrada con Sobel_x + Sobel_y
```

```
laplaciano1 = np.array([[0,  -1,  0],
                        [-1,  4,  -1],
                        [0,  -1,  0]], dtype=np.float32)
print(laplaciano1)

fig7= cv2.filter2D(img, -1, laplaciano1, borderType=0)
cv2_imshow(fig7)

fig7g = cv2.cvtColor(fig7, cv2.COLOR_BGR2GRAY )
ret, fig7bw = cv2.threshold(fig7g,50,255,cv2.THRESH_BINARY)
cv2_imshow(255-fig7bw) # imagen filtrada con Laplaciano básico
```

```
laplaciano2 = np.array([[-1,  -1v -1],
                        [-1,  8,  -1],
                        [-1,  -1,  -1]], dtype=np.float32)
print(laplaciano2)

fig8= cv2.filter2D(img, -1, laplaciano2, borderType=0)
cv2_imshow(fig8)

fig8g = cv2.cvtColor(fig8, cv2.COLOR_BGR2GRAY )
ret, fig8bw = cv2.threshold(fig8g,50,255,cv2.THRESH_BINARY)
cv2_imshow(255-fig8bw) # imagen filtrada con Laplaciano alternativo
```

```
edges_canny = cv2.Canny(img,220,55)
cv2_imshow(255-edges_canny) # imagen filtrada con algoritmo Canny
```

Empezaremos analizando las imágenes filtradas con *Prewitt*. La obtenida con *Prewitt_x* detecta bordes especialmente en la dirección horizontal, como la altura de la bebida dentro de la copa, o el soporte horizontal del techo del restaurante. En el caso de la imagen filtrada con *Prewitt_y,* no se detectan los bordes mencionados anteriormente, pero sí los bordes correspondientes a las columnas verticales de soporte del techo. Finalmente, la imagen obtenida al sumar las dos anteriores es más completa que sus antecesoras por separado, mostrando bordes en ambas direcciones.

En el caso de las imágenes obtenidas con Sobel, los resultados son similares a las obtenidas con *Prewitt*. Sin embargo, se puede apreciar mayor demarcación en algunos bordes.

Por otro lado, las imágenes obtenidas con el filtro *Laplaciano* (en sus dos versiones) muestran el borde vertical de la copa, aunque es más notorio con el Laplaciano alternativo. En ambos casos, las imágenes filtradas tienen bordes delgados, a diferencia de sus antecesoras.

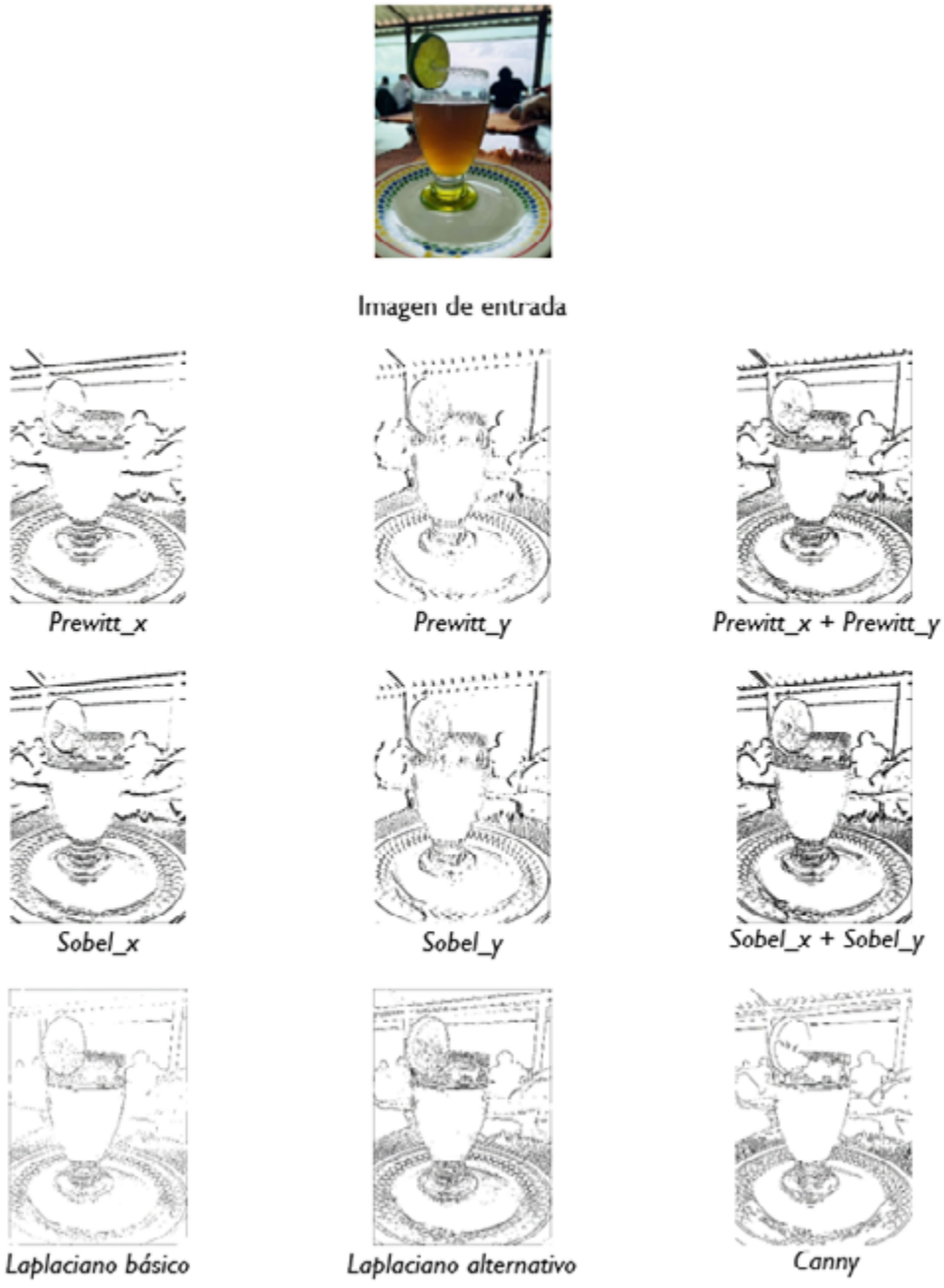


Figura 116. Imagen de entrada y detección de bordes con diferentes tipos de filtros.
Fuente: repositorio personal de los autores.

Finalmente, con el algoritmo Canny se tienen bordes delgados en todas las direcciones, y aparecen bordes en zonas de la imagen que con los otros filtros no se visualizaban, por ejemplo, las ondulaciones en el tejado.

6.9. TRANSFORMADA DFT Y DCT

En esta sección se abordan dos transformadas en imágenes, del dominio espacial al dominio frecuencial. Específicamente, las correspondientes con la Transformada de Fourier Discreta (DFT) y la Transformada Consenoidal Discreta (DCT).

6.9.1. DFT (Discrete Fourier Transform)

La DFT de una imagen se calcula a partir de la siguiente ecuación:

$$F(k,l) = \sum_{a=0}^{M-1} \sum_{b=0}^{N-1} f(a,b) e^{-i2\pi\left(\frac{ka}{M}+\frac{lb}{N}\right)} \qquad \textbf{Ecuación 54}$$

Teniendo en cuenta que,

$$e^{ix} = \{cos(x)\} + \{i * sin(x)\} \qquad \textbf{Ecuación 55}$$

Donde $F(k,l)$ es la Transformada Discreta de Fourier, mientras que *f(a,b)* es la imagen en el dominio espacial de tamaño (M,N). Es decir, el resultado de la DFT se obtiene al multiplicar la imagen en el dominio espacial *f(a,b)* por la función base (que en este caso es una señal exponencial compleja) y sumar el resultado para cada pareja (k,l). Se resalta que tanto los valores *(a,b)* como los valores (k,l) son enteros.

Cuando se grafica la DFT de una imagen, no se puede relacionar fácilmente el resultado obtenido con la imagen original. Típicamente, si existen cambios significativos de dirección en la imagen, éstos se verán reflejados en la DFT (patrones de líneas blancas). Si la imagen se invierte en el eje vertical (*flip* vertical), el efecto que se tiene en su DFT es precisamente el de inversión. De forma similar, si la imagen si invierte respecto al aje horizontal (*flip* horizontal), también se tendrá el efecto en su DFT de inversión. En ambos casos, la inversión en la DFT es en relación con el eje vertical, de tal forma que la DFT de la imagen invertida horizontal es igual a la DFT de la imagen invertida vertical. Por otro lado, si a la imagen se le aplica doble inversión (una por cada eje), su DFT es igual al de la imagen original (sin invertir).

La Figura 84 presenta un ejemplo de una imagen y su correspondiente DFT para diferentes tipos de manipulaciones de la imagen. Se resalta que la DFT de la imagen original es igual a la DFT de doble *flip*; mientras que, la DFT de *flip* vertical es igual a la DFT de *flip* horizontal.

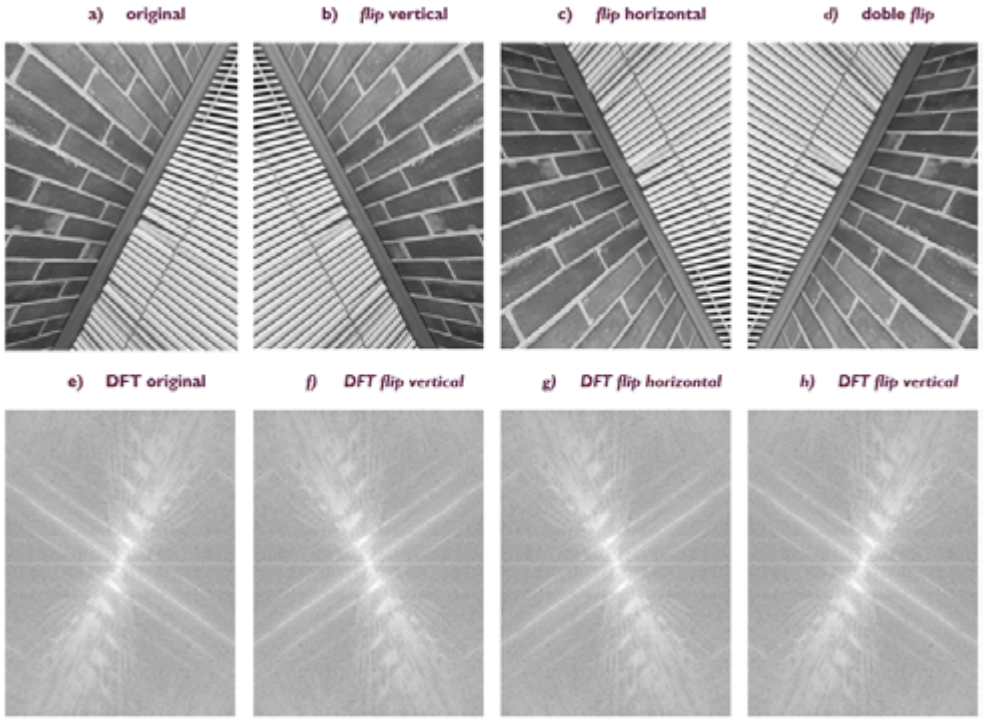


Figura 117. Imagen con su respectiva DFT.

Para calcular la DFT de una imagen en Python, utilizamos el siguiente código:

Paso 1) Cargue de librerías de lectura de la imagen

```
import numpy as np
import cv2
from google.colab.patches import cv2_imshow
url= "/content/oficina.png"
img = cv2.imread(url)
```

Paso 2) Convertir la imagen RGB a escala de grises y representarla en punto flotante de 32 bits

```
img_gray=cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
cv2_imshow(img_gray)
img_float32 = np.float32(img_gray)
```

Paso 3) Calcular la DFT y visualizar el resultado en escala logarítmica

```
dft = cv2.dft(img_float32, flags = cv2.DFT_COMPLEX_OUTPUT)
dft_shift = np.fft.fftshift(dft)
magnitude_spectrum = 20*np.log(cv2.magnitude(dft_shift[:,:,0],dft_
shift[:,:,1]))
cv2_imshow(magnitude_spectrum)
```

Paso 4) Invertir la imagen en el eje vertical, calcular su DFT y graficar

```
flipVertical = cv2.flip(img_float32, 1)
cv2_imshow(flipVertical)
dft = cv2.dft(flipVertical, flags = cv2.DFT_COMPLEX_OUTPUT)
dft_shift = np.fft.fftshift(dft)
magnitude_spectrum = 20*np.log(cv2.magnitude(dft_shift[:,:,0],dft_
shift[:,:,1]))
cv2_imshow(magnitude_spectrum)
```

Paso 5) Invertir la imagen en el eje horizontal, calcular su DFT y graficar

```
flipHorizontal = cv2.flip(img_float32, 0)
cv2_imshow(flipHorizontal)
dft = cv2.dft(flipHorizontal, flags = cv2.DFT_COMPLEX_OUTPUT)
dft_shift = np.fft.fftshift(dft)
magnitude_spectrum = 20*np.log(cv2.magnitude(dft_shift[:,:,0],dft_
shift[:,:,1]))
cv2_imshow(magnitude_spectrum)
```

Paso 6) Doble inversión de la imagen (horizontal y vertical), calcular su DFT y graficar

flipBoth = cv2.flip(img_float32, -1)

```
cv2_imshow(flipBoth)
dft = cv2.dft(flipBoth, flags = cv2.DFT_COMPLEX_OUTPUT)
dft_shift = np.fft.fftshift(dft)
magnitude_spectrum = 20*np.log(cv2.magnitude(dft_shift[:,:,0],dft_
shift[:,:,1]))
cv2_imshow(magnitude_spectrum)
```

6.9.2. DCT (Discrete Cosine Transform)

Esta transformada es muy útil para la compresión de imágenes, dado que gran parte de la información de la imagen (la más significativa o representativa) se concentra en pocos coeficientes espectrales. Hace parte del algoritmo de compresión de imágenes conocido como JPEG (*Joint Photographic Experts Group*).

A diferencia de la DFT, en este caso todos sus coeficientes son reales, calculados a partir de la siguiente ecuación:

$$C(k,l) = \sum_{a=0}^{M-1} \sum_{b=0}^{N-1} f(a,b) \cos\left(\frac{\pi}{M}\left(a+\frac{1}{2}\right)k\right) \cos\left(\frac{\pi}{N}\left(b+\frac{1}{2}\right)l\right) \qquad \textbf{Ecuación 56}$$

Donde $C(k,l)$ corresponde a la DCT de la imagen $f(a,b)$ de tamaño (M,N).

Típicamente, la DCT se calcula por bloques de la imagen, es decir, la imagen se divide en zonas y a cada zona se la aplica la DCT. A continuación, se presenta un ejemplo de la DCT para la imagen completa, y para diferentes tamaños de bloque.

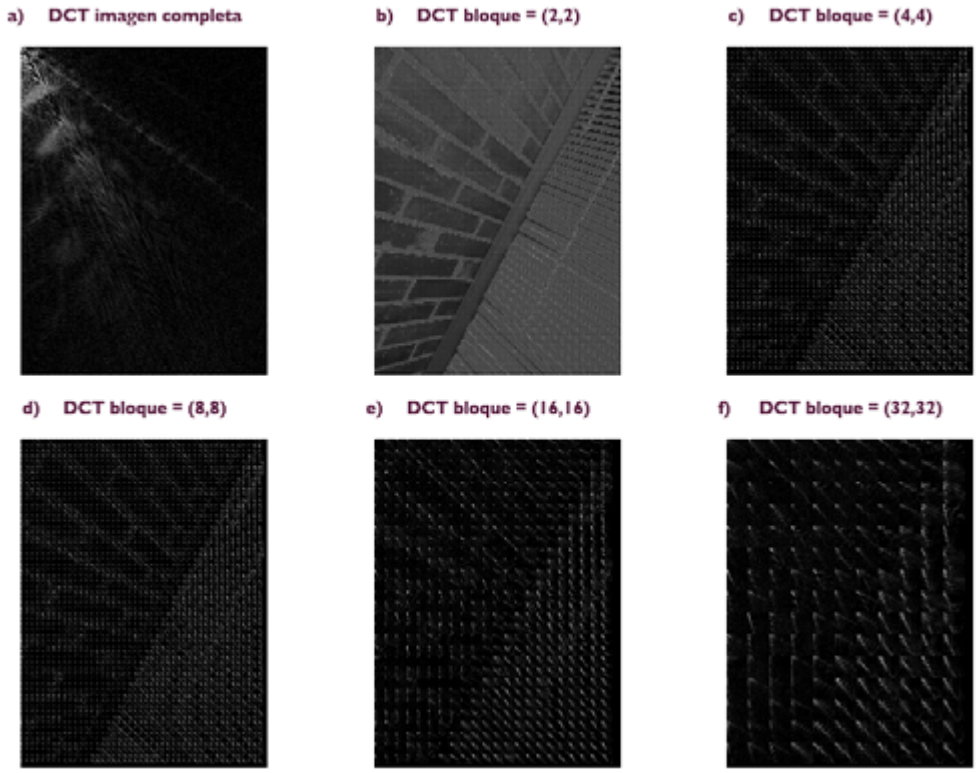


Figura 118. DCT de la imagen de la Figura 84.a.

A diferencia de la DFT, sí es posible encontrar una relación directa entre la DCT y la imagen de entrada, cuando el tamaño del bloque es pequeño. Por ejemplo, en la Figura 118b, se alcanza a apreciar la pared y la persiana de la oficina; mientras que, en la Figura 118c y Figura 118d, se visualizan líneas diagonales correspondientes a la separación entre filas de ladrillos. Cuando el tamaño del bloque es de (32,32) o superior, ya no se alcanzan a identificar los patrones de la imagen.

En este caso, el código de Python para obtener las gráficas anteriores, se presenta a continuación:

Paso 1) Cargue de librerías de lectura de la imagen

```
import numpy as np
import cv2
from google.colab.patches import cv2_imshow
url= "/content/oficina.png"
img = cv2.imread(url)
```

Paso 2) Convertir la imagen RGB a escala de grises y representarla en punto flotante de 32 bits

```
img_gray=cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
cv2_imshow(img_gray)
img_float32 = np.float32(img_gray)
```

Paso 3) Calcular la DCT y visualizar el resultado

```
dct = cv2.dct(img_float32)
cv2_imshow(dct)
```

Nota: en este caso no se necesitan re-ordenar los coeficientes, como sí se realizó en el caso de la DFT. Adicionalmente, no se calcula la magnitud, dado que los valores son reales. Tampoco, se grafica en escala logarítmica.

Paso 4) Definir el tamaño del block, calcular la cantidad de bloques y crear un DCT de salida de valores cero.

```
B=2 #blocksize
img1 = img_float32
h= img1.shape[0]
w =img1.shape[1]
blocksV=np.int(h/B)
blocksH=np.int(w/B)
transformed=np.zeros([h, w])
```

Nota: este ejemplo está diseñado para bloques cuadrados. En este caso es de (2,2).

Paso 4) Aplicar la DCT por bloque y escribir el resultado en la zona de salida correspondiente.

```
for row in range(blocksV):
    for col in range(blocksH):
      currentblock = cv2.dct(img1[row*B:(row+1)*B,col*B:(col+1)*B])
        transformed[row*B:(row+1)*B,col*B:(col+1)*B]= currentblock
cv2_imshow((transformed))
```

6.9.3. Comprensión de imágenes con la DCT

Como se había mencionado previamente, una de las aplicaciones de la DCT es en la comprensión de imágenes, específicamente en el estándar JPEG. A continuación, se explicará brevemente en que consiste ese método de comprensión.

Lo primero a resaltar es que JPEG es un método de comprensión con pérdida de información (o *lossy*), que significa que parte de los datos se pierden en el proceso de compresión y no se puede recuperar la imagen exactamente igual a la original; no obstante, de forma visual, no se apreciarán diferencias significativas entre la imagen original y su versión comprimida. Su principal ventaja sobre métodos de compresión sin pérdida de información (o *lossless*) es que permite obtener una tasa de compre-

sión mayor, conocida como CR (*compression rate*), la cual corresponde a la relación entre el tamaño de la imagen sin comprimir y el tamaño de la imagen comprimida.

Los principales bloques que hacen parte del método JPEG son: DCT, cuantización inteligente, y codificación RL y *Huffman*. De forma muy resumida, los pasos son los siguientes[15]:

a. Aplicar DCT por bloques de la imagen, por ejemplo, de tamaño (8,8). El resultado es otra imagen del mismo tamaño, cuyos datos corresponden a coeficientes espectrales.

b. Aplicar cuantización a los coeficientes espectrales, dividiendo su valor entre un factor de cuantización. De esta manera, se reduce la cantidad de valores de salida (y la precisión de los datos). Adicionalmente, el proceso es inteligente, dado que el factor de cuantización no es constante, sino que, depende de la amplitud del coeficiente a cuantizar. A los coeficientes que representan frecuencias mayores se les aplica un factor de cuantización mayor.

c. A los coeficientes cuantizados se les aplica el método de codificación run-length (RL). Este método aprovecha la gran cantidad de ceros consecutivos que se obtienen al combinar la DCT con la cuantización inteligente. El barrido sobre los coeficientes cuantizados se realiza en forma de zig-zag, empezando en el extremo superior izquierdo de la matriz (DCT cuantizada). La longitud de la trama de salida es mucho menor a la cantidad de coeficientes cuantizados del paso b.

d. Finalmente, se aplica codificación Huffman. La idea principal de este método es representar los "símbolos" de mayor ocurrencia de la trama con la menor cantidad de bits, mientras que, los de menor ocurrencia con la mayor cantidad de bits. Entonces, los coeficientes espectrales cuantizados y codificados con RL tendrán una representación binaria que es significativamente menor a multiplicar el tamaño de la imagen por 8 bits (en el caso de imágenes a escala de grises) o por 24 bits (en el caso de imágenes a color de 3 canales). Los valores de compresión pueden llegar a 100 veces.

15 https://cs.stanford.edu/people/eroberts/courses/soco/projects/data-compression/lossy/jpeg/index.htm