

PROCESAMIENTO **DIGITAL** DE **VOZ E IMÁGENES** EN LA ERA DE LA **INTELIGENCIA ARTIFICIAL**

DORA MARÍA BALLESTEROS

UNIVERSIDAD MILITAR NUEVA GRANADA,
FACULTAD DE INGENIERÍA.

INGENIERÍA EN TELECOMUNICACIONES





IA

INVESTIGACIÓN EDUCATIVA & PEDAGÓGICA IBEROAMERICANA

editorial redipe

editorial redipe © 2026

Título original:

Libro de investigación.

PROCESAMIENTO DIGITAL DE VOZ E IMÁGENES EN LA ERA DE LA INTELIGENCIA ARTIFICIAL

Autor: DORA MARÍA BALLESTEROS

ISBN: 978-1-957395-68-5
Primera edición: AGOSTO DE 2026

SELLO Editorial
Editorial REDIPE (95857440), Nueva York – Cali
Red de Pedagogía S.A.S. NIT: 900460139-2
© de la ilustración de la cubierta

Comité Editorial:

Valdir Heitor Barzotto, Universidad de Sao Paulo, Brasil
Carlos Arboleda A. PhD Investigador Southern Connecticut State University, Estados Unidos
Agustín de La Herrán Gascón, Ph D. Universidad Autónoma de Madrid, España
Mario Germán Gil Claros, Grupo de Investigación Redipe
Rodrigo Ruay Garcés, Chile. Coordinador Macroproyecto Investigativo Iberoamericano EvaluaciónEducativa
Julio César Arboleda, Ph D. Dirección General Redipe. Grupo de investigación Educación y Desarrollo humano, Universidad de San Buenaventura

Queda prohibida, salvo excepción prevista en la ley, la reproducción (electrónica, química, mecánica, óptica, de grabación o de fotocopia), distribución, comunicación pública y transformación de cualquier parte de ésta publicación -incluído el diseño de la cubierta- sin la previa autorización escrita de los titulares de la propiedad intelectual y de la Editorial. La infracción de los derechos mencionados puede ser constitutiva de delito contra la propiedad intelectual. Los Editores no se pronuncian, ni expresan ni implícitamente, respecto a la exactitud de la información contenida en este libro, razón por la cual no puede asumir ningún tipo de responsabilidad en caso de error u omisión.

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

Colombia *Colombia*
Impreso en Cali, *Printed in Cali,*

PROCESAMIENTO DIGITAL DE VOZ E IMÁGENES EN LA ERA DE LA INTELIGENCIA ARTIFICIAL

Dora Maria Ballesteros

dora.ballesteros@unimilitar.edu.co

UNIVERSIDAD MILITAR NUEVA GRANADA
INGENIERÍA EN TELECOMUNICACIONES

DORA MARIA BALLESTEROS

Dora María Ballesteros es profesora e investigadora de la Universidad Militar Nueva Granada (UMNG) desde 2007. Es Ingeniera Electrónica, con Maestría y Doctorado en Ingeniería Electrónica, y ha dedicado más de veinte años a la docencia, la investigación y el desarrollo de aplicaciones en procesamiento digital de señales, inteligencia artificial y ciencia de datos. Es autora de siete libros y cerca de 90 artículos científicos. Su trabajo se centra en la integración de fundamentos matemáticos, experimentación reproducible y soluciones aplicadas a problemas reales de ingeniería.

Fruto de la labor docente e investigativa desarrollada en la Universidad Militar Nueva Granada, este libro presenta una visión integrada del procesamiento digital de señales tradicional y los enfoques modernos basados en inteligencia artificial. Utilizando Python como entorno de visualización y experimentación, la obra combina fundamentos matemáticos, análisis de señales, experimentación con Python y casos de estudio que ilustran la aplicación de técnicas tradicionales y modernas para resolver problemas reales de ingeniería.

MI CUARTO LIBRO, EL MÁS ESPERADO

Las señales han sido parte de mi vida durante más de veinte años. Las he estudiado, enseñado y programado como profesora e investigadora universitaria, trabajando con señales biomédicas, voz e imágenes digitales. Este es mi cuarto libro de procesamiento digital de señales (PDS): comenzó con mi primera publicación en 2010, continuó con los libros de 2018 y 2023, y hoy llega esta nueva edición. Lo más bonito es que ninguno ha sido una simple actualización del anterior; cada uno ha representado un nuevo hito en mi forma de entender, investigar y enseñar esta disciplina.

Y ahora, en 2026, llegó un momento en el que sentí que ya no era suficiente enseñar procesamiento digital de señales únicamente desde el enfoque tradicional. Me di cuenta de que no podía limitarme a explicar únicamente lo que se utilizó hace años y que todavía sigue siendo valioso, sino también aquello que probablemente se utilizará de manera creciente cuando mis estudiantes se gradúen y comiencen su vida profesional.

Las tecnologías cambian, las herramientas evolucionan y los algoritmos se transforman. Sin embargo, la capacidad de comprender una señal, interpretar la información que contiene y utilizarla para resolver problemas seguirá siendo una habilidad valiosa durante muchos años. Los conceptos permanecen; lo que cambia son las herramientas que construimos sobre ellos.

Por esto, con este libro (*que no será el último, se los garantizo*) espero acompañar no solo a mis estudiantes actuales, sino también a quienes regresen a estas páginas años después de haberse graduado. Me gustaría que, al volver a leerlo, encuentren conceptos que continúen siendo útiles, herramientas que sigan teniendo aplicación y, sobre todo, una forma de pensar la ingeniería que conserve su valor a pesar del paso del tiempo.

Y, para finalizar estas breves palabras, quiero dejarte una selección de publicaciones resultado de los proyectos de investigación que he desarrollado durante los últimos años. En ellas encontrarás aplicaciones del procesamiento digital de señales en voz e imágenes, utilizando tanto técnicas clásicas como enfoques basados en inteligencia artificial. Si este libro despierta tu curiosidad, espero que estas lecturas sean el siguiente paso para seguir aprendiendo, investigando y descubriendo nuevas posibilidades.

Dora Ma. Ballesteros



LECTURAS RECOMENDADAS

Si deseas profundizar en algunos de los temas abordados en este libro, a continuación, encontrarás una selección de publicaciones resultado de proyectos de investigación desarrollados durante los últimos años. Las he organizado de acuerdo con las principales temáticas estudiadas a lo largo del libro, con el propósito de facilitar la continuidad de tu proceso de aprendizaje.

Procesamiento Digital de Señales



- Renza, D., Ballesteros, D. M., & Lemus, C. A. (2018). Authenticity verification of audio signals based on fragile watermarking for audio forensics. *Expert Systems with Applications*, 91, 211–222.
  https://doi.org/10.1016/j.eswa.2017.08.055
- Ballesteros, D. M., & Moreno, J. M. (2012). Highly transparent steganography model of speech signals using efficient wavelet masking. *Expert Systems with Applications*, 39(10), 9141–9149.
  https://doi.org/10.1016/j.eswa.2012.02.102



Procesamiento Digital de Imágenes



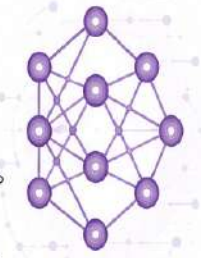
- Pachón, C., Renza, D., & Ballesteros, D. M. (2023). Is my pruned model trustworthy? PE-Score: A new CAM-based evaluation metric. *Big Data and Cognitive Computing*, 7(2), 102.
  https://doi.org/10.3390/bdcc7020102
- Ballesteros, D. M., Peña, J., & Renza, D. (2018). A novel image encryption scheme based on Collatz conjecture. *Entropy*, 20(12), 904.
  https://doi.org/10.3390/e20120901



Inteligencia Artificial aplicada al procesamiento de Señales e Imágenes



- Ballesteros, D. M., Suarez, D., & Pachón, C. (2026). Spectral multi-representation fusion for audio deepfake detection. *Algorithms*, 19(7), 549.
  https://doi.org/10.3390/a19070549
- Pachón, C., & Ballesteros, D. M. (2026). FakeVoiceFinder: An opensource framework for synthetic and deepfake audio detection. *Big Data and Cognitive Computing*, 10(1), 25.
  https://doi.org/10.3390/bdcc10010025
- Pachón, C. G., Pinzón-Arenas, J. O., & Ballesteros, D. M. (2024). Pruning policy for image classification problems based on deep learning. *Informatics*, 11(3), 67.
  https://doi.org/10.3390/informatics11030067
- Chavarro, A. F., Renza, D., & Ballesteros, D. M. (2023). Influence of hyperparameters in deep learning models for coffee rust detection. *Applied Sciences*, 13(7), 4565.
  https://doi.org/10.3390/app13074565
- Pachón, C. G., Ballesteros, D. M., & Renza, D. (2023). An efficient deep learning model using network pruning for fake banknote recognition. *Expert Systems with Applications*, 231, 120961.
  https://doi.org/10.1016/j.eswa.2023.120961
- Rodríguez-Ortega, Y., Ballesteros, D. M., & Renza, D. (2021). Copy-move forgery detection (CMFD) using deep learning for image and video forensics. *Journal of Imaging*, 7(3), 59.
  https://doi.org/10.3390/jimaging7030059
- Ballesteros, D. M., Rodríguez-Ortega, Y., Renza, D., & Arce, G. (2021). Deep4SNet: deep learning for fake speech classification. *IEEE Access*, 9, 145156–145169.
  https://doi.org/10.1109/access.2021.115545





Estas publicaciones son solo un punto de partida. Te invito a explorarlas, cuestionarlas y seguir construyendo tu propio camino de aprendizaje.



ANTES DE EMPEZAR

Este libro habrá cumplido su propósito si,
al finalizar su lectura, comprendes que la mejor forma de
aprender no consiste en elegir entre los métodos clásicos
o la Inteligencia Artificial, sino en entender cuándo,
por qué y cómo utilizar cada uno de ellos.

REPOSITORIO

Este libro fue concebido para leerse con un computador al lado. A lo largo de sus capítulos encontrarás ejemplos desarrollados en Python y notebooks disponibles en un repositorio público. Mi recomendación es que vayas más allá de simplemente ejecutarlos: analízalos, modifícalos, experimenta con ellos y observa cómo cambian los resultados. En ingeniería, muchas veces el aprendizaje más valioso ocurre precisamente cuando una solución no produce el resultado esperado.

En el siguiente repositorio encontrarás los notebooks utilizados a lo largo del libro, organizados por capítulos. Además, se incluyen los ejemplos matemáticos desarrollados durante el texto, los casos de estudio, las imágenes y los archivos de audio empleados en los diferentes experimentos. Mi invitación es a utilizarlos como punto de partida para crear tus propias soluciones y seguir explorando este fascinante mundo del procesamiento digital de señales y la Inteligencia Artificial.

Repositorio oficial del libro:

https://github.com/doramariaballesteros/Libro_procesamiento_digital_de_voz_e_imagenes_en_la_era_de_la_IA

TABLA DE CONTENIDO
LA IA Y EL PROCESAMIENTO DIGITAL DE SEÑALES

PAG

SECCIÓN I.
PROCESAMIENTO DIGITAL DE AUDIO E INTELIGENCIA ARTIFICIAL 21

CAPÍTULO 1. Fundamentos del Procesamiento Digital de Señales y filtros digitales..... 25
CAPÍTULO 2. Eliminación de ruido con filtros digitales..... 91
CAPÍTULO 3. IA para localización de Interferencias y filtrado espectral..... 125

SECCIÓN II.
PROCESAMIENTO DIGITAL DE IMÁGENES E INTELIGENCIA ARTIFICIAL..... 147

CAPÍTULO 4. Fundamentos del Procesamiento Digital de Imágenes (PDI)..... 151
CAPÍTULO 5. Segmentación de Imágenes con técnicas clásicas de pdi..... 203
CAPÍTULO 6. Segmentación Automática con Inteligencia Artificial..... 223

AL CERRAR ESTE LIBRO…..... 251

LA IA Y EL PROCESAMIENTO DIGITAL DE SEÑALES

La inteligencia artificial llegó para quedarse. Hoy forma parte de múltiples sistemas que utilizamos diariamente y está transformando la manera en que las máquinas analizan información, reconocen patrones y apoyan procesos de toma de decisión.

Esta transformación también ha impactado profundamente el procesamiento digital de señales. Disciplinas tradicionales como el filtrado digital, el análisis espectral y la extracción de características continúan siendo fundamentales, pero ahora conviven con modelos capaces de aprender automáticamente a partir de grandes volúmenes de datos.



Figura 1. Beneficios de la IA en el Procesamiento Digital de Señales

La Figura 1 muestra algunas de las razones por las cuales la IA se ha convertido en una herramienta tan importante dentro del procesamiento digital de señales. Gracias a ella es posible mejorar tareas de clasificación, detección de patrones y reducción de ruido, así como extraer información cada vez más compleja a partir de señales de voz, imágenes, datos biomédicos y otras fuentes de información. Estas capacidades han impulsado el desarrollo de nuevas aplicaciones y han ampliado significativamente el alcance de esta disciplina.

Sin embargo, y en esto quiero hacer especial énfasis, la inteligencia artificial no reemplaza los fundamentos clásicos del procesamiento digital de señales; **los complementa**. Por esta razón, el profesional moderno necesita comprender ambos mundos. Este libro busca precisamente construir ese puente entre los conceptos tradicionales y las herramientas inteligentes que hoy forman parte de la ingeniería contemporánea.

SECCIÓN I

PROCESAMIENTO DIGITAL DE AUDIO E INTELIGENCIA ARTIFICIAL

Llevo más de veinte años trabajando con Procesamiento Digital de Señales (PDS), particularmente con señales unidimensionales que evolucionan en el tiempo. Durante estos años he utilizado técnicas clásicas de análisis, filtrado y reconocimiento de patrones, pero también he sido testigo de la llegada de nuevas herramientas basadas en Inteligencia Artificial (IA), las cuales han ampliado considerablemente las posibilidades de procesamiento de este tipo de señales.

Entre todas las señales con las que he trabajado, existe una que ha tenido un significado especial para mí: la señal de voz. Fue la protagonista de mi tesis doctoral y continúa siendo una de las señales que más disfruto estudiar, investigar y enseñar. Detrás de una voz encontramos información relacionada con frecuencias, energía, ruido, emociones, identidades y patrones que pueden analizarse desde múltiples perspectivas.

En esta primera sección utilizaremos la señal de voz como hilo conductor para estudiar una de las tareas más importantes del Procesamiento Digital de Señales: la reducción de ruido. A partir de un mismo caso de estudio, recorreremos tanto herramientas clásicas de filtrado digital como estrategias modernas basadas en Inteligencia Artificial, analizando sus ventajas, limitaciones y criterios de evaluación.

La sección está organizada en tres capítulos estrechamente relacionados entre sí (Figura 2):

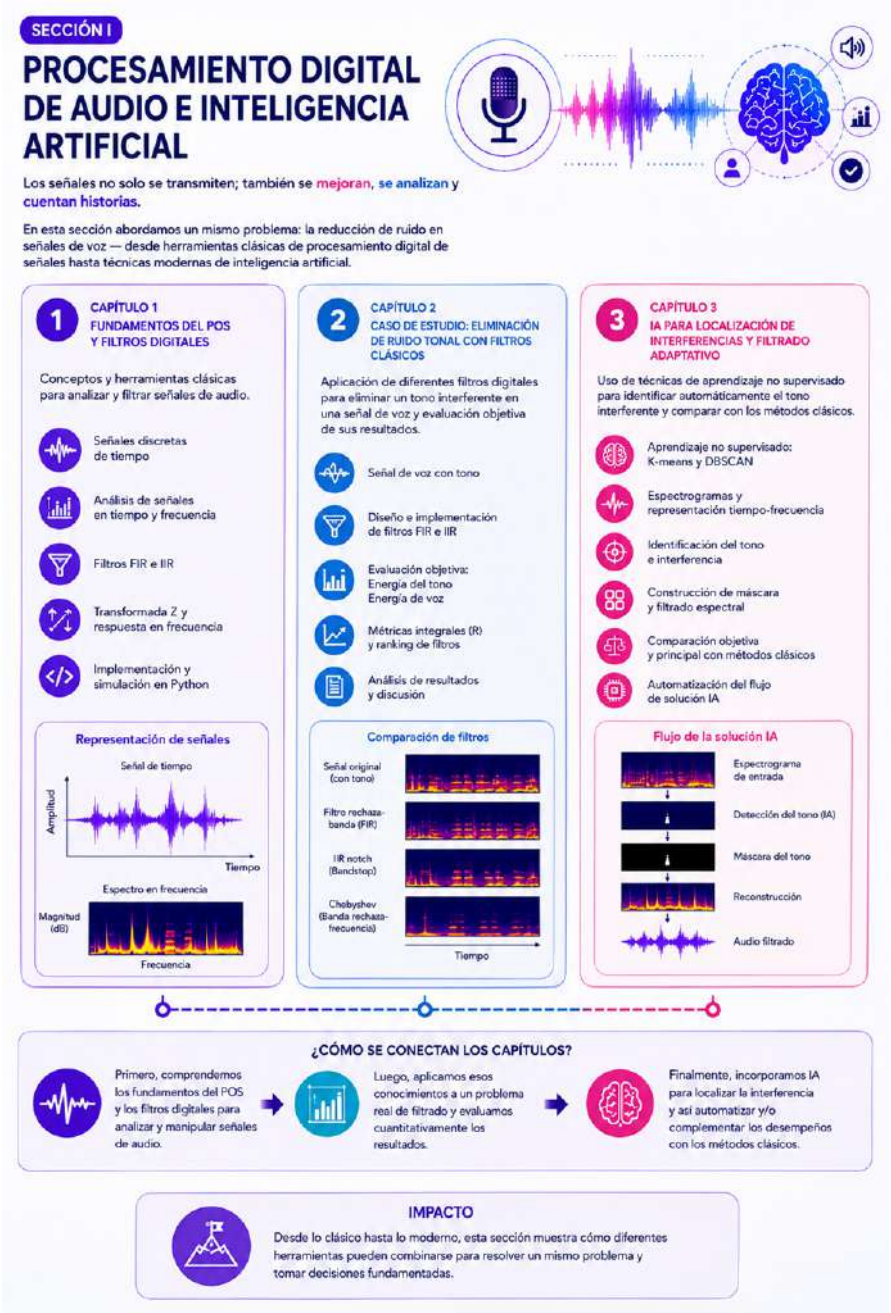


Figura 2. Sección1 – recorrido temático: desde el procesamiento clásico de señales de audio hasta aplicaciones modernas de inteligencia artificial sobre señales de voz.

En el Capítulo 1 estudiaremos los fundamentos necesarios para comprender el funcionamiento de los filtros digitales. Introduciremos conceptos relacionados con señales discretas, sistemas, Transformada Z, filtros FIR e IIR, así como diferentes estrategias de diseño y análisis utilizando Python como herramienta de simulación.

En el Capítulo 2 aplicaremos estos conocimientos a un caso de estudio real de reducción de ruido tonal en una señal de voz. Compararemos diferentes filtros digitales, analizaremos sus resultados mediante criterios objetivos de evaluación y construiremos una metodología para seleccionar la solución más adecuada.

En el Capítulo 3 exploraremos cómo la Inteligencia Artificial puede integrarse dentro de un problema de filtrado de audio. Utilizaremos técnicas de aprendizaje no supervisado para identificar automáticamente la interferencia presente en el espectrograma y compararemos sus resultados con los obtenidos mediante los métodos clásicos estudiados anteriormente.

Más allá de aprender algoritmos o escribir código, el objetivo de esta sección es comprender cómo diferentes herramientas pueden combinarse para resolver un mismo problema.

Porque en ingeniería rara vez existe una única solución correcta; lo realmente importante es entender las fortalezas y limitaciones de cada alternativa para tomar decisiones fundamentadas.

CAPÍTULO I

FUNDAMENTOS DEL PROCESAMIENTO DIGITAL DE SEÑALES Y FILTROS DIGITALES

Las señales rara vez llegan a nosotros en condiciones ideales. Durante su adquisición, transmisión o almacenamiento pueden aparecer perturbaciones que afectan su calidad y dificultan tanto su percepción por parte de un oyente como su posterior procesamiento automático. En el caso de la voz, estas alteraciones resultan especialmente relevantes, ya que se trata de una señal ampliamente utilizada en aplicaciones de telecomunicaciones, biometría, asistentes virtuales e inteligencia artificial.

En este capítulo estudiaremos el diseño y aplicación de filtros FIR e IIR orientados principalmente a la reducción de ruido en señales unidimensionales. Además de presentar sus fundamentos teóricos, analizaremos criterios de diseño, ventajas, limitaciones y escenarios de aplicación.

A través de conceptos matemáticos, análisis espectral y experimentación práctica en Python, podrás observar cómo las decisiones de diseño influyen directamente en la calidad de la señal obtenida. Más allá de aplicar filtros, el objetivo es desarrollar la capacidad de interpretar el comportamiento de una señal y tomar decisiones fundamentadas frente a distintos problemas de procesamiento.

1.1. Señales y Transformada Z

Una señal discreta es aquella cuya variable independiente toma valores en instantes específicos (de tiempo) y separados entre sí. A diferencia de una señal continua en la cual la variable independiente está definida para cualquier valor (de tiempo), una señal discreta está compuesta por un conjunto finito o infinito de muestras obtenidas en posiciones determinadas.

Para este tipo de señal, la variable independiente la representamos mediante n, donde cada valor de n corresponde a una muestra de la señal. Por esta razón, el eje horizontal no corresponde a tiempo continuo, sino a la secuencia de muestras que conforman la señal. En la práctica, podemos imaginar una señal discreta como una lista ordenada de números que describen el comportamiento de un fenómeno físico en distintos instantes de observación. Es precisamente esta representación la que permite almacenar, transmitir y procesar señales mediante herramientas computacionales.

Matemáticamente, una señal discreta suele expresarse como:
$x[n]$

donde x representa la amplitud de la señal y n identifica la posición de cada muestra dentro de la secuencia.

Las señales discretas aparecen de forma natural en sistemas digitales. Por ejemplo, cuando una señal de voz es capturada por un micrófono y posteriormente almacenada en un computador, esta se convierte en una secuencia de muestras numéricas que pueden ser procesadas mediante algoritmos digitales.

Ejemplo 1. Señal discreta en Python de duración finita

Como primer ejemplo, consideremos una señal discreta definida por los valores de sus muestras para $n = -2n = -2$ y $n = 5$ (Ver Figura 2).

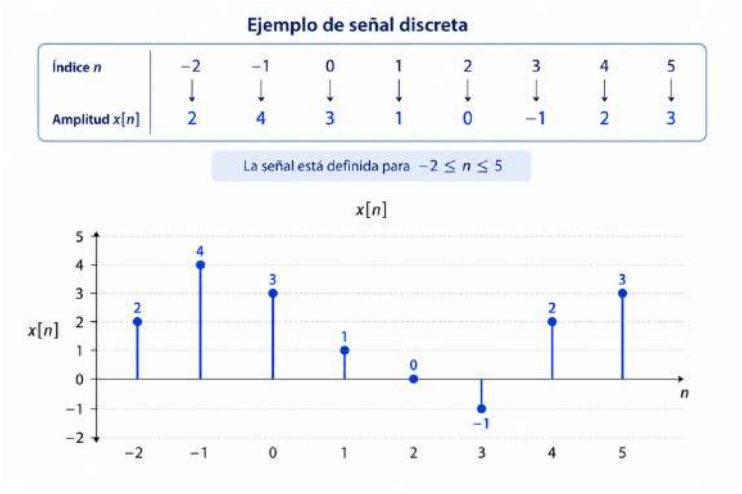


Figura 3. Ejemplo 1: señal discreta definida mediante un conjunto finito de muestras. Cada valor de n representa la posición de una muestra dentro de la secuencia, mientras que $x[n]$ es la amplitud de la señal.

Esta señal contiene un total de 8 muestras, y se puede expresar como una sumatoria de impulsos desplazados, así:

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

Cabe recordar que un impulso ubicado en $n = 0$ se representa como $\delta[n]$. Si desplazamos dicho impulso a la posición $n = k$, utilizamos la expresión $\delta[n-k]$. Esta notación indica que el impulso aparecerá exactamente cuando $n = k$. Por ejemplo, $\delta[n-3]$ representa un impulso ubicado en $n = 3$, mientras que $\delta[n+2]$ representa un impulso ubicado en $n = -2$. En general, los términos de la forma $\delta[n-k]$ corresponden a impulsos desplazados hacia la derecha del origen, y los términos de la forma $\delta[n+k]$corresponden a impulsos desplazados hacia la izquierda.

Nota aclaratoria: un signo negativo desplaza el impulso hacia la derecha ($\delta[n-k]$), mientras que un signo positivo lo desplaza hacia la izquierda ($\delta[n+k]$).

Ahora, vamos a escribir el siguiente código en Python, para crear y dibujar esta señal:

```python
import numpy as np
import matplotlib.pyplot as plt

# Definición de la señal discreta
n = np.array([-2, -1, 0, 1, 2, 3, 4, 5])
x = np.array([2, 4, 3, 1, 0, -1, 2, 3])

# Gráfica tipo stem
plt.figure(figsize=(9, 5))

plt.stem(n, x, basefmt="k-", linefmt="b-", markerfmt="bo")

# Ejes
plt.axhline(0, color="black", linewidth=1)
plt.axvline(0, color="black", linewidth=1)

# Etiquetas
plt.xlabel(r"$n$", fontsize=12)
plt.ylabel(r"$x[n]$", fontsize=12)

# Marcas de los ejes
plt.xticks(n)
plt.yticks(np.arange(-2, 6, 1))

# Cuadrícula
plt.grid(True, linestyle="--", alpha=0.5)

# Valores sobre cada muestra
for ni, xi in zip(n, x):
    plt.text(ni, xi + 0.2, str(xi), ha="center", fontsize=11)

plt.show()
```

Figura 4. Código en Python del Ejemplo 1. Los vectores *n* y *x* almacenan los índices y amplitudes de la señal, respectivamente, mientras que la función stem() permite visualizar cada muestra de manera individual.

Este código en Python permite representar gráficamente la señal discreta definida para $-2 \leq n \leq 5$. Para ello, se emplea la función stem(), la cual es adecuada para visualizar señales discretas, ya que representa cada muestra mediante una línea vertical. Conceptualmente, estas líneas pueden interpretarse como impulsos desplazados ubicados en las posiciones $n = k$, cuya amplitud corresponde al valor de la muestra $x[k]$.

Ahora vamos a conocer por primera vez la Transformada Z (TZ) de esta señal. Por el momento, no nos preocuparemos por cómo se calcula; simplemente observaremos el resultado e intentaremos identificar algunas similitudes entre $x[n]$y $X(z)$.

Para nuestro Ejemplo 1, la TZ de la señal se expresa como:

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

A partir de esta expresión podemos realizar las siguientes observaciones:

- La expresión contiene el mismo número de términos que impulsos tiene la señal original.
- Cada impulso de la señal genera un término en la Transformada Z.
- Si un impulso está ubicado en valores negativos de n (es decir, a la izquierda del origen), la potencia de z aparece con exponente positivo.
- Si un impulso está ubicado en valores positivos de n (es decir, a la derecha del origen), la potencia de z aparece con exponente negativo.
- El impulso ubicado en $n = 0$ genera el término z^0, el cual es igual a 1.
- El coeficiente que multiplica a cada potencia de z coincide con la amplitud del impulso correspondiente en la señal original.

Por ejemplo:

$$2\delta[n+2] \longrightarrow 2z^2$$

$$4\delta[n+1] \longrightarrow 4z^1$$

$$3\delta[n] \longrightarrow 3z^0$$

$$-\delta[n-3] \longrightarrow -z^{-3}$$

Al observar estos resultados, parece existir una relación directa entre la posición de cada impulso en $x[n]$y el exponente de $z$en $X(z)$.

1.2. Función de Transferencia, Polos y Ceros

En el Ejemplo 1, de manera intuitiva hemos calculado la Transformada Z (TZ) de una señal de duración finita. Sin embargo, esta transformada no se utiliza únicamente para representar señales, sino también para representar sistemas discretos.

Supongamos que una señal de entrada $x[n]$ ingresa a un sistema digital y produce una señal de salida $y[n]$:

$$x[n] \longrightarrow \boxed{\text{Sistema}} \longrightarrow y[n]$$

Al aplicar la Transformada Z tanto a la entrada como a la salida, obtenemos:

$$x[n] \overset{TZ}{\rightarrow} X(z)$$

$$y[n] \overset{TZ}{\rightarrow} Y(z)$$

La relación entre ambas señales se define mediante la **función de transferencia**:

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

Esta expresión nos permite describir matemáticamente el comportamiento del sistema en el dominio Z.

Ahora bien, recordemos que un sistema discreto puede caracterizarse completamente mediante su *respuesta al impulso*, denotada por $h[n]$. Al aplicar TZ a dicha respuesta obtenemos:

$$h[n] \overset{TZ}{\rightarrow} H(z)$$

Es decir, la función de transferencia $H(z)$ corresponde a la TZ de la respuesta al impulso del sistema.

Por esta razón, la función de transferencia puede interpretarse como una representación alternativa del sistema: mientras que $h[n]$ describe su comportamiento en el dominio discreto, $H(z)$ describe ese mismo comportamiento en el dominio Z.

Así como una señal puede representarse mediante $x[n]$ o mediante $X(z)$, un sistema discreto también puede representarse mediante su respuesta al impulso $h[n]$ o mediante su función de transferencia $H(z)$. Esto nos permite analizar el tipo de sistema, por ejemplo, si es un filtro FIR o IIR, si es causal o no causal y en qué condiciones es estable.

Ejemplo 2. Función de transferencia a partir de la ecuación de entrada-salida del sistema

Supongamos que tenemos el siguiente sistema discreto:

$$y[n] = \frac{1}{9}\{3x[n] + 5x[n-1] + x[n-2]\}$$

Y queremos conocer su función de transferencia.

Antes de entrar en el desarrollo matemático, vamos a utilizar una de las propiedades de la TZ, correspondiente a la ***propiedad de desplazamiento***, que consiste en:

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

Es decir, si desplazo una señal, la TZ de la señal desplazada va a ser igual a la de la señal original multiplicada por el término z^{-k}.

Entonces, al aplicar la TZ a cada uno de los términos de la ecuación de entrada-salida del sistema, tenemos que:

$$Y(z) = \frac{1}{9}\{3X(z) + 5z^{-1}X(z) + z^{-2}X(z)\}$$

Factorizando $X(z)$:

$$Y(z) = \frac{1}{9}(3+5z^{-1}+z^{-2})X(z)$$

Como:

$$Y(z) = H(z)X(z)$$

entonces:

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

$$H(z) = \frac{1}{9}(3+5z^{-1}+z^{-2})$$

Por lo tanto:

$$\boxed{H(z) = \frac{3 + 5z^{-1} + z^{-2}}{9}}$$

Ejemplo 3. Polos y ceros a partir de la función de transferencia

En el Ejemplo 2 se obtuvo la siguiente función de transferencia:

$$H(z) = \frac{3 + 5z^{-1} + z^{-2}}{9}$$

Para identificar polos y ceros, resulta conveniente expresar $H(z)$ en potencias positivas de z. Multiplicando numerador y denominador por z^2, obtenemos:

$$H(z) = \frac{3z^2 + 5z + 1}{9z^2}$$

Los ceros corresponden a las raíces del polinomio del numerador de $H(z)$; mientras que, los polos corresponden a las raíces del polinomio de su denominador

Ahora podemos analizar por separado el numerador y el denominador.

Paso 1. Cálculo de los ceros

Los ceros corresponden a los valores de z que hacen cero el numerador:

$$3z^2 + 5z + 1 = 0$$

Aplicando la fórmula cuadrática:

$$z = \frac{-5 \pm \sqrt{25 - 12}}{6}$$

$$z = \frac{-5 \pm \sqrt{13}}{6}$$

Por tanto, los ceros son:

$$z_1 = \frac{-5 + \sqrt{13}}{6} \approx -0.232$$

$$z_2 = \frac{-5 - \sqrt{13}}{6} \approx -1.434$$

Paso 2. Cálculo de los polos

Los polos corresponden a los valores de z que hacen cero el denominador:

$$9z^2 = 0$$

$$z^2 = 0$$

$$z = 0$$

Como la potencia es dos, existen dos polos coincidentes en el origen:

$$p_1 = p_2 = 0$$

Para la gráfica de polos y ceros, en el **plano Z** utilizamos círculos ($\circ$) para los ceros, y cruces ($\times$) para los polos.

El siguiente código en Python nos permite dibujar los polos y ceros de nuestro sistema:

```python
import numpy as np
import matplotlib.pyplot as plt

# H(z) = (3z² + 5z + 1)/(9z²)
num = [3, 5, 1]
den = [9, 0, 0]

zeros = np.roots(num)
poles = np.roots(den)
theta = np.linspace(0, 2*np.pi, 500)
plt.figure(figsize=(8, 8))

# Círculo unitario
plt.plot(np.cos(theta), np.sin(theta), '--', linewidth=2,
         label='Círculo unitario')
# Ejes
plt.axhline(0, color='black', linewidth=1)
plt.axvline(0, color='black', linewidth=1)

# Ceros: ahora rellenos para que se vean
plt.plot(np.real(zeros), np.imag(zeros),'o', markersize=12, label='Ce

# Polos
plt.plot(np.real(poles), np.imag(poles),
         'x', markersize=12, markeredgewidth=3, label='Polos')

# Etiquetas de ceros
for i, z in enumerate(zeros, start=1):
    plt.annotate(f'c{i} = {np.real(z):.3f}',
                 xy=(np.real(z), np.imag(z)),
                 xytext=(np.real(z), 0.25),
                 arrowprops=dict(arrowstyle='->'),
                 ha='center')

# Etiqueta única del polo doble
plt.annotate('p = 0\nmultiplicidad 2',
             xy=(0, 0),
             xytext=(0.55, 0.35),
             arrowprops=dict(arrowstyle='->'))

plt.xlabel('Parte real'); plt.ylabel('Parte imaginaria')
plt.title('Diagrama de polos y ceros')

plt.grid(True); plt.axis('equal'); plt.xlim(-2, 2)
plt.ylim(-2, 2); plt.legend(); plt.show()
```

Figura 5. Código en Python del Ejemplo 3: Diagrama de Polos y Ceros

Permitiéndonos obtener la siguiente gráfica de polos y ceros:

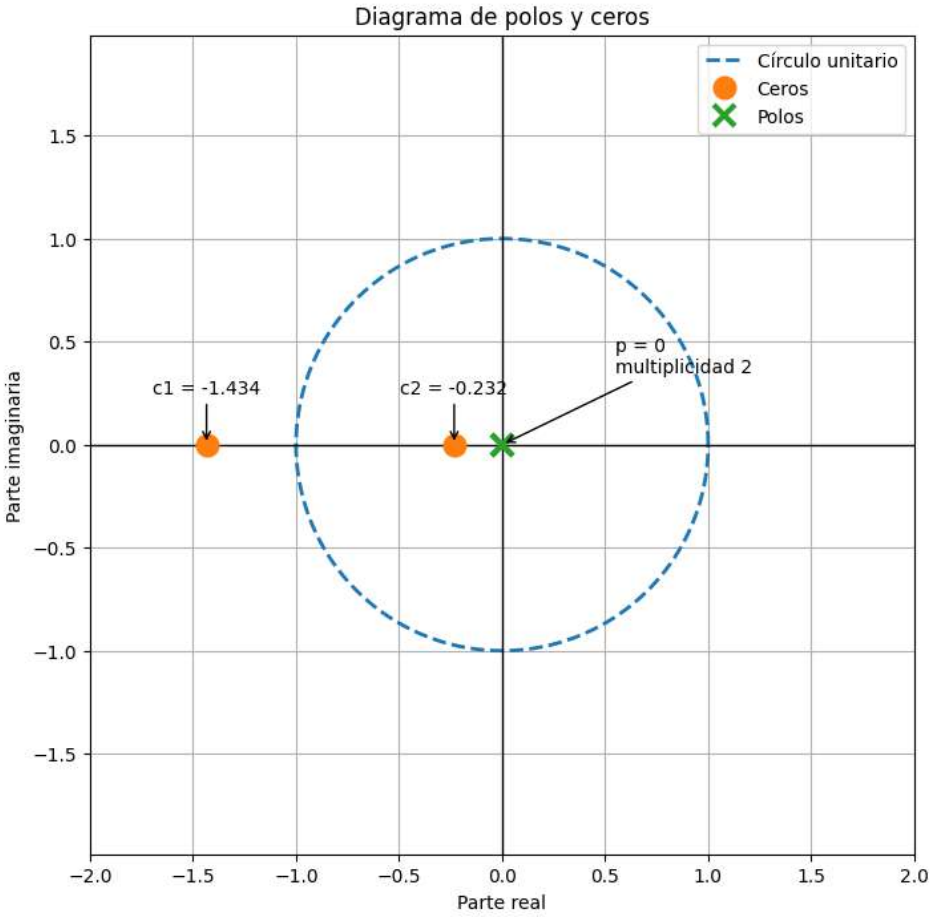


Figura 6. Diagrama de polos y ceros del sistema del Ejemplo 3. Los círculos representan los ceros del sistema y las cruces representan los polos. La línea punteada corresponde al círculo unitario ($|z| = 1$), sirve como referencia para analizar la ubicación relativa de polos y ceros dentro del plano Z.

1.3. Significado de los Polos y Ceros

Te preguntarás, ¿por qué es importante esta gráfica de polos y ceros? La respuesta la obtenemos en la siguiente imagen. Los ceros están asociados al tipo de filtro y los polos a la selectividad y estabilidad del filtro.

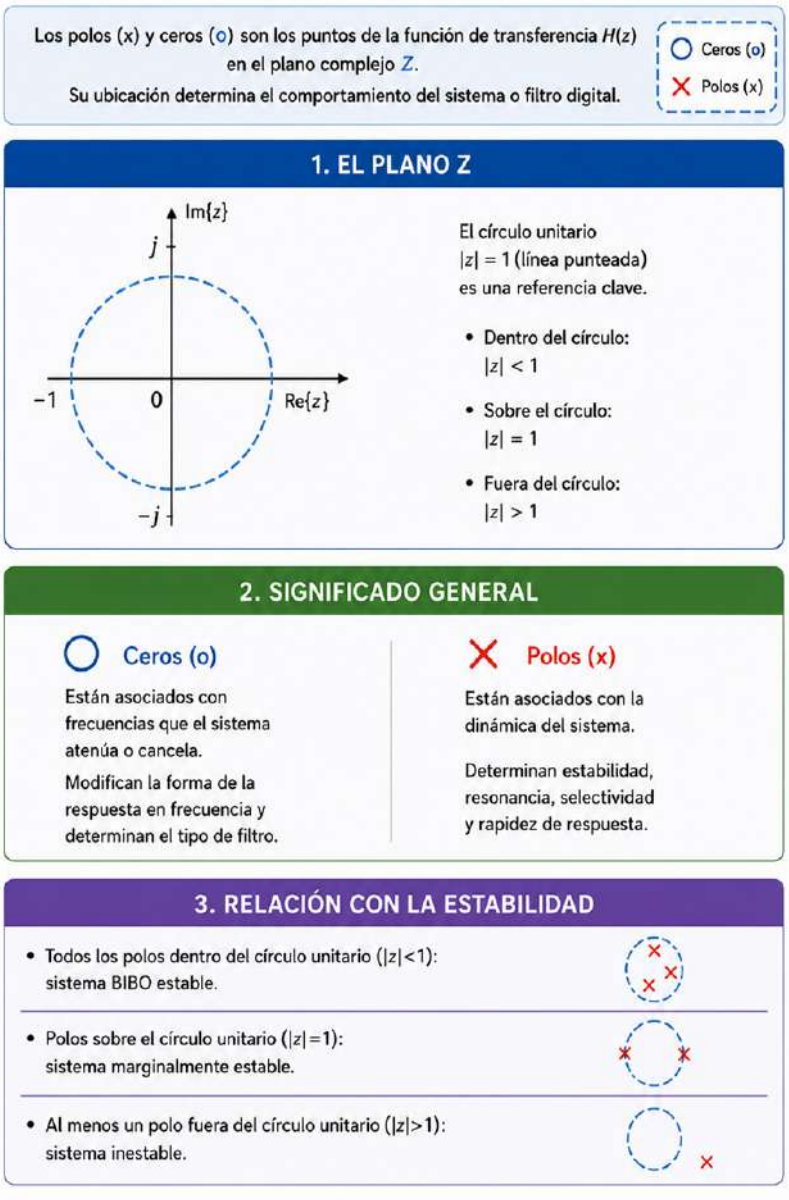


Figura 7. Aspectos importantes de la ubicación de los polos y ceros.

Ahora, vamos a analizar con detalla el significado de cada uno de ellos. En el caso de los polos, debemos considerar dos aspectos: distancia del origen (Figura 8), y ángulo de su ubicación (Figura 9).



Figura 8. Influencia de la ubicación radial del polo en el plano-z sobre la respuesta en frecuencia de un filtro pasa-bajo. Se comparan tres valores del radio, mostrando su efecto sobre la pendiente de transición y la selectividad del filtro.

En términos de distancia al origen (Figura 8), entre más próximo está el/los polos al origen, el filtro tiene una respuesta más suave; mientras que, entre más alejado se encuentre del origen, su respuesta es más pronunciada, o es más selectivo.

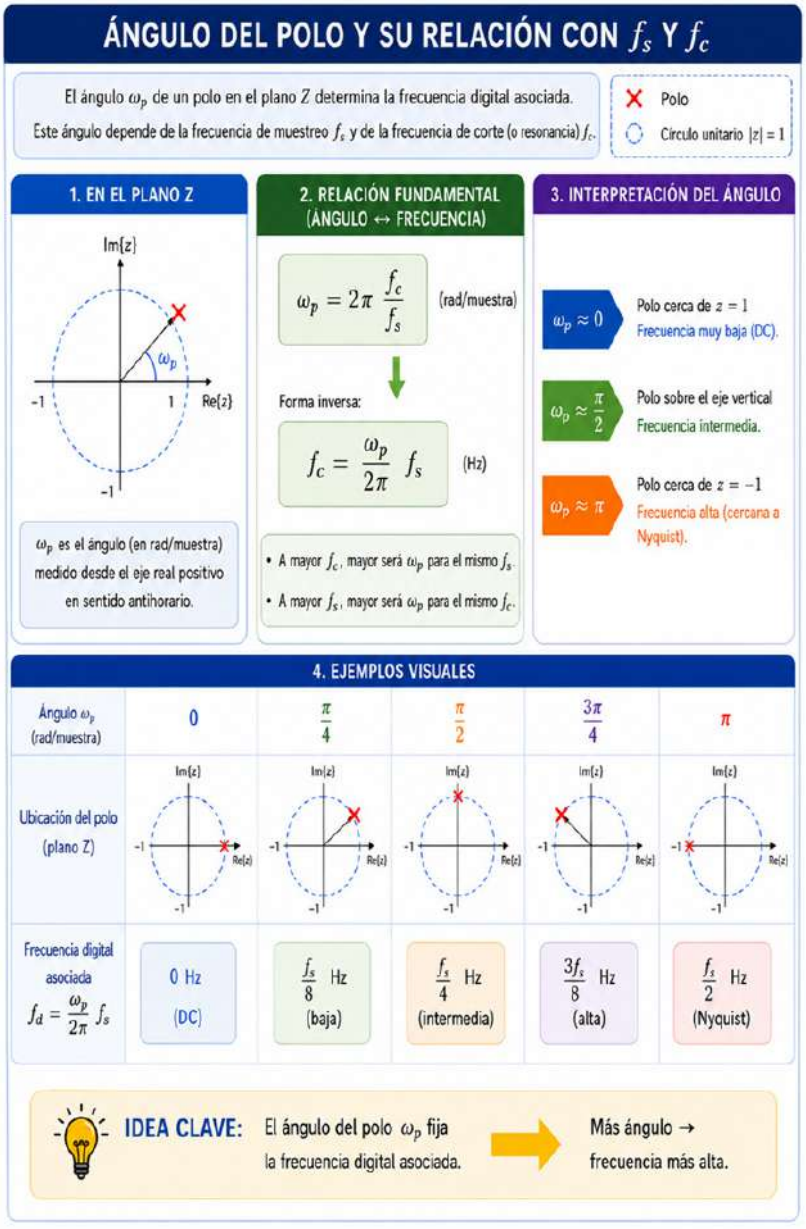


Figura 9. Interpretación del ángulo de los polos y su relación con *fs* y *fc*.

En términos del ángulo de ubicación, tenemos la Figura 9. A medida que la frecuencia de corte del filtro digital se aproxime a la mitad de la frecuencia de muestreo, los polos ampliarán su ángulo de ubicación, es decir, se desplazarán del semiplano derecho al semiplano izquierdo.

Finalmente, el significado de la ubicación de los ceros lo tenemos en la Figura 10. Su ubicación está relacionada con el tipo de filtro:

- Un cero en $z = -1$ ($\omega = \pi$) atenúa las altas frecuencias y se asocia con un filtro pasa-bajos.
- Un cero en $z = 1$ ($\omega = 0$) atenúa las bajas frecuencias y se asocia con un filtro pasa-altos.
- Cuando existen ceros simultáneamente en $z = 1$ y $z = -1$, se atenúan tanto las bajas como las altas frecuencias (favoreciendo el paso de un rango intermedio de frecuencias), y se asocia con un filtro pasa-banda.
- Finalmente, un par de ceros complejos conjugados ubicados en $z = e^{\pm j\omega_0}$ genera una atenuación localizada alrededor de la frecuencia ω_0, característica de los filtros rechaza-banda o *notch*.

En consecuencia, los ceros pueden interpretarse como puntos donde el sistema atenúa o cancela determinadas frecuencias, proporcionando una primera aproximación al tipo de filtrado que realiza el sistema.

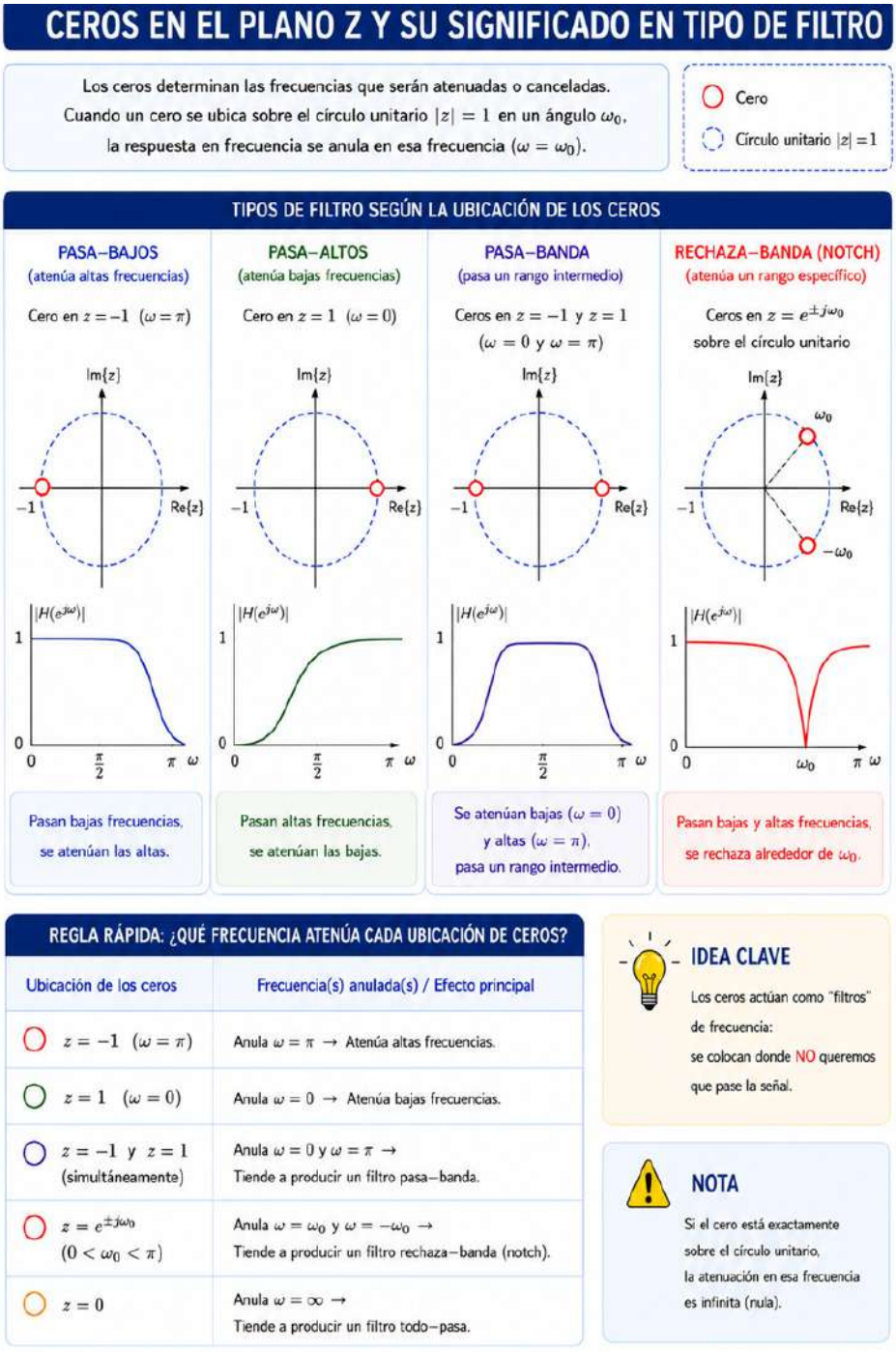


Figura 10. Interpretación de la posición de los ceros y tipo de filtro.

1.4. Filtros digitales

Cuando una señal contiene ruido, interferencias o información que no relevante, necesitamos de alguna “quedarnos con lo importante” y reducir aquello que puede dificultar su análisis. Precisamente para eso existen los filtros. Desde hace décadas, constituyen una de las herramientas más utilizadas del procesamiento de señales y continúan siendo fundamentales incluso en la era de la inteligencia artificial.

La idea de un filtro es sencilla: permitir el paso de ciertas frecuencias mientras atenúa otras. En esencia, un filtro digital cumple el mismo propósito que un filtro analógico. La diferencia es que el filtro analógico opera sobre señales continuas en el tiempo y se describe mediante la variable s, mientras que el filtro digital trabaja sobre señales discretas y utiliza la variable z. Aunque las matemáticas cambian, el objetivo sigue siendo el mismo: modificar el contenido frecuencial de una señal para facilitar su interpretación o mejorar su calidad.

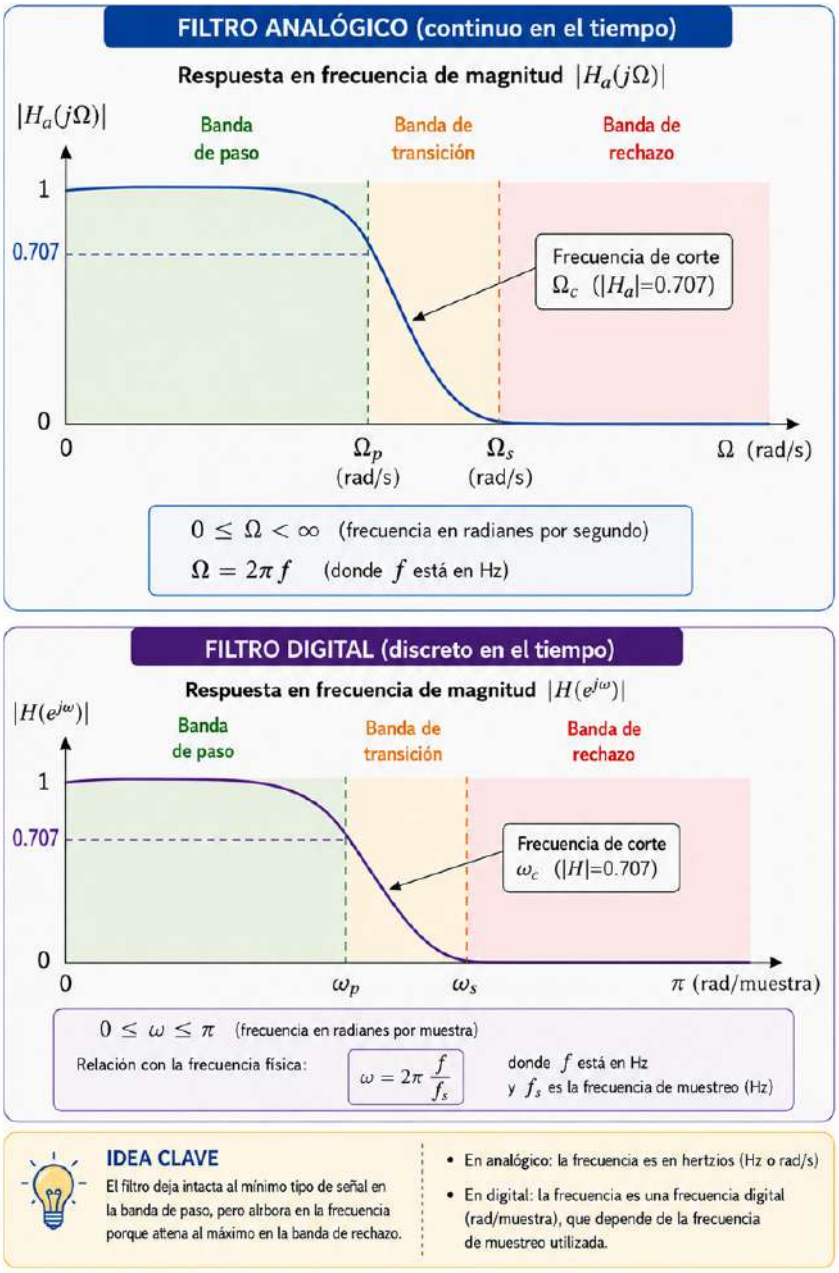


Figura 11. Comparativo entre un filtro análogo y un filtro digital. En el caso del filtro análogo, las frecuencias se expresan en rad/s o Hz, y en principio el valor máximo es infinito. Mientras que, en el caso de los filtros digitales, la frecuencia se expresa en rad/muestra y está acotada hasta π.

Los filtros digitales suelen clasificarse en dos grandes grupos: los filtros de respuesta al impulso finita (**FIR**) y los filtros de respuesta al impulso infinita (**IIR**). Los primeros se construyen a partir de una respuesta al impulso de duración finita y suelen ser más intuitivos de comprender, ya que permiten relacionar directamente el comportamiento del sistema con los coeficientes que conforman dicha respuesta. Un ejemplo sencillo de esta familia es el filtro de promedio, que les presentaré en las próximas páginas.

Por su parte, muchos filtros IIR tienen su origen en filtros análogos (ej. Butterworth, Chebyshev o Elípticos) que posteriormente son transformados al dominio digital. En otras palabras, primero diseñamos un filtro en el dominio s y luego realizamos un mapeo hacia el dominio z, obteniendo así una versión digital con características similares. Gracias a esta estrategia podemos aprovechar décadas de desarrollo en filtros analógicos y trasladar ese conocimiento al mundo digital.

1.5. Filtros FIR (Finite Impulse Response)

En este tipo de filtros, la cantidad de impulsos que conforman la respuesta al impulso, $h[n]$, es finita.

Este tema me encanta porque nos conduce a uno de los conceptos más elegantes del procesamiento digital de señales: la convolución entre la señal de entrada del filtro, es decir, la señal que queremos procesar, $x[n]$, y la respuesta al impulso del sistema, $h[n]$.

Recordemos que la respuesta al impulso $h[n]$ describe completamente el comportamiento de un sistema discreto. Por esta razón, si conocemos $h[n]$ y la señal de entrada $x[n]$, podemos entonces calcular la señal de salida mediante la operación de convolución, así:

$$y[n] = x[n] \otimes h[n]$$

donde $\otimes$ es el operador de convolución.

Desde este enfoque, un filtro FIR (incluyendo el filtro de promedio que estudiaremos a continuación), es el resultado de convolucionar la señal de entrada con una respuesta al impulso de duración finita.

Pero, aparece entonces una pregunta muy interesante:

¿qué significa realmente diseñar un filtro?

Diseñar un filtro consiste en determinar las amplitudes de los impulsos que conforman $h[n]$. Dicho de otra manera, consiste en calcular los coeficientes del filtro, los cuales definirán la función de transferencia $H(z)$ y, por tanto, la forma en que el sistema responderá a las diferentes frecuencias presentes en la señal.

Así que, ahora que comprendemos qué representan la respuesta al impulso, la función de transferencia, los polos y los ceros, estamos listos para comenzar el diseño de nuestros primeros filtros digitales.

1.5.1. Filtro de promedio

En términos de diseño, este es probablemente el filtro FIR más sencillo de construir. ¿La razón? Todas las amplitudes son iguales. Solamente necesitamos conocer la cantidad de impulsos que tendrá la respuesta al impulso, pues la amplitud de cada uno será igual a uno dividido entre dicho valor.

Por ejemplo, un filtro de promedio de cinco impulsos puede expresarse como:

$$h[n] = 0.2 \sum_{k=0}^{4} \delta[n-k]$$

En este caso, todos los impulsos tienen una amplitud de 0.2, iniciando en $n = 0$ y terminando en $n = 4$. Quizá te preguntes por qué es necesario escalar las amplitudes. La razón es simple: ***en un filtro de promedio, la suma de todos los pesos debe ser igual a uno***.

Cuando realizamos la convolución entre la respuesta al impulso y la señal de entrada, obtenemos:

$$y[n] = 0.2 \sum_{k=0}^{4} x[n-k]$$

Que también podemos escribir como:

$$y[n] = \frac{x[n] + x[n-1] + x[n-2] + x[n-3] + x[n-4]}{5}$$

Es decir, para obtener la salida en el momento actual, es necesario calcular el promedio del valor actual de la señal de entrada con sus cuatro muestras anteriores. Este promedio, trae como efecto el suavizado de la señal, es decir, funciona como un filtro pasa-bajos, que veremos en detalle a continuación.

1.5.2. Respuesta en frecuencia del filtro de promedio

La respuesta en frecuencia de un filtro de promedio presenta una forma característica compuesta por un lóbulo principal centrado en la frecuencia cero, donde alcanza su máxima amplitud, y varios lóbulos secundarios de menor amplitud, que decrecen a medida que nos alejamos del origen. Siendo el lóbulo principal en responsable del comportamiento pasa-bajos del filtro; mientras que, los lóbulos secundarios corresponden a ese “comportamiento no ideal” de la banda de rechazo.

Adicionalmente, algo muy interesante del filtro de promedio, es que la posición de los cruces por cero depende directamente del valor de M de la respuesta al impulso.

Para un filtro de promedio de longitud M, los ceros de la respuesta en frecuencia ocurren en:

$$\omega_k = \frac{2\pi k}{M}$$

donde $k = 1,2,3, \ldots$

Por tanto, el primer cruce por cero se encuentra en:

$$\omega_1 = \frac{2\pi}{M}$$

A medida que aumenta M, este primer cero se acerca al origen, haciendo que el lóbulo principal se vuelva más estrecho y disminuyendo el rango de frecuencias que el filtro deja pasar (es decir, disminuyendo la frecuencia de corte).

Ejemplo 4. Respuesta en frecuencia filtro de promedio

Vamos a ilustrar con este código en Python cómo es el comportamiento en frecuencia del filtro de promedio, para diferentes valores de M, específicamente

para $M = 4, 5, 6, y\ 7$.

```python
import numpy as np
import matplotlib.pyplot as plt

# Valores de M
M_values = [4, 5, 6, 7]

# Frecuencia digital entre -pi y pi
omega = np.linspace(-np.pi, np.pi, 4000)

plt.figure(figsize=(12, 8))

for i, M in enumerate(M_values, 1):

    # Respuesta en frecuencia
    H = np.zeros_like(omega, dtype=complex)

    for n in range(M):
        H += np.exp(-1j * omega * n)
    H = H / M

    # Magnitud lineal
    H_mag = np.abs(H)

    plt.subplot(2, 2, i); plt.plot(omega, H_mag, linewidth=2)

    plt.title(f"M = {M}", fontsize=14); plt.xlabel(r'$\omega$ (rad/muestra)')
    plt.ylabel(r'$|H(e^{j\omega})|$'); plt.xlim([-np.pi, np.pi]); plt.ylim([0, 1.05])
    plt.grid(True, alpha=0.3)

    plt.xticks(
        [-np.pi, -np.pi/2, 0, np.pi/2, np.pi],
        [r'$-\pi$', r'$-\pi/2$', '0', r'$\pi/2$', r'$\pi$']
    )

plt.tight_layout()
plt.show()
```

Figura 12. Código en Python del Ejemplo 4. Respuesta en frecuencia de filtro de promedio.

Obteniendo,

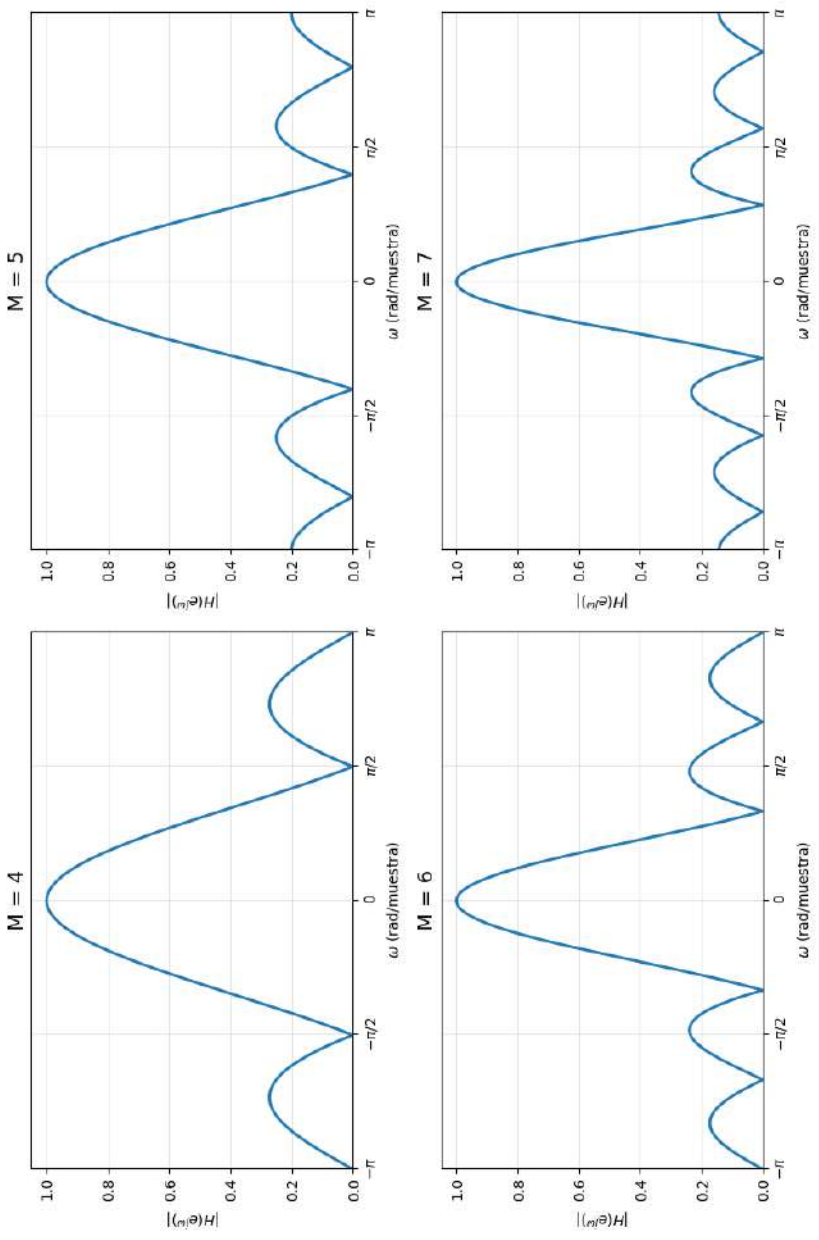


Figura 13. Respuesta en frecuencia filtro de promedio, para $M = 4, 5, 6, y\ 7$.

Podemos observar que, al aumentar M, aparecen más cruces por cero y un mayor número de lóbulos secundarios. Además, dentro del intervalo $[-\pi, \pi]$, el número total de lóbulos es igual a $M - 1$, contando tanto el lóbulo principal como los secundarios. Cuando M es impar, aparecen medios lóbulos en los extremos del intervalo; sin embargo, estos pueden considerarse conjuntamente como un lóbulo completo.

Por esta razón, cuando afirmamos que la frecuencia de corte disminuye al aumentar M, nos referimos a una disminución dentro del dominio de frecuencia digital (ω), es decir, el lóbulo principal ocupa una porción cada vez menor del intervalo $[-\pi, \pi]$.

1.5.3. Método de ventaneo

Ahora vamos a conocer el método de ventaneo, una de las formas más intuitivas de diseñar filtros FIR. La idea central consiste en comenzar con un filtro ideal, cuya respuesta en frecuencia tiene exactamente la forma que deseamos, y luego obtener una versión realizable de dicho filtro.

Por ejemplo, para un filtro pasa bajo ideal, quisiéramos una respuesta en frecuencia perfectamente rectangular:

$$H_d(e^{j\omega}) = \begin{cases} 1, & |\omega| \leq \omega_c \\ 0, & \omega_c < |\omega| \leq \pi \end{cases}$$

donde ω_c es la frecuencia de corte digital, expresada en rad/muestra.

Sin embargo, ese filtro ideal tiene un problema importante: su respuesta al impulso es de duración infinita por ambos lados del eje de tiempo. Es decir, necesitaríamos una cantidad infinita de coeficientes para implementarlo. En la práctica esto no es posible, ya que un filtro digital debe estar compuesto por una cantidad finita de muestras (más adelante, cuando estudiemos los filtros IIR y las ecuaciones recursivas, veremos cómo es posible obtener respuestas que, aunque potencialmente infinitas, pueden implementarse utilizando un número finito de coeficientes).

Por esta razón aparece la idea de la ventana. En lugar de utilizar toda la respuesta al impulso ideal $h_d[n]$, tomamos solamente una parte de ella. Para lograrlo, multiplicamos $h_d[n]$ por una función de duración finita denominada ventana, representada por $w[n]$. El resultado será una nueva respuesta al impulso, también de duración finita, que denominaremos $h_w[n]$.

En el caso de un filtro pasa bajo, se tiene:

$$h_w[n] = h_d[n-\alpha]\, w[n]$$

donde

$$h_d[n] = \begin{cases} \dfrac{\omega_c}{\pi}, & n = 0 \\ \dfrac{\sin(\omega_c n)}{\pi n}, & n \neq 0 \end{cases}$$

y

$$\alpha = \frac{M-1}{2}$$ siendo M la longitud de la ventana.

Podemos observar que $h_d[n]$ corresponde a la respuesta al impulso ideal, la cual se extiende desde $-\infty$ hasta $+\infty$. Al multiplicarla por una ventana de longitud finita, obtenemos $h_w[n]$, una **versión truncada** que sí puede implementarse como un filtro FIR.

Pero el efecto de multiplicar $h_d[n]$ por una ventana $w[n]$ no es gratuito. Recordemos que una multiplicación en el dominio del tiempo equivale a una convolución en frecuencia:

$$H_w(e^{j\omega}) = H_d(e^{j\omega}) \otimes W(e^{j\omega})$$

Por lo tanto, la respuesta ideal ya no se conservará exactamente. La ventana suavizará las discontinuidades del filtro ideal, generando una banda de transición y ondulaciones cuya magnitud dependerá de la ventana utilizada. Es decir, una ventana distinta producirá una respuesta en frecuencia distinta (pero, conservando el tipo de filtro).

La siguiente figura resume el procedimiento de diseño de filtros FIR mediante el método de ventaneo. Se muestra tanto el proceso de truncamiento

de la respuesta al impulso ideal en el dominio del tiempo como sus consecuencias en el dominio de la frecuencia, donde aparecen una banda de transición y ondulaciones determinadas por la ventana utilizada.

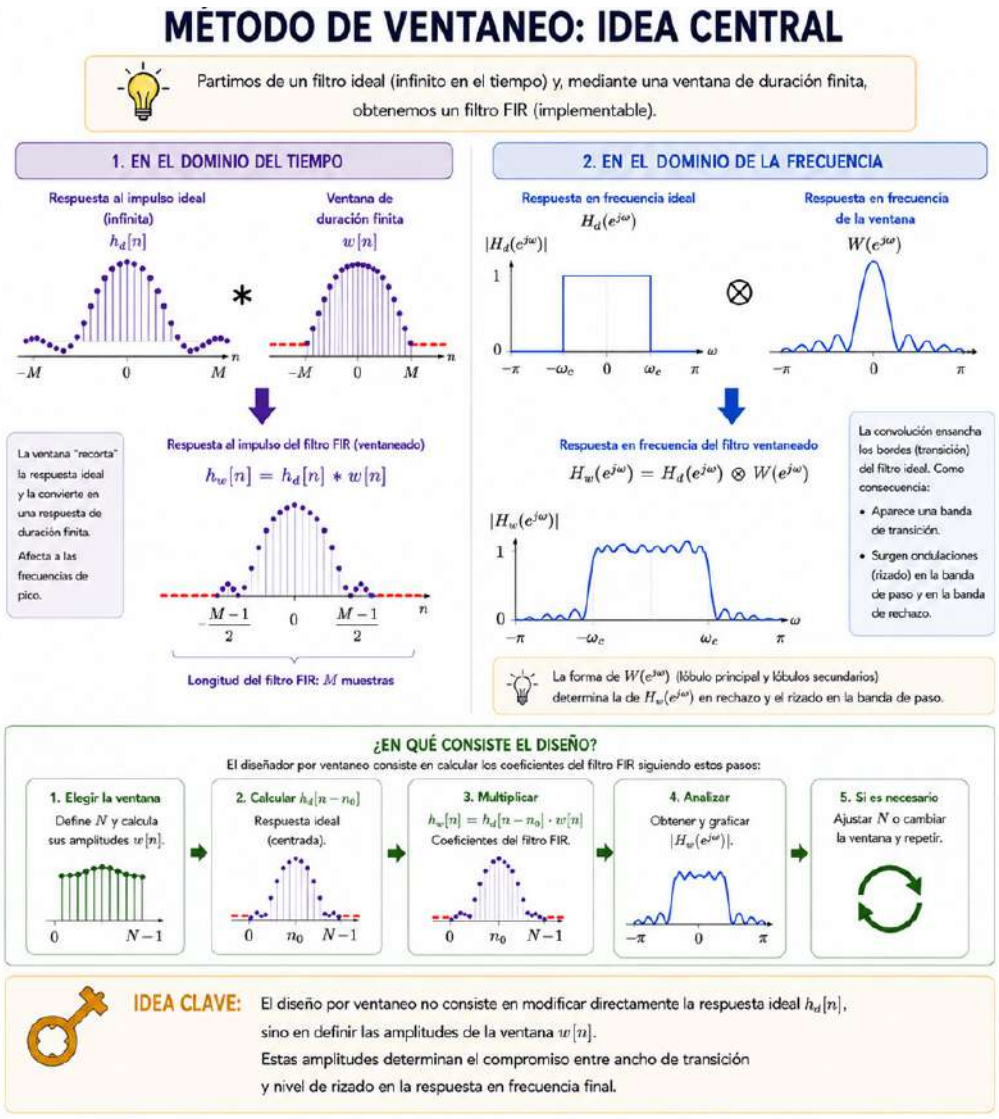


Figura 14. Método de ventaneo para el diseño de filtros FIR. El operador * corresponde a multiplicación; mientras que el operador ⊗ corresponde a convolución.

Pero ¿cuáles son las ventanas más comúnmente utilizadas para truncar la respuesta al impulso del filtro ideal?

A lo largo de los años se han propuesto numerosas ventanas, cada una con características particulares en términos de ancho del lóbulo principal y atenuación de los lóbulos secundarios. Sin embargo, en aplicaciones de procesamiento digital de señales, algunas de las más utilizadas son: ventana rectangular, Hanning (Hann), Hamming y Blackman.

La diferencia entre ellas radica en la forma de sus amplitudes $w[n]$. Mientras la ventana rectangular presenta amplitud constante, las demás ventanas suavizan progresivamente los extremos. Y este “pequeño” cambio tiene un impacto en la respuesta en frecuencia del filtro resultante. Por esta razón, la selección de la ventana adecuada implica encontrar un compromiso entre selectividad (ancho del lóbulo principal) y atenuación (en la banda de rechazo).

La Figura 15 compara las ventanas más utilizadas y muestra cómo sus características temporales se reflejan directamente en el comportamiento espectral del filtro resultante.

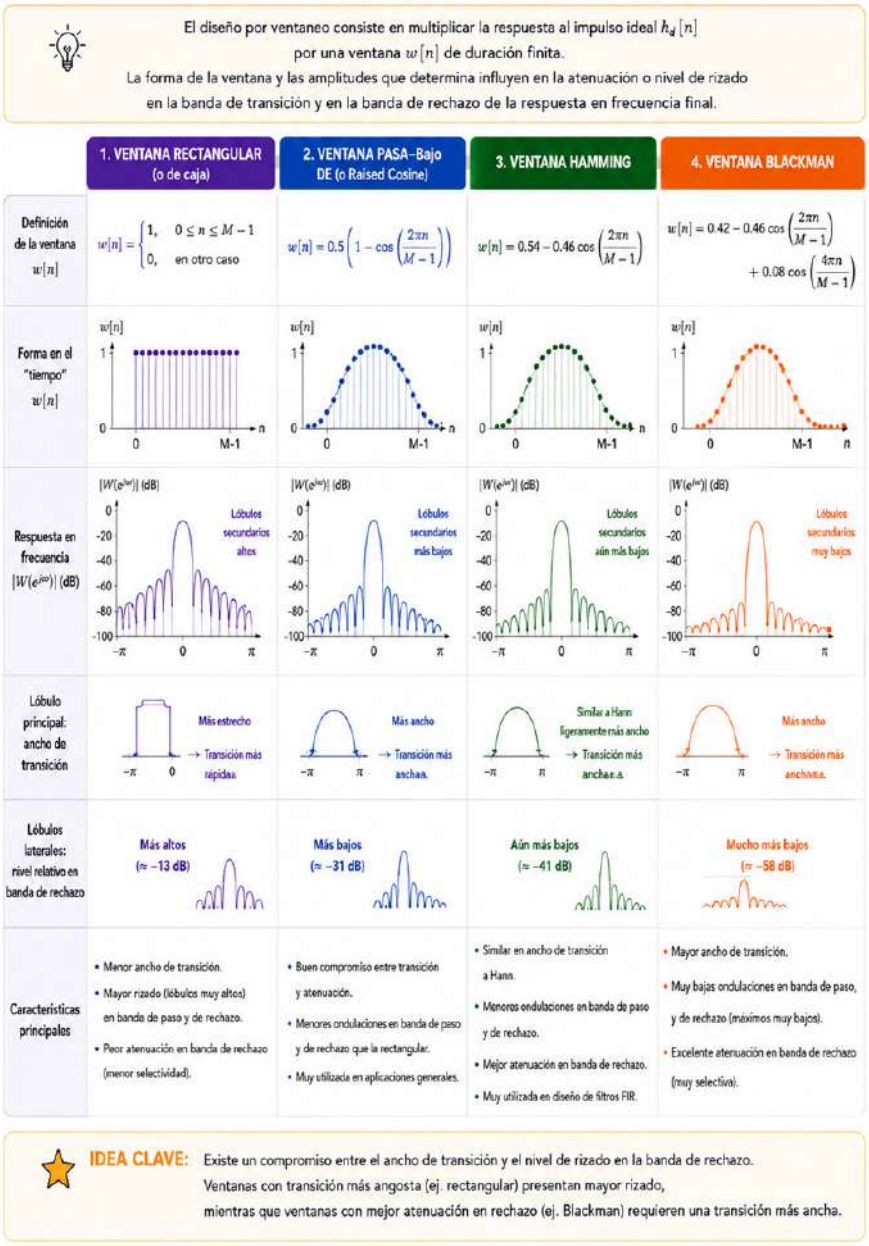


Figura 15. Comparación de cuatro ventanas en el diseño de filtros FIR. Se presentan ordenadas de **menor** a **mayor atenuación** en banda de rechazo. En general, a mayor atenuación de los lóbulos secundarios, mayor es el ensanchamiento del lóbulo principal.

De la figura anterior podemos apreciar que cada ventana posee una ecuación específica para el cálculo de sus amplitudes, las cuales dependen del valor de M seleccionado. Sin embargo, más importante que memorizar estas ecuaciones es comprender qué significa diseñar un filtro mediante ventaneo y cómo la elección de la ventana influye en la respuesta en frecuencia obtenida.

Ejemplo 5. Diseño de filtro FIR por método de ventaneo, apoyado en Python

Una vez comprendidos los principios del método de ventaneo, podemos apoyarnos en Python para calcular automáticamente las amplitudes de las ventanas más utilizadas. En este ejemplo se generarán las ventanas: rectangular, Hanning, Hamming y Blackman, para una longitud de $M = 15$ muestras. Esto nos permitirá visualizar sus diferencias y comprender cómo estas influyen posteriormente en el diseño del filtro FIR.

```python
import numpy as np
import matplotlib.pyplot as plt

# Longitud de la ventana
M = 15
n = np.arange(M)

# Ventanas
w_rect = np.ones(M)
w_hann = np.hanning(M)
w_hamm = np.hamming(M)
w_black = np.blackman(M)

# Figura 2x2
fig, ax = plt.subplots(2, 2, figsize=(10, 6))

# Rectangular
ax[0,0].stem(n, w_rect); ax[0,0].set_title('Ventana Rectangular')
ax[0,0].set_xlabel('n'); ax[0,0].set_ylabel(r'$w[n]$')
ax[0,0].grid(True)

# Hanning
ax[0,1].stem(n, w_hann); ax[0,1].set_title('Ventana Hanning')
ax[0,1].set_xlabel('n'); ax[0,1].set_ylabel(r'$w[n]$')
ax[0,1].grid(True)

# Hamming
ax[1,0].stem(n, w_hamm); ax[1,0].set_title('Ventana Hamming')
ax[1,0].set_xlabel('n'); ax[1,0].set_ylabel(r'$w[n]$')
ax[1,0].grid(True)

# Blackman
ax[1,1].stem(n, w_black); ax[1,1].set_title('Ventana Blackman')
ax[1,1].set_xlabel('n'); ax[1,1].set_ylabel(r'$w[n]$')
ax[1,1].grid(True)

plt.tight_layout(); plt.show()
```

Figura 16. Código en Python del Ejemplo 5, diseño de ventanas: rectangular, Hanning, Hamming y Blackman. $M = 15$ muestras.

Obteniendo,

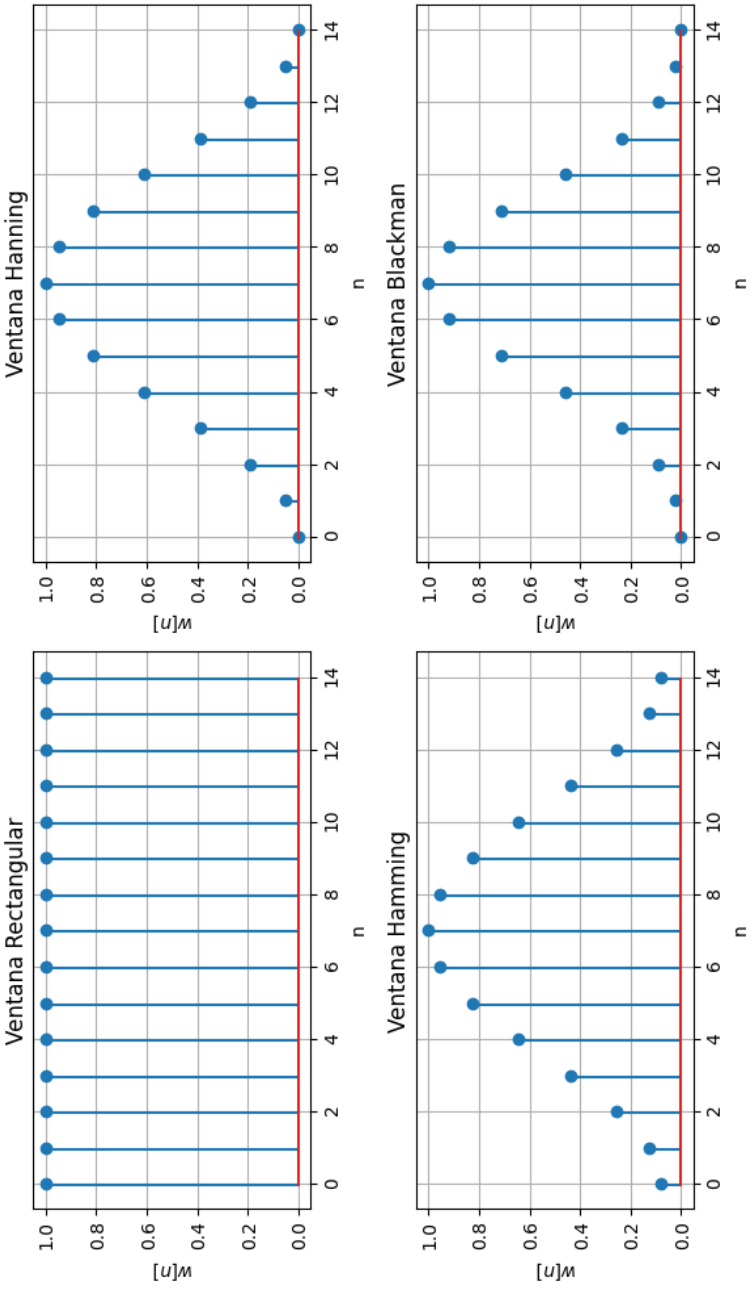


Figura 17. Ventanas del Ejemplo 5: rectangular, Hanning, Hamming y Blackman. $M = 15$ muestras.

Ahora, vamos a multiplicar estas ventanas por la respuesta al impulso de un filtro pasa-bajo idea, para obtener $h_w[n]$.

```
import numpy as np
import matplotlib.pyplot as plt


M = 15 # longitud de la ventana
wc = np.pi / 4 # frecuencia de corte en el rango [0 pi]
n = np.arange(M) # indices
alpha = (M - 1) / 2 # centro de simetría

# Respuesta al impulso ideal (pasa-bajo)
hd = np.zeros(M)

for i in range(M):
    if n[i] == alpha:
        hd[i] = wc / np.pi
    else:
        hd[i] = np.sin(wc * (n[i] - alpha)) / \
                (np.pi * (n[i] - alpha))
# Ventanas
w_rect = np.ones(M)
w_hann = np.hanning(M)
w_hamm = np.hamming(M)
w_black = np.blackman(M)

# Respuestas al impulso del filtro ideal, multiplicado por las ventanas
hw_rect = hd * w_rect
hw_hann = hd * w_hann
hw_hamm = hd * w_hamm
hw_black = hd * w_black

# Gráficas
fig, ax = plt.subplots(2, 2, figsize=(10, 6),
                       sharex=True, sharey=True)

ventanas = [hw_rect, hw_hann, hw_hamm, hw_black]
titulos = ['Rectangular', 'Hanning','Hamming','Blackman']

for eje, hw, titulo in zip(ax.ravel(), ventanas, titulos):

    eje.stem(n, hw); eje.set_title(titulo)
    eje.set_xlabel('n'); eje.set_ylabel(r'$h_w[n]$')
    eje.grid(True)

plt.tight_layout()
plt.show()
```

Figura 18. Código en Python del Ejemplo 5 para la respuesta al impulso de filtro FIR diseñado por método de ventaneo, para cuatro ventanas: rectangular, Hanning, Hamming y Blackman, con $M = 15$ muestras y $\omega_{cd} = \pi/4$.

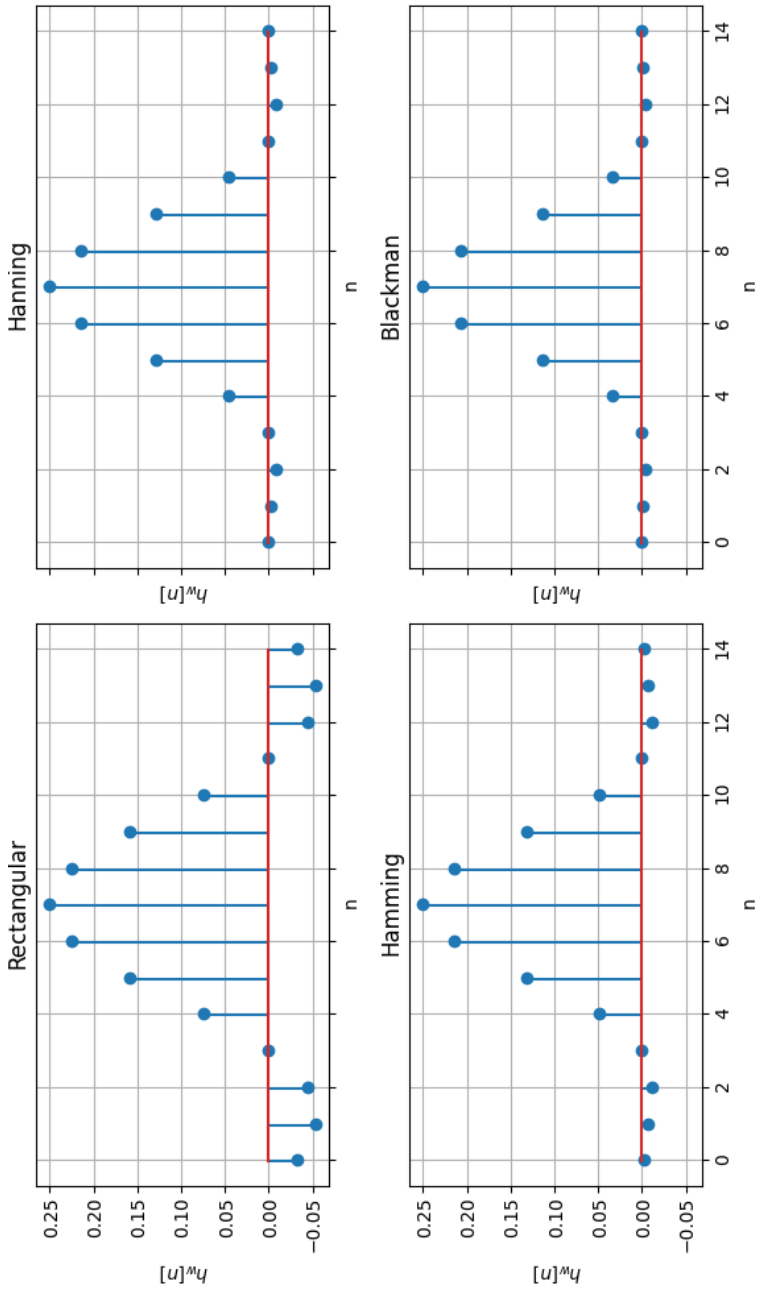


Figura 19. Respuesta al impulso del filtro FIR diseñado por método de ventaneo (Ejemplo 5), para cuatro ventanas: rectangular, Hanning, Hamming y Blackman, con $M = 15$ muestras y $\omega_{cd} = \pi/4$.

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

# Respuestas en frecuencia
omega, H_rect = freqz(hw_rect, worN=2048)
_, H_hann = freqz(hw_hann, worN=2048)
_, H_hamm = freqz(hw_hamm, worN=2048)
_, H_black = freqz(hw_black, worN=2048)

# Gráfica comparativa
plt.figure(figsize=(10,6))

plt.plot(omega, 20*np.log10(np.abs(H_rect)+1e-10),
         label='Rectangular')

plt.plot(omega, 20*np.log10(np.abs(H_hann)+1e-10),
         label='Hanning')

plt.plot(omega, 20*np.log10(np.abs(H_hamm)+1e-10),
         label='Hamming')

plt.plot(omega, 20*np.log10(np.abs(H_black)+1e-10),
         label='Blackman')

plt.axvline(wc,
            linestyle='--',
            label=r'$\omega_c=\pi/4$')

plt.xlim([0, np.pi])
plt.ylim([-100, 10])

plt.xlabel(r'Frecuencia digital $\omega$ [rad/muestra]')
plt.ylabel('Magnitud [dB]')
plt.title('Respuesta en frecuencia de filtros FIR diseñados mediante ventaneo')
plt.grid(True)
plt.legend()

plt.show()
```

Figura 20. Código en Python del Ejemplo 5 para la respuesta en frecuencia de filtro FIR diseñado por método de ventaneo, para cuatro ventanas: rectangular, Hanning, Hamming y Blackman, con $M = 15$ muestras y $\omega_{cd} = \pi/4$.

Obteniendo,

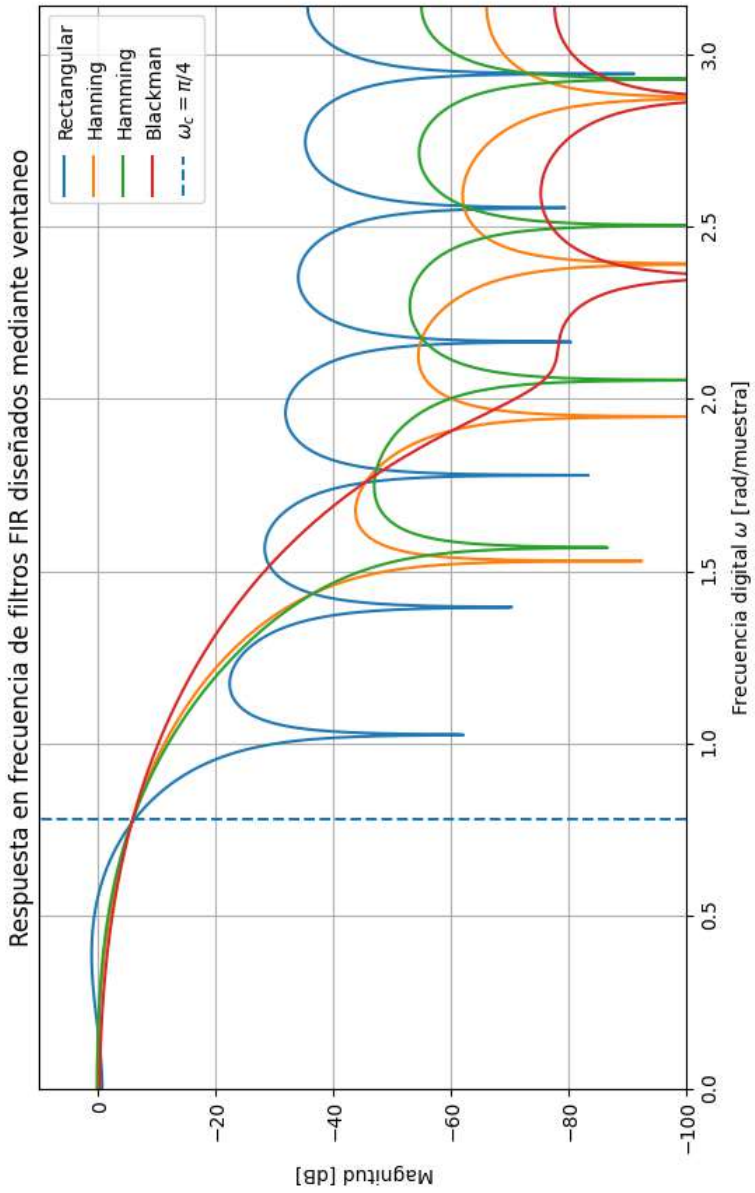


Figura 21. Respuesta en frecuencia de filtro FIR del Ejemplo 5 diseñado por método de ventaneo, para cuatro ventanas: rectangular, Hanning, Hamming y Blackman, con $M = 15$ muestras y $\omega_{cd} = \pi/4$.

De la figura anterior, quiero que nos enfocamos en lo siguiente:

- Todos los filtros diseñados tienen la misma frecuencia de corte, correspondiente a $\omega_{cd} = \pi/4$, es decir, 0.785rad/muestra.
- El filtro diseñado con ventana **Blackman** presenta la mayor atenuación en la banda de rechazo, mientras que el filtro diseñado con ventana **rectangular** presenta la menor atenuación.
- Las respuestas en frecuencia de los filtros diseñados a partir de las ventanas **Hanning** y **Hamming** son muy similares entre sí.
- El filtro con la **banda de transición más estrecha** corresponde a la ventana rectangular, mientras que el filtro con la **banda de transición más amplia** corresponde a la ventana Blackman.
-

Ejemplo 6. Efecto de la longitud de la ventana

En el ejemplo anterior diseñados cuatro filtros FIR utilizando cuatro ventanas diferentes, pero ¿qué ocurre si dejamos fijos el tipo de ventana y frecuencia de corte, y únicamente modificamos el valor de M?

Intuitivamente, una ventana más larga conserva una mayor porción de la respuesta al impulso ideal $h_d[n]$, por lo que el filtro obtenido se aproxima mejor al comportamiento deseado. Por el contrario, una ventana más corta produce una aproximación más limitada y, en consecuencia, una respuesta en frecuencia más alejada de la ideal.

Para ilustrar este efecto, utilizaremos la ventana Blackman y compararemos las respuestas en frecuencia obtenidas para diferentes valores de M.

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

wc = np.pi / 4 # Frecuencia de corte
M_values = [15, 31, 63, 127] # Valores de M
fig, ax = plt.subplots(2, 2, figsize=(10, 7))

for eje, M in zip(ax.ravel(), M_values):
    n = np.arange(M)
    alpha = (M - 1) / 2
    # Respuesta al impulso ideal
    hd = np.zeros(M)

    for i in range(M):
        if n[i] == alpha:
            hd[i] = wc / np.pi
        else:
            hd[i] = np.sin(wc * (n[i] - alpha)) / \
                    (np.pi * (n[i] - alpha))

    w = np.blackman(M) # Ventana Blackman
    hw = hd * w  # Respuesta al impulso ventaneada
    omega, H = freqz(hw, worN=2048) # Respuesta en frecuencia

    eje.plot(omega, 20 * np.log10(np.abs(H) + 1e-10))
    eje.axvline(wc, linestyle='--')
    eje.set_title(f'M = {M}'); eje.set_xlabel(r'$\omega$ [rad/muestra]')
    eje.set_ylabel('Magnitud [dB]'); eje.set_xlim([0, np.pi])
    eje.set_ylim([-140, 10]); eje.grid(True)

plt.tight_layout()
plt.show()
```

Figura 22. Código en Python del Ejemplo 6: comparación respuesta en frecuencia filtro FIR diseñado con ventana Blackman, con $M = 15, 31, 63$ y 127 muestras, y $\omega_{cd} = \pi/4$.

Obteniendo las siguientes respuestas en frecuencia:

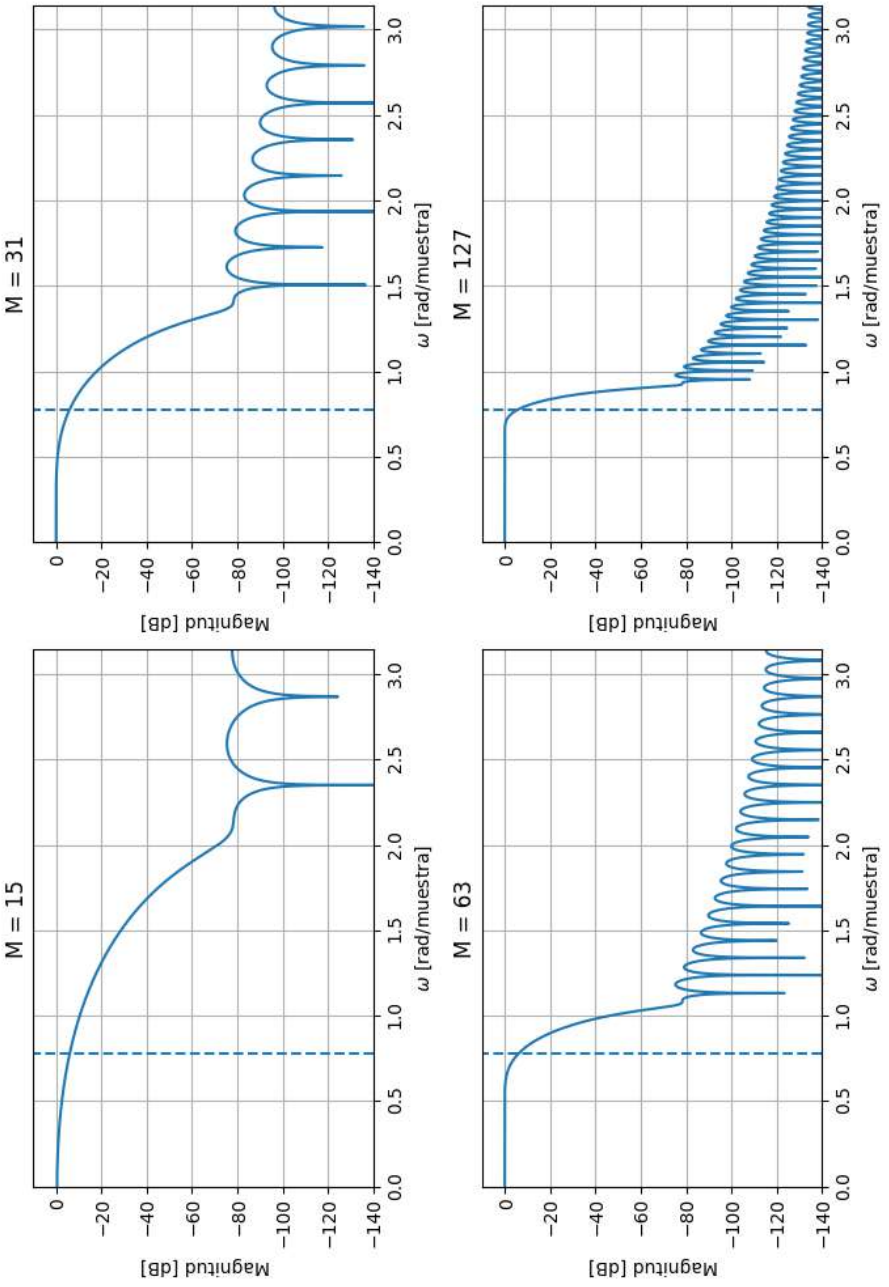


Figura 23. Respuesta en frecuencia filtro FIR diseñado con ventana Blackman (Ejemplo 6), con $M = 15, 31, 63$ y 127 muestras y $\omega_{cd} = \pi/4$.

¿Qué observamos en las cuatro respuestas en frecuencia anteriores?

- En todos los casos, la ganancia en la frecuencia de corte es aproximadamente -6dB, una característica propia de los filtros FIR diseñados mediante el método de ventaneo.
- Sin embargo, la principal diferencia aparece en la banda de transición. A medida que aumenta la longitud M, la transición entre la banda de paso y la banda de rechazo se vuelve más abrupta, haciendo que la atenuación deseada se alcance en frecuencias cada vez más próximas a la frecuencia de corte.

Es importante resaltar que, a diferencia de los filtros de promedio, en los filtros diseñados mediante el método de ventaneo el parámetro M está directamente relacionado con el orden del filtro, dado por $M - 1$. Por esta razón, incrementar M equivale a aumentar el orden del filtro, permitiendo una mejor aproximación a la respuesta en frecuencia especificada por el diseñador. No obstante, esta mejora se obtiene a costa de un mayor número de coeficientes y, por tanto, de una mayor carga computacional.

1.5.4. Filtros FIR: filtro de promedio vs ventaneo

Ya hemos visto dos opciones de diseño de filtros FIR: el filtro de promedio y el método de ventaneo. En el primero, el diseño de las amplitudes en $h[n]$ es muy sencillo, dado que todas son iguales. En el segundo, existe una fórmula matemática por tipo de ventana para calcular sus amplitudes, las cuales difieren entre sí.

Pero, las diferencias van mucho más allá. Para ello, he creado la siguiente figura en la cual comparo ambos métodos de diseño en términos de la tipología de filtro que se puede diseñar, qué representa M y cuál es su interpretación práctica, forma típica de $h[n]$ y un resumen de cierre conceptual.

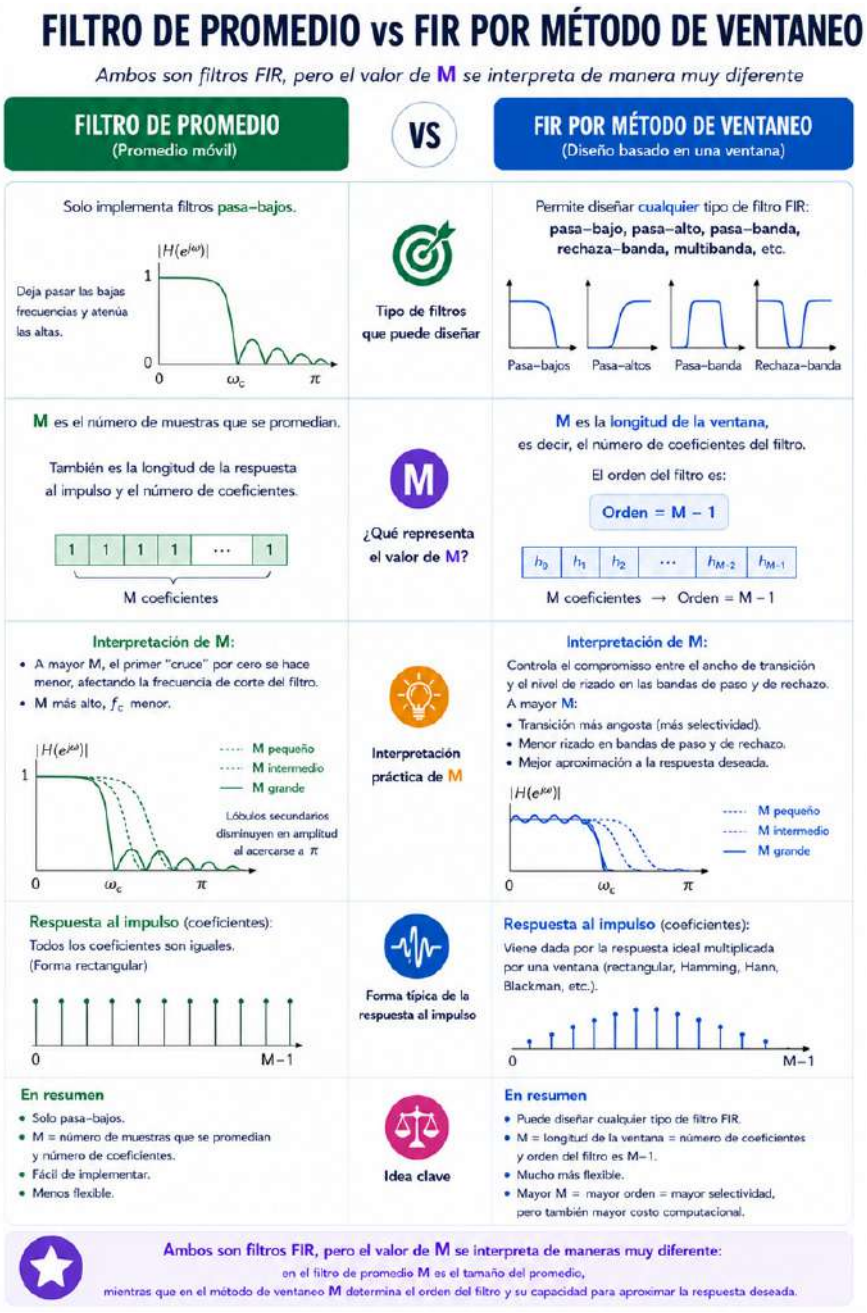


Figura 24. Comparación de filtros FIR diseñado por: promedio (izquierda), método de ventaneo (derecha).

Aspectos para resaltar:

- El filtro de promedio únicamente permite implementar filtros pasa-bajos; mientras que con el método de ventaneo se puede diseñar cualquier tipo de filtro (ej. Pasa-altos).
- En el caso del filtro de promedio, el valor de M controla la posición del primer cruce por cero de la respuesta en frecuencia y, en consecuencia, su frecuencia de corte efectiva. A medida que M aumenta, dicho cruce se acerca al origen y el filtro se vuelve más selectivo. En cambio, con el método de ventaneo, la frecuencia de corte no depende de M, sino de la seleccionada por el usuario en el rango $[0 \quad \pi]$. No obstante, el valor de M sí tiene un efecto, pero en la pendiente entre la banda de paso y la de rechazo, a mayor M, mayor pendiente.
- La tercera diferencia importante es la forma de la respuesta al impulso h[n]. En el filtro de promedio todos los coeficientes tienen la misma amplitud, mientras que en el método de ventaneo las amplitudes varían de acuerdo con la ventana seleccionada.

En otras palabras, aunque ambos métodos producen filtros FIR, responden a filosofías de diseño diferentes. El filtro de promedio apuesta por la simplicidad y la facilidad de implementación, mientras que el método de ventaneo ofrece una mayor flexibilidad para aproximar respuestas en frecuencia específicas y diseñar filtros más selectivos.

1.6. Filtros IIR (Infinite Impulse Response)

Hasta ahora hemos estudiado filtros FIR, cuya principal característica es que su respuesta al impulso $h[n]$ tiene una duración finita. Esto significa que, después de un número determinado de muestras, la respuesta del sistema desaparece completamente.

Pero no todos los filtros digitales se comportan de esta manera.

En esta siguiente sección estudiaremos un tipo de filtro diferente desde la perspectiva de $h[n]$: los filtros IIR. En estos sistemas, la respuesta al impulso puede extenderse indefinidamente en el tiempo, razón por la cual reciben el nombre de respuesta al impulso infinita (*Infinite Impulse Response*).

A continuación, conoceremos el concepto de Transforma Bilineal y su papel en el diseño de filtros IIR, y cómo podemos apoyarnos en Python para el diseño de este tipo de filtros.

1.6.1. Transformada Bilineal

Hasta este punto hemos diseñado filtros FIR definiendo directamente los coeficientes de la respuesta al impulso $h[n]$. Sin embargo, existe otra estrategia muy utilizada en procesamiento digital de señales: partir de un filtro análogo ya conocido y transformarlo en un filtro digital equivalente.

Esta idea resulta especialmente interesante porque el diseño de filtros análogos posee una larga tradición en ingeniería. Durante décadas se desarrollaron familias de filtros como *Butterworth*, *Chebyshev* y *Elípticos*, cuyas características son ampliamente conocidas y estudiadas.

La pregunta entonces es: ¿cómo podemos aprovechar ese conocimiento en el mundo digital?

La respuesta consiste en establecer una relación matemática entre el dominio de Laplace, representado por la variable s, y el dominio Z, representado por la variable z. De esta manera, a partir de una función de transferencia análoga $H(s)$, es posible obtener una función de transferencia digital $H(z)$ con características similares.

Uno de los métodos más utilizados para realizar esta transformación es la **Transformada Bilineal**, la cual emplea una *ecuación de mapeo* que relaciona ambos dominios y permite trasladar polos y ceros desde el plano s hacia el plano z, dada por:

$$s = \frac{2}{T}\frac{1-z^{-1}}{1+z^{-1}}$$

donde T es el periodo de muestreo (es decir $1/f_s$). Así, en cada una de las partes de $H(s)$ donde aparezca *s*, se reemplaza por su equivalente en z.

A continuación, tenemos una gráfica que nos permite resumir los pasos para el diseño de filtros IIR utilizando la Transformada Bilineal:

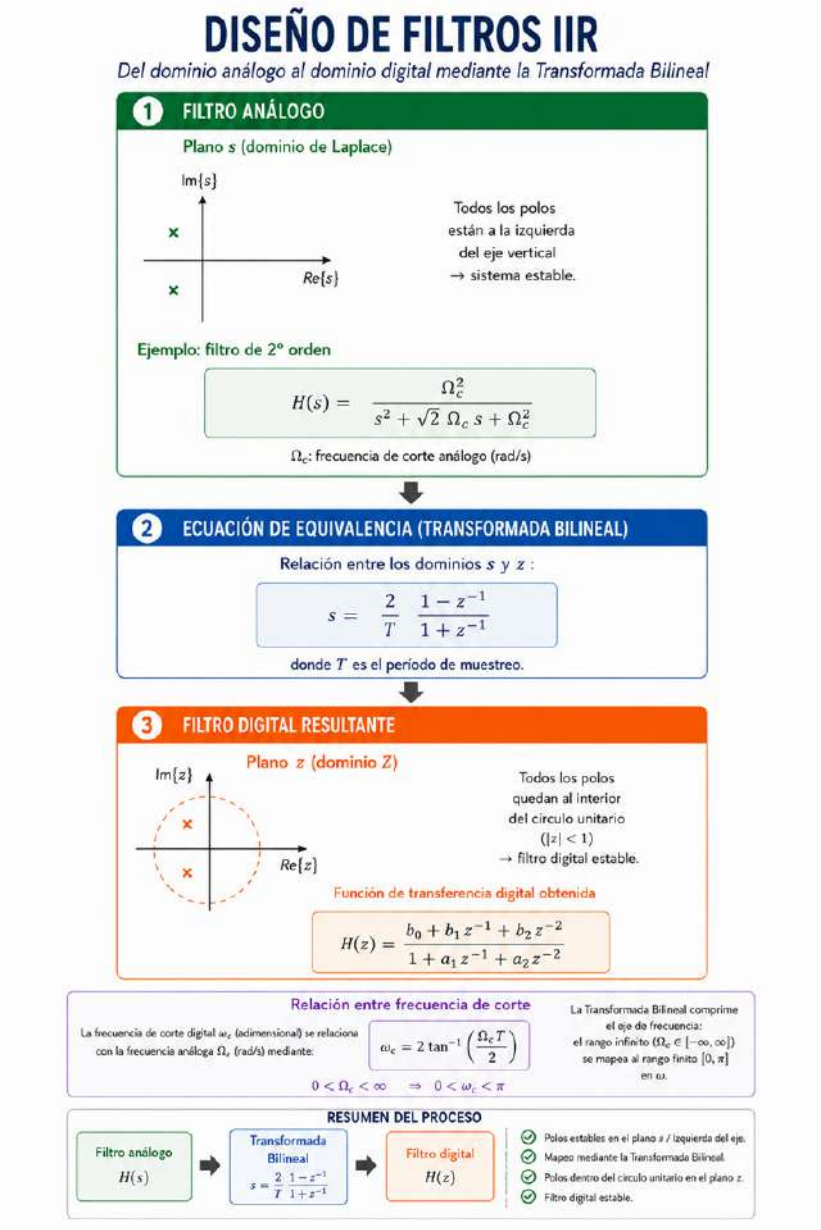


Figura 25. Proceso general de diseño de filtros IIR mediante Transformada Bilineal, a partir de una función de transferencia análoga $H(s)$ y su correspondiente mapeo al dominio digital $H(z)$.

Es importante enfatizar que, si partimos de un filtro análogo estable, obtendremos un filtro digital estable, y del mismo tipo en términos de su respuesta en frecuencia. No obstante, aunque el rango de frecuencias del filtro análogo se extiende entre $0 \leq \Omega_c < \infty$, el filtro digital correspondiente queda limitado al intervalo $0 \leq \omega_c < \pi$.

La Transformada Bilineal preserva la estabilidad y la naturaleza del filtro (pasa-bajo, pasa-altos, etc.), pero comprime el eje infinito de frecuencias analógicas al rango finito de frecuencias digitales $[0, \pi)$.

1.6.2. Diseño de filtros Butterworth mediante Transformada Bilineal

Dentro de los filtros análogos, el filtro Butterworth tiene unas características especiales que vale la pena mencionar:

- **No presenta rizado en la banda de paso**, por lo que la ganancia varía de forma suave y continua dentro del rango de frecuencias que se desea conservar.
- Presenta una **respuesta máximamente plana en la banda de paso**, característica que le da una transición suave entre la banda de paso y la banda de rechazo.
- Sus polos se encuentran distribuidos en el semiplano izquierdo del plano s, garantizando la **estabilidad del filtro análogo**.
- No posee ceros finitos en $H(s)$. Por esta razón, su comportamiento está determinado exclusivamente por la ubicación de sus **polos**.
- A medida que aumenta el orden del filtro, la pendiente de la banda de transición se incrementa, produciendo una separación más pronunciada entre la banda de paso y la banda de rechazo.

En el siguiente ejemplo diseñaremos filtros Butterworth pasa-bajos de segundo, tercer y cuarto orden. Posteriormente, aplicaremos la Transformada Bilineal para obtener sus versiones digitales y analizar el efecto del orden sobre la respuesta en frecuencia resultante.

Ejemplo 7. Diseño de filtro Butterworth pasa-bajo mediante Transformada Bilineal, apoyado en Python

A continuación, diseñaremos tres filtros pasa-bajos Butterworth. Todos comparten la misma frecuencia de corte del filtro análogo, pero difieren en el orden del filtro. El objetivo es observar cómo el orden modifica la pendiente de la banda de transición, manteniendo la misma frecuencia de corte.

Inicialmente, definimos los siguientes parámetros de diseño:

- Frecuencia de corte del filtro análogo, f_c, expresada en [Hz].
- Frecuencia de muestreo, f_s, la cual debe ser al menos dos veces superior a f_c, de acuerdo con el criterio de Nyquist.
- Frecuencia de corte del filtro análogo, Ω_c, expresada en [rad/s] y calculada a partir de f_c.
- Órdenes de los filtros a diseñar.

Los coeficientes de la función de transferencia del filtro análogo $H(s)$, tanto del numerador (b_s) como del denominador (a_s), se calculan mediante la instrucción **butter** de la librería **SciPy**. Posteriormente, se aplica la **Transformada Bilineal** a partir de b_s y a_s, obteniendo los coeficientes de la función de transferencia digital $H(z)$, tanto del numerador (b_z) como del denominador (a_z).

Finalmente, la respuesta en frecuencia del filtro digital se calcula mediante la instrucción **freqz** y se representa en decibelios (dB), permitiendo analizar el efecto que tiene el orden del filtro sobre la selectividad de la respuesta obtenida.

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

fs = 8000                 # Frecuencia de muestreo [Hz]
fc = 1000                 # Frecuencia de corte filtro análogo en [Hz]
Omega_c = 2*np.pi*fc      # frecuencia de corte filtro análogo en rad/s
wc = 2*np.arctan(Omega_c/(2*fs)) # Frecuencia de corte digital en rad/muestra
ordenes = [2, 3, 4] # orden del filtro

plt.figure(figsize=(8, 5))

for N in ordenes:
    # 1. Diseño del filtro análogo Butterworth
    b_s, a_s = signal.butter(N, Omega_c, btype='low', analog=True)

    # 2. Transformada Bilineal: H(s) -> H(z)
    b_z, a_z = signal.bilinear(b_s, a_s, fs=fs)

    # 3. Respuesta en frecuencia digital
    w, H = signal.freqz(b_z, a_z, worN=2048)

    # Gráfica en dB
    H_dB = 20 * np.log10(np.abs(H) + 1e-12)
    plt.plot(w, H_dB, label=f'Orden {N}')

# Se calcula la frecuencia de corte experimental usando una de las respuestas,
# ya que todos los filtros cruzan aproximadamente en el mismo valor de wc.
idx_wc = np.argmin(np.abs(H_dB + 3))

# Frecuencia de corte experimental
wc_exp = w[idx_wc]

plt.axvline(wc, linestyle='--', label=r'$\omega_c$')
plt.axhline(-3, linestyle=':', label='-3 dB')

plt.title('Filtros IIR pasa-bajos mediante Transformada Bilineal')
plt.xlabel(r'Frecuencia digital $\omega$ [rad/muestra]')
plt.ylabel(r'Magnitud $|H(e^{j\omega})|$ [dB]')
plt.xlim(0, np.pi); plt.ylim(-80, 5)
plt.grid(True); plt.legend(); plt.show()

print(f"wc = {wc:.4f} rad/muestra")
print(f"wc_normalizada = {wc/np.pi:.4f} normalizada en [0 1]")

print(f"wc_experimental = {wc_exp:.4f} rad/muestra")
print(f"wc_experimental_normalizada = {wc_exp/np.pi:.4f} normalizada en [0 1]")
```

Figura 26. Código en Python del Ejemplo 7: filtro pasa-bajo Butterworth utilizando Transformada Bilineal: fc=1000 Hz, fs = 8 KHz, N=2, 3, 4.

Obteniendo:

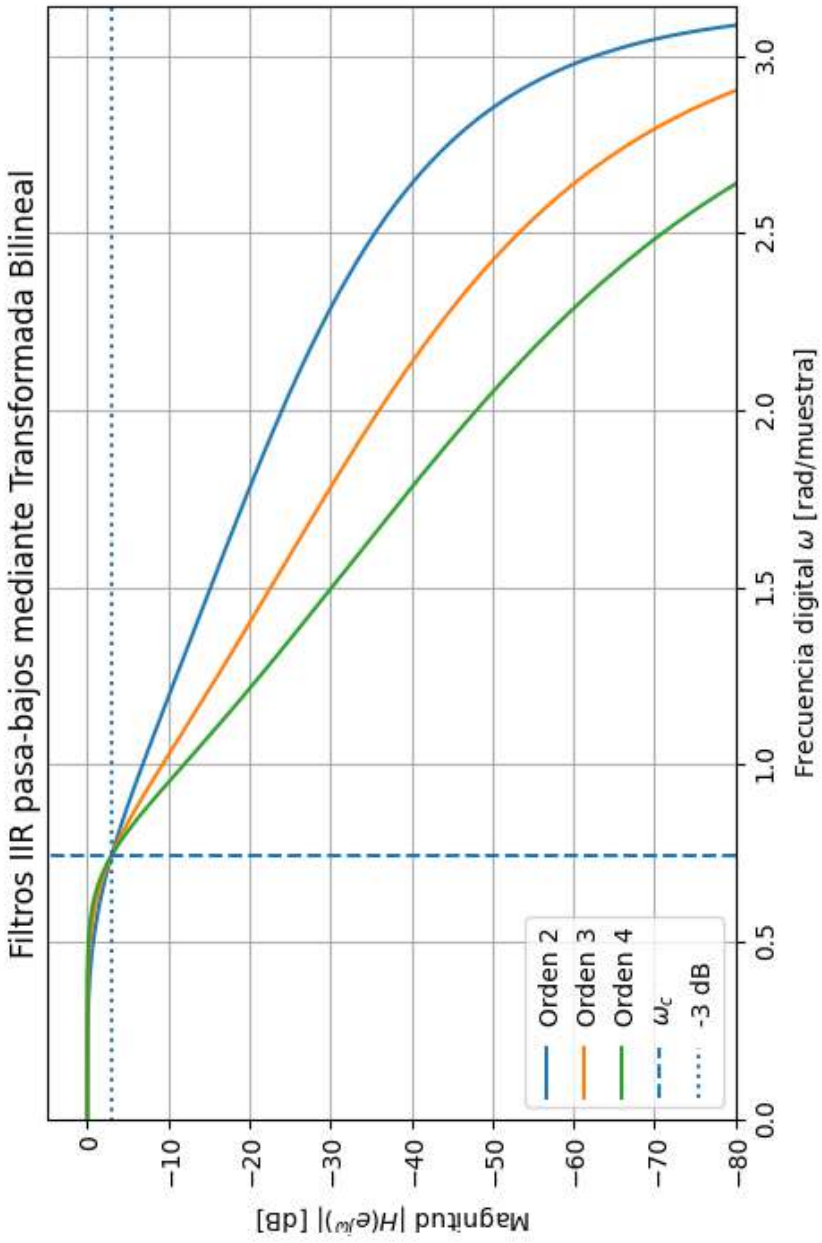


Figura 27. Respuesta en frecuencia del Ejemplo 7: filtro pasa-bajo Butterworth utilizando Transformada Bilineal: fc=1000 Hz, fs = 8 KHz, N=2, 3, 4.

Podemos observar que, a medida que aumenta el orden del filtro N, la respuesta en frecuencia presenta una transición más abrupta entre la banda de paso y la banda de rechazo. Sin embargo, todos los filtros cruzan aproximadamente en el mismo valor de frecuencia de corte digital ω_c.

Adicionalmente, los valores teórico y experimental de ω_c son:

$\omega_c = 0.7484$ rad/muestra

$\omega_{c_normalizada} = 0.2382$

$\omega_{c_experimental} = 0.7486$ rad/muestra

$\omega_{c_experimental\ normalizada} = 0.2383$

Podemos comprobar que existe una excelente concordancia entre el valor teórico, obtenido mediante la ecuación de mapeo que relaciona Ω_c con ω_c, y el valor experimental estimado a partir de la respuesta en frecuencia del filtro digital. Esta coincidencia confirma la validez del proceso de diseño basado en la Transformada Bilineal.

Ejemplo 8. Diseño de filtro Butterworth pasa-alto mediante Transformada Bilineal, apoyado en Python

Vamos ahora a cambiar el tipo de filtro. En lugar de diseñar un filtro pasa-bajos, diseñaremos un filtro pasa-altos. Para ello, en la función signal.butter() se modifica el parámetro correspondiente al tipo de filtro, usando: btype='high'.

De esta manera, el filtro análogo de partida ya no será pasa-bajos, sino pasa-altos. Posteriormente, mediante la Transformada Bilineal, se obtiene su versión digital.

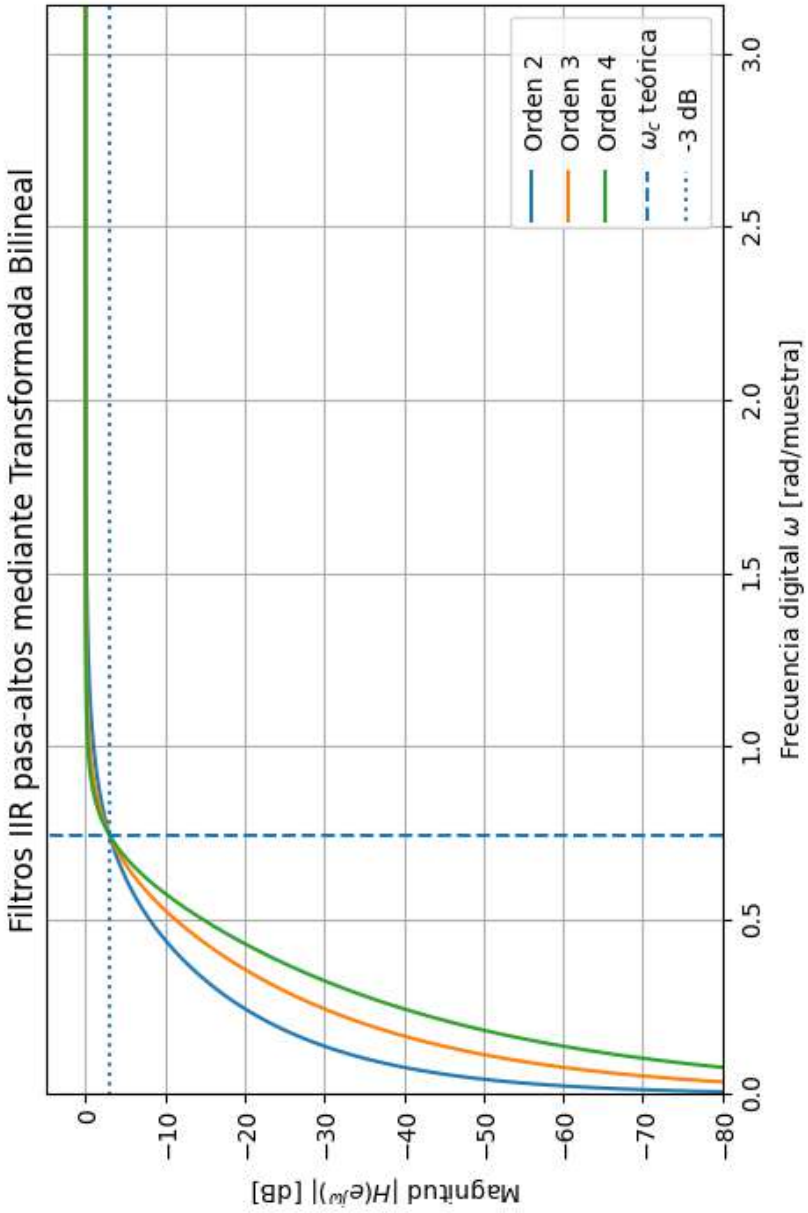


Figura 28. Respuesta en frecuencia del Ejemplo 8: filtro pasa-altos Butterworth utilizando Transformada Bilineal: $f_c = 1000$Hz, $f_s = 8$kHz, $N = 2,3,4$.

Los valores de frecuencia de corte del filtro digital son los mismos del ejemplo anterior, ya que se conservaron los mismos valores de f_c y f_s. Sin embargo, la forma de la respuesta cambia: ahora el filtro atenúa las bajas frecuencias y permite el paso de las componentes de alta frecuencia.

Pero ¿cómo sabemos cuál es la función de transferencia de la que partimos, es decir, H(s) ?, y ¿cuál es la función de transferencia obtenida para el filtro digital, H(z)?

La respuesta está en los coeficientes calculados por Python. Los vectores b_s y a_s contienen los coeficientes del numerador y denominador de $H(s)$, respectivamente. Por su parte, los vectores b_z y a_z contienen los coeficientes del numerador y denominador de $H(z)$.

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

# Coeficientes de H(s)
print(f"coeficientes del numerador de H(s) = {np.array2string(b_s,
                                                    precision=4, suppress_small=True)} ")
print(f"coeficientes del denominador de H(s) = {np.array2string(a_s,
                                                     precision=4, suppress_small=True)}")

# Coeficientes de H(z)
print(f"coeficientes del numerador de H(z) = {np.array2string(b_z,
                                                    precision=4, suppress_small=True)} ")
print(f"coeficientes del denominador de H(z) = {np.array2string(a_z, precision=4,
                                                     suppress_small=True)}")
```

Figura 29. Código en Python del Ejemplo 8: coeficientes de $H(s)$ y $H(z)$ del filtro pasa-altos Butterworth.

```
coeficientes del numerador de H(s) = [1. 0. 0. 0. 0.]
coeficientes del denominador de H(s) = [1.0000e+00 1.6419e+04
1.3479e+08 6.4819e+11 1.5585e+15]
coeficientes del numerador de H(z) = [ 0.3657 -1.4627  2.194  -1.4627
0.3657]
coeficientes del denominador de H(z) = [ 1.     -2.0626  1.8611 -0.7931
0.1338]
```

Ahora, vamos a construir $H(s)$ utilizando los dos primeros vectores anteriores. Estos coeficientes se organizan con potencias descendentes de s, empezando en s^4 y terminando en s^0. Por tanto:

$$H(s) = \frac{s^4}{s^4 + 1.64 \times 10^4 s^3 + 1.35 \times 10^8 s^2 + 6.48 \times 10^{11} s + 1.56 \times 10^{15}}$$

Nota aclaratoria: se escribe el resultado con dos cifras significativas para facilitar la lectura.

Ahora, utilizando los coeficientes del numerador y denominador obtenidos para el filtro digital, es posible construir la función de transferencia $H(z)$. Aunque matemáticamente puede expresarse mediante potencias positivas de z, en procesamiento digital de señales es habitual utilizar potencias negativas, ya que esta representación permite identificar directamente los retardos del sistema y facilita la obtención de la ecuación de entrada-salida.

Por tanto, la función de transferencia del filtro digital queda dada por:

$$H(z) = \frac{0.3657 - 1.4627z^{-1} + 2.1940z^{-2} - 1.4627z^{-3} + 0.3657z^{-4}}{1 - 2.0626z^{-1} + 1.8611z^{-2} - 0.7931z^{-3} + 0.1338z^{-4}}$$

Recordando que $H(z) = Y(z)/X(z)$,
podemos obtener:

$$Y(z)(1 - 2.0626z^{-1} + 1.8611z^{-2} - 0.7931z^{-3} + 0.1338z^{-4}) = X(z)(0.3657 - 1.4627z^{-1} + 2.1940z^{-2} - 1.4627z^{-3} + 0.3657z^{-4})$$

Ahora, aplicando la Transformada Z inversa a ambos lados de la ecuación anterior, y recordando que cada término z^{-k} representa un retardo de k muestras en el tiempo discreto, se obtiene:

$$y[n] - 2.0626y[n-1] + 1.8611y[n-2] - 0.7931y[n-3] + 0.1338y[n-4] = 0.3657x[n] - 1.4627x[n-1] + 2.1940x[n-2] - 1.4627x[n-3] + 0.3657x[n-4]$$

Por tanto, la ecuación de entrada-salida es (despejando $y[n]$):

$$\begin{aligned} y[n] = \; & 2.0626y[n-1] - 1.8611y[n-2] + 0.7931y[n-3] - 0.1338y[n-4] \\ & +0.3657x[n] - 1.4627x[n-1] + 2.1940x[n-2] \\ & -1.4627x[n-3] + 0.3657x[n-4] \end{aligned}$$

$$\begin{aligned} y[n] = 2.0626y[n-1] - 1.8611y[n-2] + 0.7931y[n-3] - 0.1338y[n-4] + 0.3657x[n] \\ - 1.4627x[n-1] + 2.1940x[n-2] - 1.4627x[n-3] + 0.3657x[n-4] \end{aligned}$$

Es decir, para calcular la salida actual del filtro, es necesario conocer tanto las cuatro salidas anteriores del sistema como la muestra actual de la señal de entrada y sus cuatro muestras previas. En otras palabras, la salida en el instante n depende no solo de la entrada, sino también de valores pasados de la propia salida, característica que distingue a los filtros IIR de los filtros FIR.

1.7. Implementación de filtros FIR e IIR

Hasta este punto hemos estudiado diferentes métodos de diseño de filtros digitales y hemos obtenido sus correspondientes ecuaciones de entrada-salida. Sin embargo, una pregunta que te puedes estar haciendo es:

¿cómo se implementan realmente estos filtros en un computador, un procesador digital de señales o un sistema embebido?

La respuesta se encuentra en los diagramas de bloques. Cada término de la ecuación de entrada-salida puede asociarse con operaciones básicas como retardos, multiplicaciones y sumas. A partir de estas operaciones es posible construir estructuras que implementan el comportamiento del filtro muestra a muestra.

En esta sección estudiaremos las estructuras básicas de implementación para filtros FIR e IIR, analizando cómo la presencia o ausencia de realimentación modifica tanto el diagrama de bloques como los recursos computacionales necesarios.

1.7.1. Implementación de filtros FIR

Consideremos nuevamente un filtro FIR de orden $M-1$, cuya ecuación de entrada-salida puede escribirse como:

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

Esta ecuación indica que la salida se obtiene como una combinación ponderada de la muestra actual de entrada y de varias muestras pasadas. Cada término $x[n-k]$ puede implementarse mediante un bloque de retardo, mientras que los coeficientes b_k corresponden a multiplicadores y la suma final combina todas las contribuciones.

Ejemplo 9. Diagrama de bloques de un filtro FIR

Supongamos que tenemos un filtro FIR con la siguiente ecuación de entrada-salida del sistema:

$$y[n] = 0.1x[n] + 0.2x[n-1] + 0.4x[n-2] + 0.2x[n-3] + 0.1x[n-4]$$

Aplicando TZ a ambos lados de la ecuación anterior, y utilizando la propiedad de desplazamiento temporal, se obtiene:

$$Y(z) = 0.1X(z) + 0.2z^{-1}X(z) + 0.4z^{-2}X(z) + 0.2z^{-3}X(z) + 0.1z^{-4}X(z)$$

Propiedad de desplazamiento: un retardo de k muestras en el dominio del tiempo se representa como un factor z^{-k} en el dominio z.

Factorizando $X(z)$:

$$Y(z) = X(z)(0.1+0.2z^{-1}+0.4z^{-2}+0.2z^{-3}+0.1z^{-4})$$

Recordando que:

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

se obtiene la función de transferencia:

$$H(z) = 0.1 + 0.2z^{-1} + 0.4z^{-2} + 0.2z^{-3} + 0.1z^{-4}$$

Observamos que el filtro FIR no tiene términos de realimentación asociados a $y[n-k]$. Por eso, su función de transferencia solo depende de los coeficientes aplicados a la entrada actual y a sus muestras pasadas.

Vamos ahora a representar el filtro mediante un diagrama de bloques (Figura 30). Para ello, utilizaremos tres tipos de elementos básicos: unidades de retardo (z^{-1}), amplificadores y sumadores.

A partir de la ecuación de entrada-salida, es posible determinar los recursos necesarios para implementar el filtro:

- **Cuatro unidades de retardo**, ya que el mayor retardo presente en la señal de entrada corresponde a $x[n-4]$.
- **Cinco amplificadores**, debido a que cada uno de los cinco términos de la forma $x[n-k]$debe multiplicarse por su coeficiente correspondiente.
- **Un sumador de cinco entradas**, encargado de combinar las contribuciones de los cinco términos para producir la señal de salida.

Con estos elementos es posible construir la estructura transversal del filtro FIR, mostrada en la Figura 30.

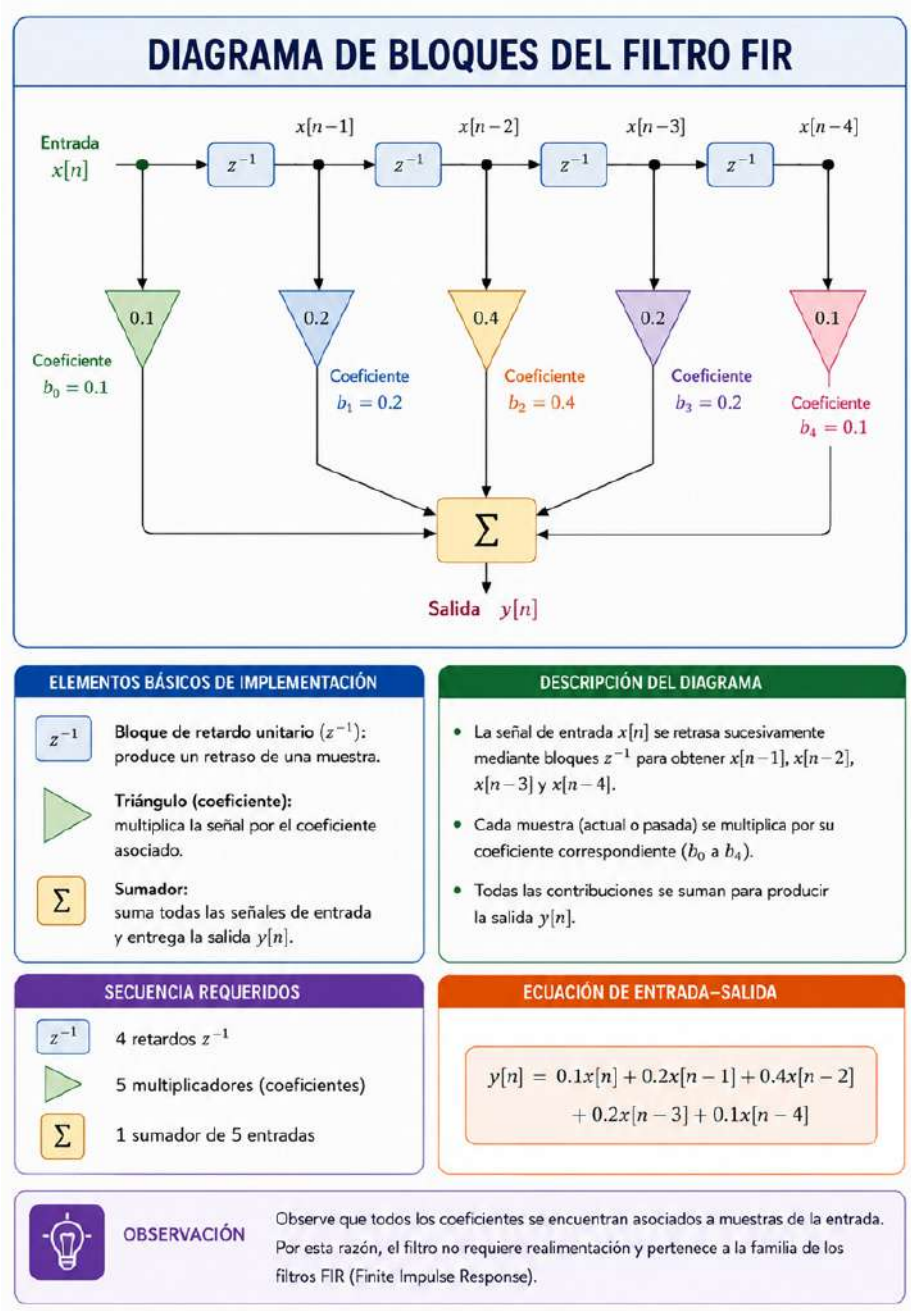


Figura 30. Diagrama de bloques del filtro FIR del Ejemplo 9: bloques de retardo, bloques amplificadores y bloque sumador.

Pensemos, por ejemplo, que cada unidad de retardo requiere un elemento de almacenamiento, como un registro o un *flip-flop*. En consecuencia, un filtro con 100 retardos necesitará una cantidad mucho mayor de memoria que un filtro con solo 10 retardos. De forma similar, al aumentar el orden del filtro también se incrementa el número de bloques de amplificación y de operaciones de suma necesarias para calcular cada muestra de salida. Por esta razón, el diseño de filtros implica un compromiso entre desempeño e implementación: filtros de mayor orden suelen ofrecer una mejor respuesta en frecuencia, pero a costa de un mayor consumo de recursos de memoria y procesamiento.

1.7.2. Implementación de filtros IIR

A primera vista, los filtros IIR parecen plantear un problema difícil de resolver. Si su respuesta al impulso es infinita, podríamos pensar que también necesitaríamos una cantidad infinita de coeficientes, retardos y recursos para implementarlos. Afortunadamente, no es así.

La idea detrás de los filtros IIR es bastante ingeniosa. En lugar de construir la salida únicamente a partir de la señal de entrada y sus retardos (como ocurre en los filtros FIR), podemos mejor aprovechar retardos de la misma salida. Gracias a este mecanismo de realimentación, es posible generar una respuesta al impulso de duración infinita utilizando únicamente una cantidad finita de coeficientes, retardos y operaciones matemáticas.

Pero ¿cómo es posible almacenar el efecto de infinitas muestras pasadas utilizando tan pocos recursos? Para responder esta pregunta, comencemos analizando un acumulador infinito ideal.

Inspiración conceptual: un acumulador infinito

Consideremos el siguiente sistema:

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

el cual podemos reescribir, así:

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

En este caso, la salida corresponde a la suma de la muestra actual y de todas las muestras pasadas de la señal de entrada. A simple vista, parecería que la implementación requeriría una cantidad infinita de retardos.

Si aplicamos un retardo de una muestra a la ecuación anterior, obtenemos:

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

Al comparar ambas expresiones, observamos que todos los términos de la suma, excepto $x[n]$, corresponden precisamente a $y[n-1]$. Por tanto, podemos reescribir la salida como:

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

La Figura 32 ilustra gráficamente este procedimiento.

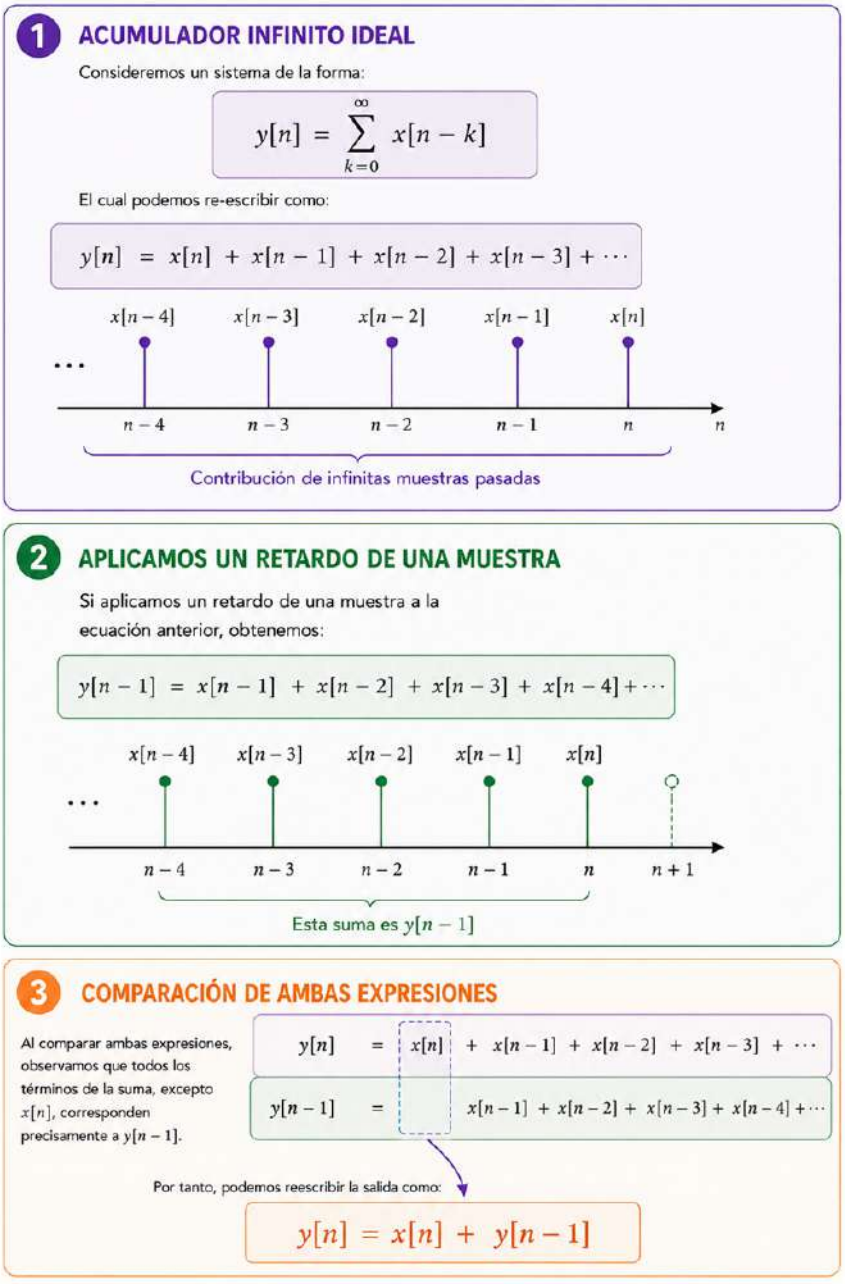


Figura 31. Un acumulador infinito puede reescribirse mediante una ecuación recursiva, reemplazando infinitos retardos de la entrada por un único retardo de la forma $y[n-1]$.

¿Por qué esta reescritura es tan importante?

El resultado anterior es mucho más profundo de lo que parece a primera vista. Inicialmente, la implementación directa del acumulador requeriría una cantidad infinita de retardos, ya que la salida depende de infinitas muestras pasadas de la señal de entrada. Sin embargo, al expresar la salida en términos de

la salida retardada $y[n-1]$, reemplazamos una estructura con infinitos retardos de la entrada por una estructura mucho más sencilla, basada en un único retardo de la salida y un mecanismo de realimentación.

La Figura 32 compara ambas implementaciones. A la izquierda se muestra la implementación directa del acumulador infinito, mientras que a la derecha se presenta su versión recursiva, la cual sólo requiere una unidad de retardo y una realimentación.

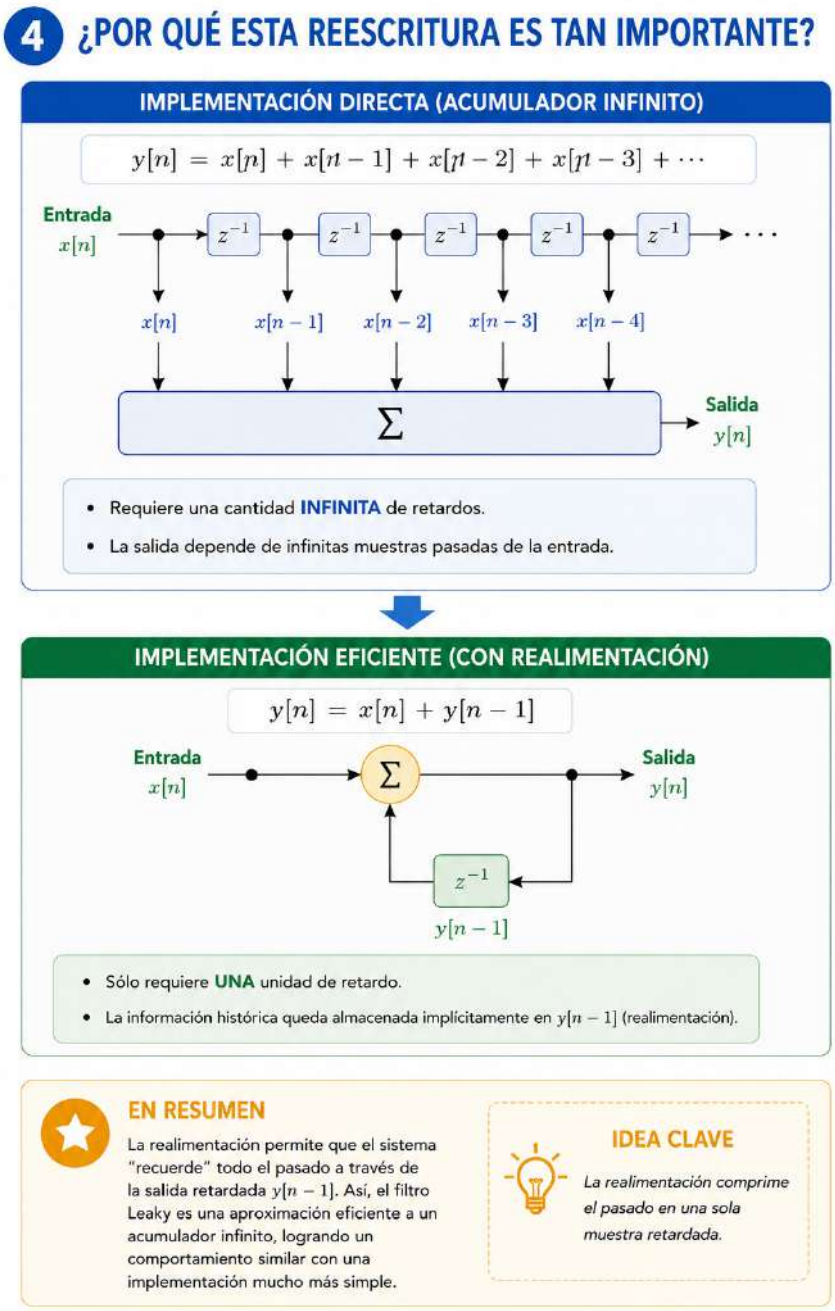


Figura 32. Comparación implementación tradicional vs. Implementación ***recursiva*** del filtro acumulador infinito.

Ejemplo 10. Diagrama de bloques del filtro Leaky y respuesta al impulso

La idea presentada anteriormente sirve como inspiración conceptual para comprender los filtros IIR. Aunque el acumulador infinito y el filtro *Leaky* no son exactamente equivalentes, ambos comparten una característica fundamental: utilizan realimentación para incorporar información proveniente del pasado.

El filtro *Leaky* es uno de los ejemplos más sencillos de filtro IIR. Su comportamiento es similar al de un filtro de promedio, ya que ambos atenúan las variaciones rápidas de la señal y ayudan a reducir el efecto del ruido. Sin embargo, a diferencia del filtro de promedio, el filtro *Leaky* incorpora una rama de realimentación, por lo que pertenece a la familia de los filtros IIR.

La ecuación de entrada-salida del filtro puede expresarse como:

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

donde $0 < \lambda < 1$es un parámetro que controla el grado de suavizado aplicado a la señal.

Podemos observar que la ecuación anterior tiene una estructura muy similar a la obtenida en el acumulador recursivo. La diferencia es que ahora tanto la entrada actual como la salida retardada se ponderan mediante los coeficientes λ y $(1 - \lambda)$ -respectivamente-, permitiendo controlar la contribución de cada una de ellas.

A partir de esta ecuación podemos identificar que:

- La salida depende de una fracción de la muestra actual $x[n]$.
- También depende de la salida retardada $y[n - 1]$, evidenciando la presencia de realimentación.
- El parámetro λ controla el compromiso entre seguimiento de la señal y suavizado.

Al igual que en el caso de los filtros FIR, la ecuación de entrada-salida nos permite determinar los recursos necesarios para implementar el sistema. En este caso, se requiere una unidad de retardo, dos amplificadores y un sumador de dos entradas, tal como se muestra en la Figura 33.

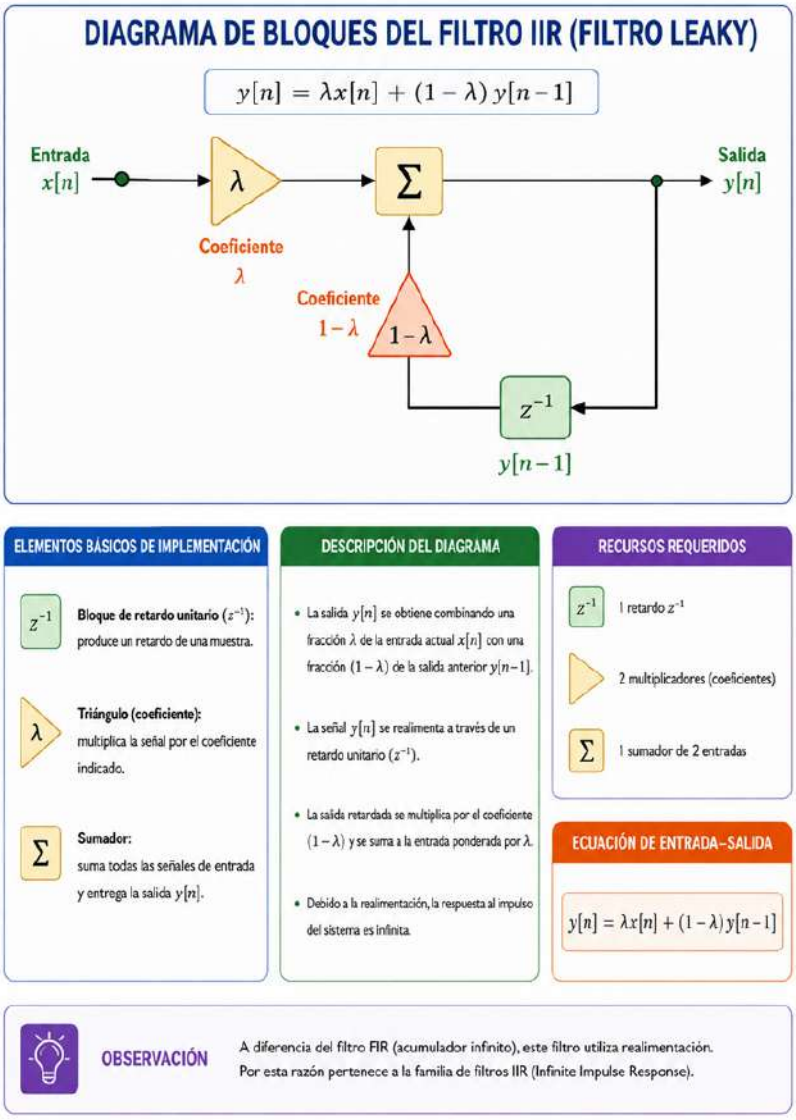


Figura 33. Diagrama de bloques del filtro IIR del Ejemplo 10. A diferencia de los filtros FIR, la salida retardada $y[n-1]$ se realimenta hacia el sumador, permitiendo que muestras anteriores influyan en la salida actual.

Ya hemos visto de donde sale el filtro Leaky, cuál fue su inspiración, y lo que significa una implementación recursiva. Ahora, vamos a revisar cómo es la respuesta al impulso del filtro.

Para ello, partimos de la ecuación:

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

y entonces, hacemos que la entrada sea una señal impulso, es decir:

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

Adicionalmente, asumiremos condiciones iniciales, en la cuales, el sistema "arranca" en el instante cero, y para todos los valores anteriores la salida es cero, es decir:

$$y[-1] = 0$$

A partir de estas condiciones, podemos calcular las primeras muestras de la respuesta al impulso.

Para $n = 0$:

$$y[0] = \lambda\delta[0] + (1 - \lambda)y[-1]$$

$$y[0] = \lambda$$

Para $n = 1$:

$$y[1] = \lambda\delta[1] + (1 - \lambda)y[0]$$

$$y[1] = \lambda(1 - \lambda)$$

Para $n = 2$:

$$y[2] = \lambda(1 - \lambda)^2$$

Para $n = 3$:

$$y[3] = \lambda(1 - \lambda)^3$$

Vemos que la salida sigue un patrón, de tal forma que la respuesta al impulso, la podemos expresar como:

$h[n] = \lambda(1 - \lambda)^n u[n]$ donde $u[n]$ corresponde a la señal escalón unitario, para hacer énfasis que el sistema inicia en $n = 0$.

Esta respuesta al impulso, aunque es de duración infinita, vemos que se atenúa a medida que nos alejamos del origen, a continuación, veremos algunos ejemplos (Figura 34):

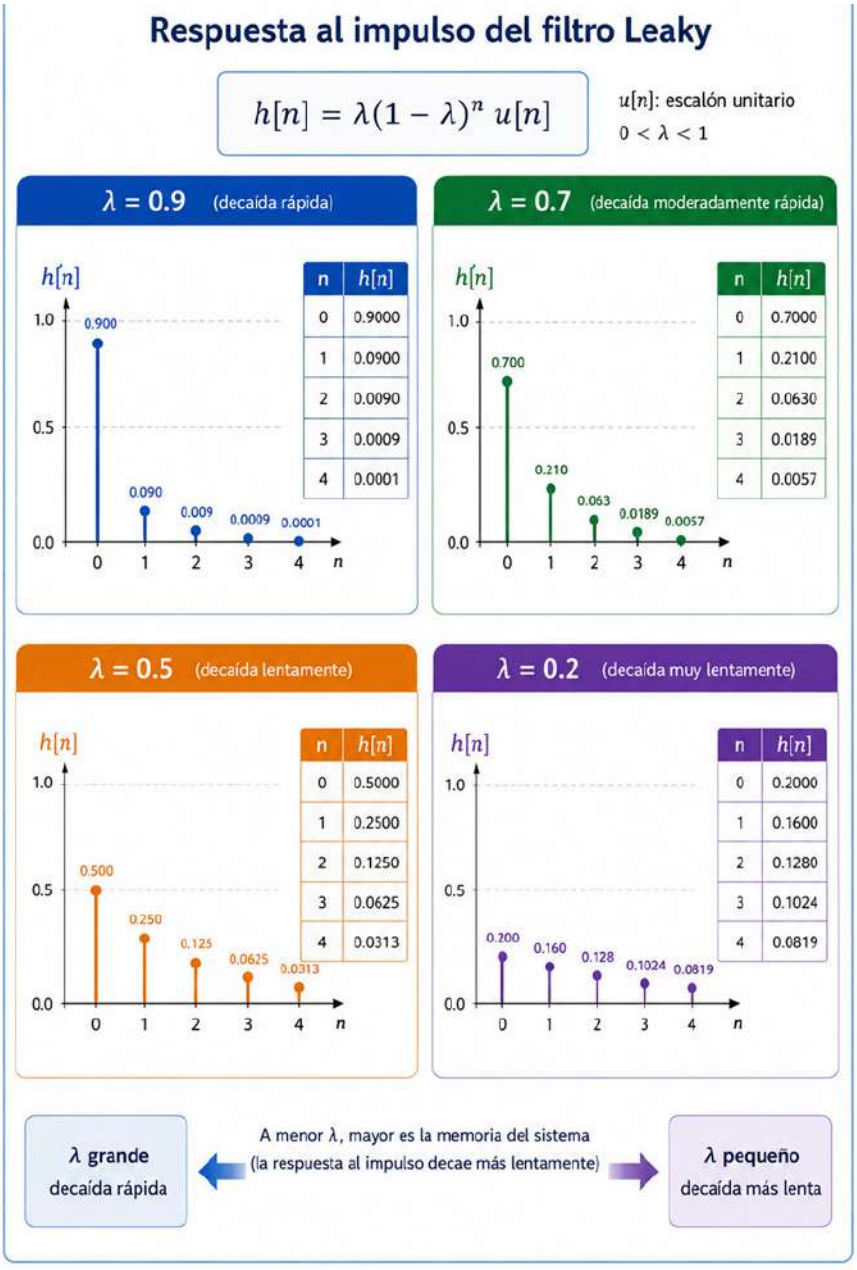


Figura 34. Respuesta al impulso filtro Leaky, para $\lambda = 0.9, 0.7, 0.5, 0.2$.

Un filtro Leaky es controlado por el valor de lambda. En el siguiente capitulo veremos su impacto en una aplicación de filtrado de señal de voz.

CAPÍTULO II

ELIMINACIÓN DE RUIDO CON FILTROS CLÁSICOS

Hasta este punto hemos estudiado conceptos fundamentales como señales discretas, polos y ceros, filtros digitales y diferentes estrategias para su implementación. Sin embargo, es posible que te estés preguntando:

¿Para qué sirve todo esto?

La respuesta es sencilla: para modificar señales y mejorar su calidad.

Aunque las técnicas presentadas en el Capítulo I las podemos aplicar a cualquier tipo de señal unidimensional, en este capítulo nos centraremos en una señal muy especial: la voz humana. Como mencioné al inicio de esta primera parte del libro, fue el tipo de señal que escogí para desarrollar mi tesis doctoral y, desde entonces, ha ocupado un lugar muy importante dentro de mi actividad investigativa. Por esta razón, además de su enorme relevancia práctica, tiene para mí un significado emocional muy especial.

Pero el objetivo de este libro va más allá de presentar conceptos teóricos. Mi intención es que el estudiante no solo aprenda Procesamiento Digital de Señales, sino que también lo experimente y lo viva. Por ello, he reservado este capítulo para aplicar los conceptos estudiados hasta ahora a través de un caso de estudio que nos permitirá observar, paso a paso, cómo diferentes técnicas de procesamiento pueden transformar una señal real.

Para ello, partiremos de una señal de voz que será contaminada artificialmente con ruido. Nuestro objetivo consistirá en reducir dicho ruido utilizando diferentes tipos de filtros digitales. Sin embargo, antes de diseñar cualquier filtro, tendremos que responder una pregunta fundamental: **¿cuáles son las características del ruido que queremos eliminar?**

La respuesta a esta pregunta nos permitirá identificar los parámetros de diseño más adecuados y comprender una de las ideas más importantes del Procesamiento Digital de Señales: antes de modificar una señal, debemos entenderla.

2.1 Análisis temporal y frecuencial de una señal de voz

En muchas ocasiones, el análisis temporal de una señal no es suficiente para comprenderla completamente. Por esta razón, también resulta necesario estudiar su comportamiento en frecuencia. Pero, existen varias opciones de análisis frecuencial, las cuales veremos en el siguiente ejemplo.

Ejemplo 11. Lectura y visualización en el dominio del tiempo y la frecuencia de una señal de voz

Para este caso de estudio trabajaremos con una señal de voz que contiene el mensaje “procesamiento digital de señales”. Inicialmente, leeremos y visualizaremos en el dominio del tiempo la señal, con el siguiente código en Python:

```python
import numpy as np
import matplotlib.pyplot as plt
import librosa

audio_file = "audio.wav"   # reemplazar por la ruta del archivo

x, fs = librosa.load(audio_file, sr=None) # hacer sr=None para preservar fs
x_norm = x / np.max(np.abs(x)) # normalizar el audio

N = len(x_norm) # total de muestras del audio
duracion = N / fs # duración en segundos del audio

print(f"Frecuencia de muestreo: {fs} Hz")
print(f"Número de muestras: {N}")
print(f"Duración: {duracion:.2f} segundos")

t = np.arange(N) / fs

plt.figure(figsize=(12,4))
plt.plot(t, x_norm)
plt.xlabel("Tiempo [s]")
plt.ylabel("Amplitud")
plt.grid(True)
plt.xlim([0, duracion])
plt.tight_layout()
plt.show()
```

Figura 35. Código en Python del Ejemplo 11 para la lectura y visualización de una señal de audio.

Y obtenemos:

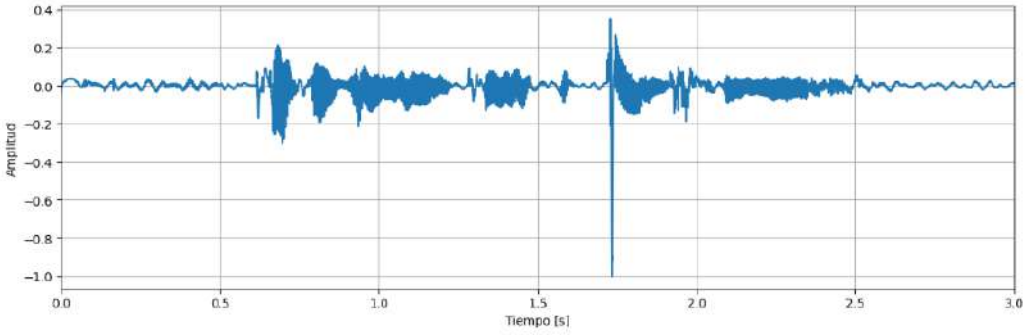


Figura 36. Ejemplo 11: lectura y visualización de una señal de audio.

```
Frecuencia de muestreo: 8000 Hz
Número de muestras: 24000
Duración: 3.00 segundos
```

El código anterior, además de leer y graficar la señal, así como de mostrar los valores de frecuencia de muestreo y duración, también nos permite

normalizar la amplitud del audio en el rango $[-1,1]$, lo cual nos facilitará la comparación entre diferentes grabaciones, evitando que las amplitudes observadas dependan de la escala original utilizada durante la adquisición.

En este caso específico, podemos observar que el pico negativo (es decir, el valle) presenta una amplitud considerablemente mayor que el pico positivo. Este comportamiento es completamente normal en señales de voz y depende tanto del contenido de la grabación como de las características propias del proceso de adquisición.

Para el análisis en frecuencia partiremos del espectro, utilizando la FFT. Dado que la frecuencia de muestreo es de 8 kHz, el espectro unilateral estará comprendido entre 0 y 4 kHz, de acuerdo con el criterio de Nyquist.

```python
import numpy as np
import matplotlib.pyplot as plt
import librosa
from scipy.fft import rfft, rfftfreq
from scipy.signal import find_peaks

N = len(x_norm)
X = rfft(x_norm)
f = rfftfreq(N, d=1/fs)
mag = np.abs(X) / N # Magnitud normalizada
mag[1:-1] = 2 * mag[1:-1] # Corrección para espectro unilateral

plt.figure(figsize=(12,4)); plt.plot(f, mag)
plt.xlabel('Frecuencia [Hz]'); plt.ylabel('Magnitud')
plt.grid(True); plt.xlim([0, fs/2])
plt.tight_layout(); plt.show()

picos, propiedades = find_peaks(
    mag,
    height=0.001,      # altura mínima del pico
    distance=20        # separación mínima entre picos, en muestras del espectro
)

frecuencias_picos = f[picos]
magnitudes_picos = mag[picos]

idx = np.argsort(magnitudes_picos)[::-1]   # ordenado de mayor a menor
print("Tres picos principales del espectro:\n")

for i in range(min(3, len(idx))):
    k = idx[i]
    print(
        f"Pico {i+1}: "
        f"Frecuencia = {frecuencias_picos[k]:.2f} Hz, "
        f"Magnitud = {magnitudes_picos[k]:.6f}"
    )
```

Figura 37. Código en Python del Ejemplo 11 para el cálculo y visualización del espectro de un audio.

Obteniendo,

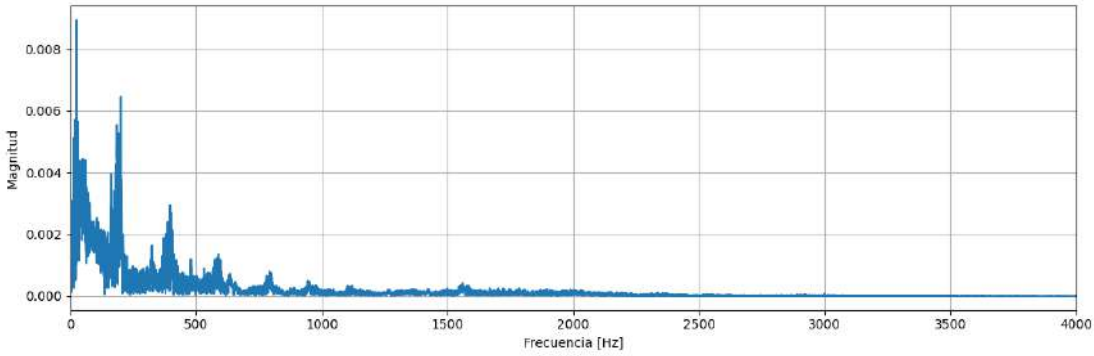


Figura 38. Ejemplo 11: espectro de una señal de audio.

Tres picos principales del espectro:

Pico 1: Frecuencia = 25.00 Hz, Magnitud = 0.008958
Pico 2: Frecuencia = 199.67 Hz, Magnitud = 0.006468
Pico 3: Frecuencia = 17.33 Hz, Magnitud = 0.005722

Nos podemos dar cuenta que la mayor parte de la energía se concentra en las frecuencias bajas, principalmente menor a 1 KHz -típico de señales de voz. Adicionalmente, aparecen varios picos espectrales asociados a las componentes dominantes de la señal de voz y a sus armónicos. A partir de 2 K Hz, la amplitud del espectro es casi cero.

Aunque el espectro es bastante útil para conocer el comportamiento en frecuencia de una señal, aparece una nueva inquietud: ¿Se puede analizar de forma simultánea el comportamiento en el tiempo y en la frecuencia de un audio?

La respuesta es sí, por medio del espectrograma.

Un espectrograma es una representación (gráfica) tiempo-frecuencia que muestra cómo se distribuye la energía espectral de una señal a lo largo del tiempo. En esta representación, el eje horizontal corresponde al tiempo, mientras que, el eje vertical corresponde a la frecuencia. Adicionalmente, la intensidad del color indica la magnitud o energía presente en cada combinación tiempo-frecuencia.

A continuación, veremos un comparativo entre espectro y espectrograma, cómo son sus ejes y qué información puedo obtener de cada uno de ellos (Figura 39).

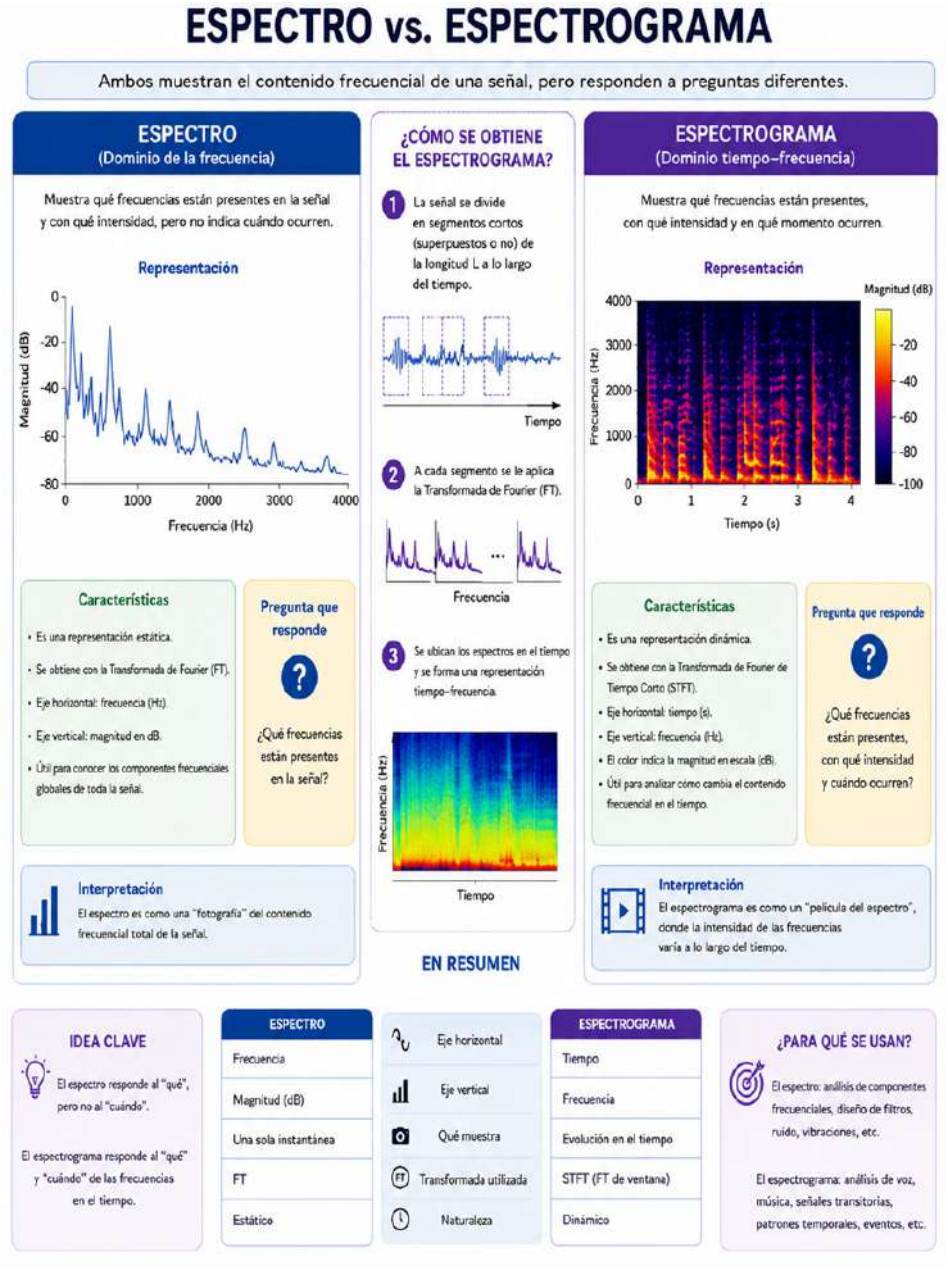


Figura 39. Comparación entre espectro y espectrograma.

Mientras el espectro ofrece una representación ***estática*** del contenido frecuencial de la señal, tenemos que el espectrograma proporciona una representación ***dinámica*** de su evolución en el tiempo. Por ejemplo, en el espectrograma podríamos apreciar en qué instante de tiempo existió un cambio significativo de frecuencia, algo que no podríamos apreciar en el espectro.

Ahora vamos a calcular y visualizar el espectrograma logarítmico de nuestra señal de voz.

```
import numpy as np
import matplotlib.pyplot as plt
import librosa
import librosa.display

n_fft = 1024
hop_length = 512
n_mels = 128

# Espectrograma logarítmico
S = np.abs(librosa.stft(x, n_fft=n_fft, hop_length=hop_length))**2
S_db = librosa.power_to_db(S, ref=np.max)

plt.figure(figsize=(8, 4))
librosa.display.specshow(
    S_db,
    sr=fs,
    hop_length=hop_length,
    x_axis="time",
    y_axis="hz"
)
plt.colorbar(format="%+2.0f dB")
plt.xlabel("Tiempo [s]")
plt.ylabel("Frecuencia [Hz]")
plt.tight_layout()
plt.show()
```

Figura 40. Código en Python del Ejemplo 11 para el cálculo y visualización del espectrograma.

Y obtenemos:

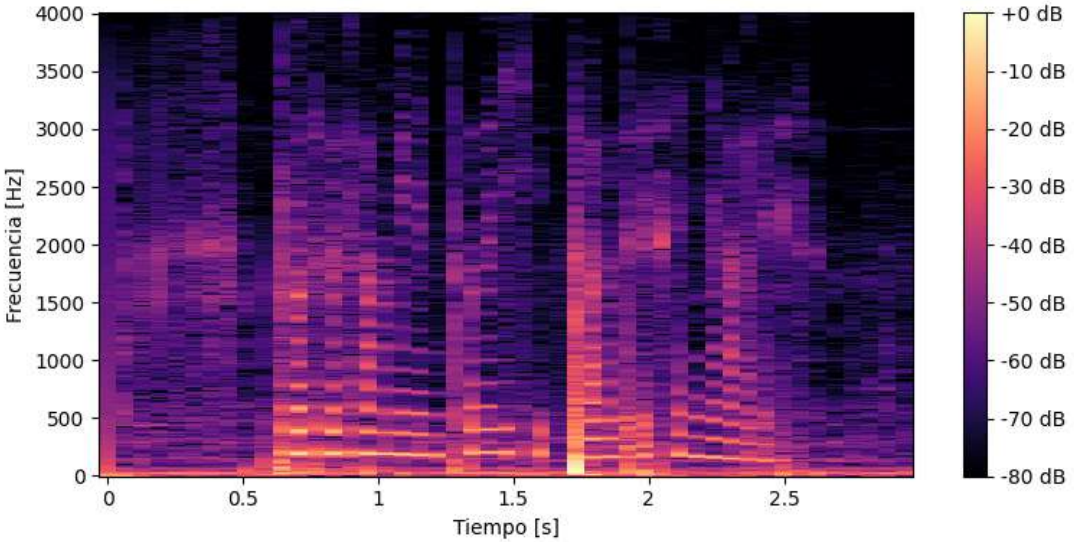


Figura 41. Ejemplo 11: espectrograma de una señal de audio.

Los colores más claros (amarillo y blanco) indican las regiones donde la señal tiene mayor energía; mientras que, los tonos morados y negros corresponden a componentes de baja energía o incluso a silencios. Podemos observar que la mayor parte de la energía se concentra en las bajas frecuencias (entre 0 y 1000 Hz), aunque algunas palabras generan componentes que se extienden hacia frecuencias más altas (alrededor de los 1500 Hz). También aparecen varias franjas oscuras a lo largo del tiempo, las cuales corresponden a silencios entre palabras o frases. En conjunto, con esta representación apreciamos cómo la voz va cambiando en el tiempo, revelando detalles que serían difíciles de identificar observando únicamente la señal en el dominio temporal o frecuencial.

En la siguiente gráfica podrás encontrar la explicación de los tres parámetros que utilizamos para calcular el espectrograma: n_fft, hop_length y n_mels. Cada parámetro en términos de su significado, qué controla, valores recomendados y su efecto de cambiar el valor. Y valores sugeridos en conjunto para tres tipos de aplicaciones diferentes: voz, música, y sonidos rápidos/ transitorios.

Y a manera de reflexión personal: ¡cómo me hubiese gustado tener esta ficha a la mano cuando empecé a investigar en audio!

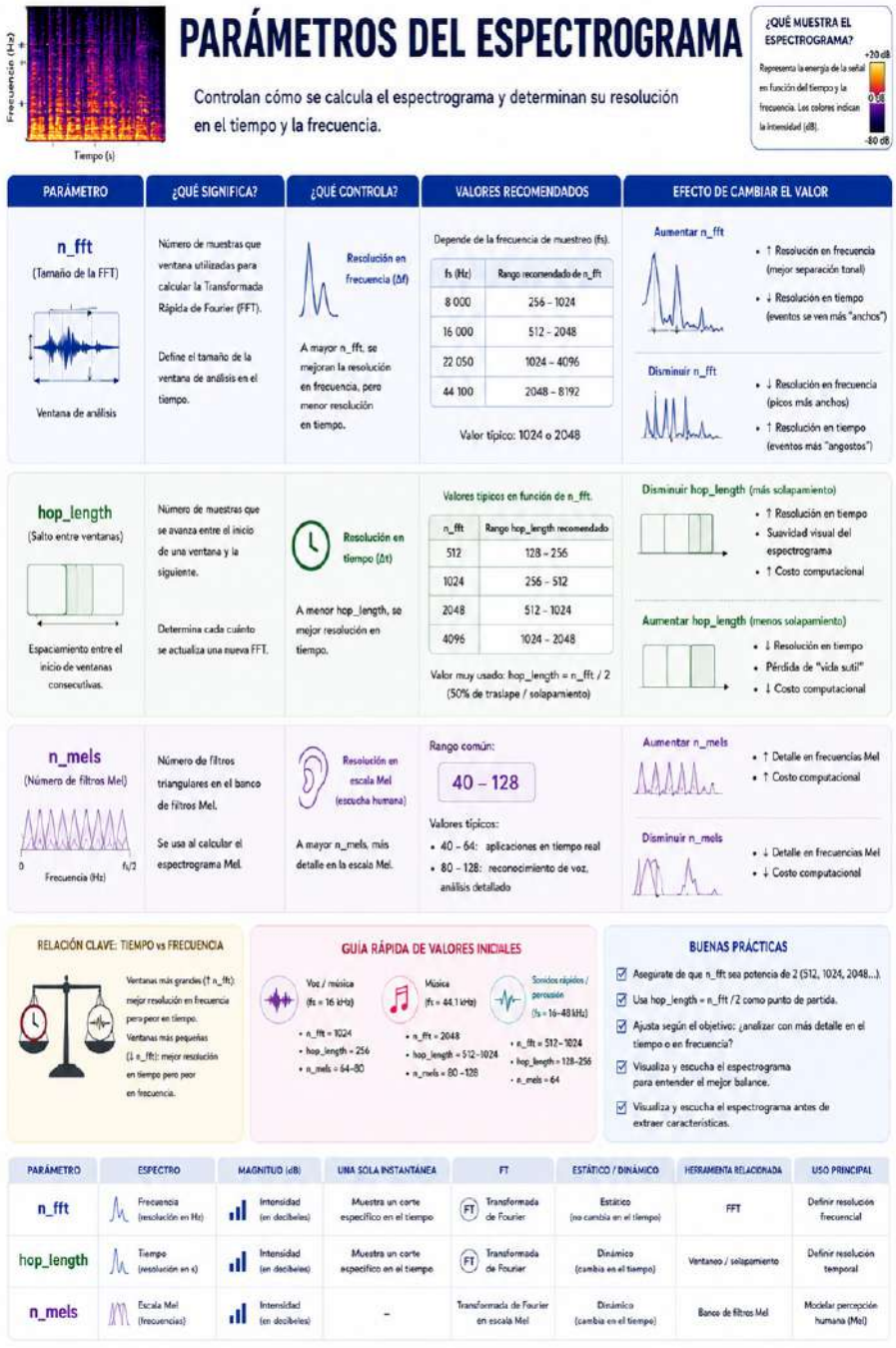


Figura 42. Parámetros del espectrograma: significado, valores recomendados, y efecto.

Terminamos este ejemplo y ya conocemos el comportamiento en tiempo, frecuencia y tiempo-frecuencia de una señal de voz que no tiene ruido. En el próximo ejemplo trabajaremos con una señal con ruido, y podremos comparar las gráficas para reconocer qué tipo de ruido tenemos presente y en qué frecuencia(s) está ubicado. Esto será esencial para seleccionar adecuadamente los parámetros de diseño de los filtros, ya que antes de eliminar un ruido debemos identificar su naturaleza y determinar en qué región del espectro se encuentra.

2.2 Señal de voz con ruido

Ahora sí, llegamos a la parte más importante de esta primera Sección: enfrentarnos a un problema real, una señal de voz contaminada con ruido. Y no, no te voy a decir qué tipo de ruido tiene, así como tampoco en qué frecuencia(s) está ubicado. Lo primero que tendrás que hacer es descubrirlo.

Una de las habilidades más importantes de un ingeniero consiste en analizar la información disponible, formular hipótesis y tomar decisiones a partir de la evidencia observada. Precisamente, eso es lo que haremos en este ejemplo.

Ejemplo 12. Análisis de la señal de voz con ruido

Lo primero que quiero que hagas, una vez tengas disponible el archivo de audio, es escucharlo atentamente. Después de escucharlo, intenta responder las siguientes preguntas:

- ¿Qué tipo de ruido crees que está presente?
- ¿El ruido se encuentra durante toda la grabación o solamente en algunos instantes?
- ¿Dirías que el ruido corresponde a una frecuencia baja o a una frecuencia alta?

Ahora bien, escuchar el audio es un excelente punto de partida, pero en muchas ocasiones nuestros oídos no son suficientes para caracterizar completamente una señal. Es por esta razón que, además del audio, te proporcionaré tres herramientas de análisis que ya conocemos: gráfica en el dominio del tiempo, el espectro y el espectrograma logarítmico (Figura 43).

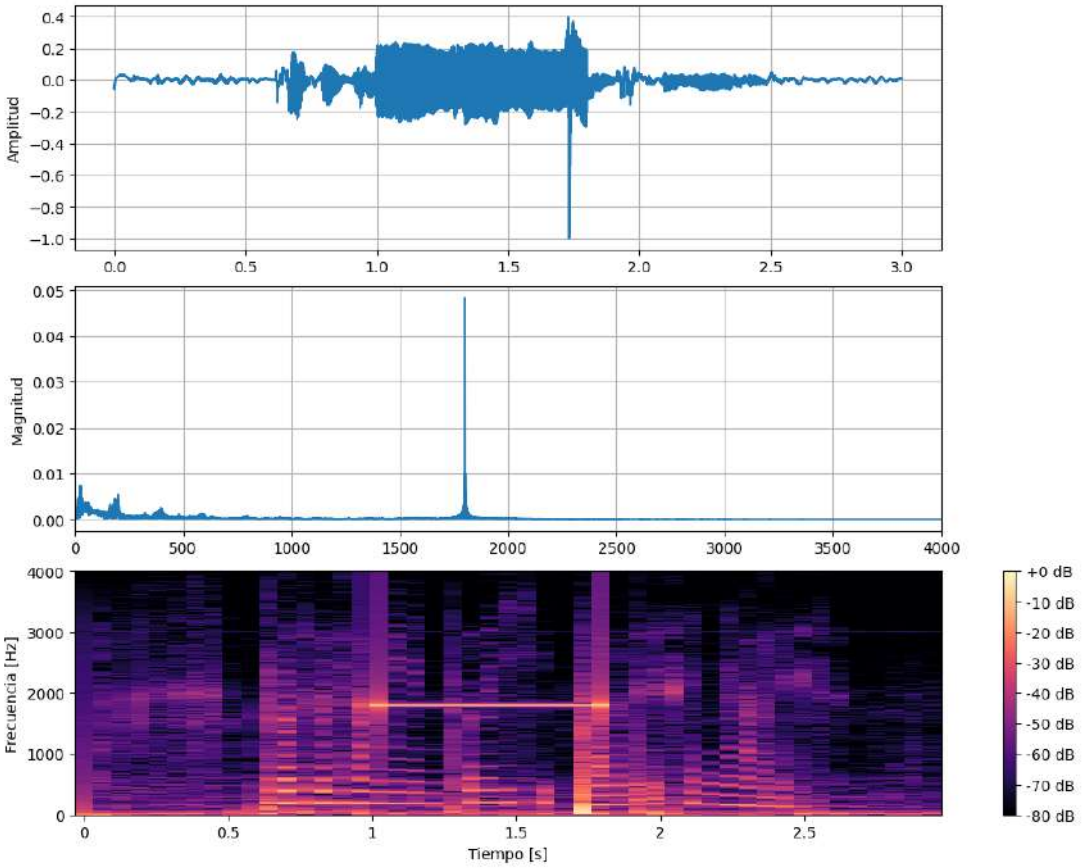


Figura 43. Señal de audio con ruido del Ejemplo 12: dominio temporal (superior), espectro (central) y espectrograma en escala logarítmico (inferior).

Antes de leer la explicación que aparece a continuación, observa cuidadosamente las figuras y responde las siguientes preguntas:

- ¿En qué frecuencia parece concentrarse la energía asociada al ruido?
- ¿El ruido está presente durante toda la grabación o solamente durante un intervalo específico?
- ¿Qué representación te proporciona más información para identificar la naturaleza del ruido: ¿el dominio del tiempo, el espectro o el espectrograma?

Vamos a ver si coincidimos con el análisis de las gráficas:

- En el dominio del tiempo, observamos que la señal se vuelve más “ancha” entre los segundos 1 y 1.8, aproximadamente. Sin embargo, a partir de esta representación no podemos identificar con certeza qué tipo de ruido está presente.
- En el espectro, observamos un pico de energía alrededor de los 1800 Hz, el cual no forma parte del comportamiento típico de una señal de voz. Esto nos permite inferir que existe una componente tonal concentrada en dicha frecuencia.
- En el espectrograma, aparece una línea horizontal de color amarillo, es decir, una región de alta energía, ubicada alrededor de los 1800 Hz. Sin embargo, esta componente no está presente durante toda la grabación, sino únicamente entre los segundos 1 y 1.8.
- Así que, en el dominio temporal identificamos aproximadamente cuándo aparece el ruido; en el espectro identificamos qué tipo de componente frecuencial lo caracteriza; y en el espectrograma obtenemos simultáneamente ambas informaciones: cuándo aparece y en qué frecuencia está ubicado.
-

Y claro, si previamente no conociéramos cuál es el comportamiento “normal” de una señal de voz, difícilmente habríamos podido identificar la presencia del ruido, determinar su frecuencia y reconocer que corresponde a una componente externa a la señal original.

Ya habiendo caracterizado el ruido, la pregunta que nos queda es:

¿qué tipo de filtro sería el más adecuado para reducir, en la medida de lo posible, esta interferencia?

La respuesta no es única:

- Podríamos diseñar un **filtro pasa-banda de banda ancha**, comprendido, por ejemplo, entre **50 Hz y 1200 Hz**. Dado que en este intervalo se concentra gran parte de la energía de la señal de voz, estaríamos preservando la información más relevante y dejando por fuera el tono ubicado en **1800 Hz**.

- Una segunda opción consiste en utilizar un **filtro notch (rechaza-banda)** con frecuencia de resonancia de **1800 Hz**. De esta forma, eliminaríamos específicamente el tono, afectando mínimamente el resto del contenido frecuencial de la señal.
- Una tercera alternativa es un filtro **pasa-bajos** con frecuencia de corte de **1200 Hz**. En este caso, también lograríamos atenuar el tono de 1800 Hz al encontrarse por encima de la frecuencia de corte seleccionada.

Personalmente, me inclino por la tercera opción. Además de reducir el tono, el filtro pasa-bajos nos permitirá eliminar componentes de alta frecuencia que aportan poca información a la señal de voz. Por ejemplo, también estaríamos atenuando el contenido comprendido entre 1200 Hz y 1800 Hz, región donde observamos una contribución energética relativamente baja en comparación con las frecuencias inferiores. Así que, para este caso de estudio, este será el tipo de filtro que diseñaremos a continuación.

2.3 Filtros digitales aplicados al caso de estudio

¡Ahora sí, manos a la obra! Ha llegado el momento de aplicar los conceptos estudiados en el Capítulo 1 a un problema real de procesamiento digital de señales. Para ello, trabajaremos con la señal de voz presentada en el **Ejemplo 12**, la cual fue contaminada con un ruido tipo tono durante un fragmento específico del audio.

Nuestro objetivo será evaluar cómo diferentes filtros digitales permiten atenuar esta interferencia y mejorar la calidad de la señal. A lo largo de esta sección analizaremos tanto filtros **FIR** (*Finite Impulse Response*) como filtros **IIR** (*Infinite Impulse Response*), observando sus ventajas, limitaciones y comportamiento sobre la señal de voz.

Comenzaremos con el **filtro de promedio**, uno de los filtros FIR más simples y utilizados, y avanzaremos gradualmente hacia estructuras más elaboradas, finalizando con el **filtro Leaky**, un filtro IIR sencillo pero muy útil para comprender el efecto de la realimentación en los sistemas digitales.

En todos los ejemplos utilizaremos la función **filtfilt()**, ampliamente empleada en procesamiento digital de señales. Esta función recibe como entrada los vectores **b** y **a**, correspondientes a los coeficientes del numerador y del denominador de la función de transferencia del filtro, respectivamente. Adicionalmente, recibe la señal que se desea filtrar.

Una característica importante de **filtfilt()** es que aplica el filtrado dos veces: primero en la dirección normal de la señal y posteriormente en dirección inversa. Como resultado, se obtiene una respuesta de fase nula, evitando desplazamientos temporales y distorsiones de fase en la señal filtrada. Esta propiedad es especialmente útil en aplicaciones de audio, donde la preservación de la forma temporal de la señal resulta fundamental.

Ejemplo 13. Filtrado de la señal con filtro de promedio

Vamos a diseñar un filtro de promedio y a ajustar el valor de M, de tal forma que, el primer cruce por cero del filtro (es decir, su lóbulo principal), sea

$$\omega_{cN} = \frac{f_c}{f_s/2} * \pi = \frac{1200}{4000} * \pi = 0.3 * \pi = 0.942$$

Recordando del Capítulo 1, para el filtro de promedio, el primer cruce por cero ocurre en:

$$\omega_1 = \frac{2\pi}{M}$$

Queremos que:

$$\omega_1 \approx 0.3\pi$$

Entonces:

$$\frac{2\pi}{M} = 0.3\pi$$

Cancelando π:

$$\frac{2}{M} = 0.3$$

$$M = \frac{2}{0.3} = 6.67$$

Como M debe ser entero, las opciones cercanas son:

$$M = 6 \Rightarrow \omega_1 = \frac{2\pi}{6} = 0.333\pi \qquad M = 7 \Rightarrow \omega_1 = \frac{2\pi}{7} = 0.286\pi$$

Y seleccionamos el valor mayor, que nos daría una frecuencia de corte ligeramente menor a la seleccionada de 1200 Hz, aproximadamente en:

$$f_1 = \frac{f_s}{M} = \frac{8000}{7} = 1142.86 \text{ Hz}$$

Así garantizamos que el tono de 1800 Hz quede por fuera del lóbulo principal del filtro de promedio.

Es decir, tenemos que la respuesta al impulso es

$$h[k] = \frac{1}{7}, k = 0,1, \ldots,6$$

De esta forma, cada muestra de salida se obtiene como el promedio de siete muestras consecutivas de la señal de entrada.

```python
import numpy as np
import matplotlib.pyplot as plt
import librosa
from scipy.signal import filtfilt, spectrogram
from IPython.display import Audio, display

x_ruidosa_filepath = "audio_noisy.wav" # Lectura del audio
x, fs = librosa.load(x_ruidosa_filepath, sr=None)

M = 7 # Filtrado con filtro de promedio (M = 7)
b, a = np.ones(M) / M, [1]
x_filtrada = filtfilt(b, a, x)

f1, t1, Sxx1 = spectrogram(x, fs=fs, nperseg=256, noverlap=128) # espectrograma ruidosa
f2, t2, Sxx2 = spectrogram(x_filtrada, fs=fs, nperseg=256, noverlap=128) # espectrograma filtrada
vmin, vmax = -120, -20
fig, ax = plt.subplots(2, 1, figsize=(12, 8))

im1 = ax[0].pcolormesh(t1, f1, 10*np.log10(Sxx1 + 1e-12),
                       shading='gouraud', vmin=vmin, vmax=vmax)
ax[0].set_title('Espectrograma de la señal ruidosa'); ax[0].set_ylabel('Frecuencia [Hz]')
ax[0].set_ylim([0, 4000])
cbar1 = fig.colorbar(im1, ax=ax[0]); cbar1.set_label('Magnitud [dB]')

im2 = ax[1].pcolormesh(t2, f2, 10*np.log10(Sxx2 + 1e-12),
                       shading='gouraud', vmin=vmin, vmax=vmax)
ax[1].set_title('Espectrograma de la señal filtrada (Filtro de promedio, M = 7)')
ax[1].set_xlabel('Tiempo [s]'); ax[1].set_ylabel('Frecuencia [Hz]')
ax[1].set_ylim([0, 4000])
cbar2 = fig.colorbar(im2, ax=ax[1]); cbar2.set_label('Magnitud [dB]')

plt.tight_layout();plt.show()

print("Audio ruidoso:")
display(Audio(x, rate=fs)) # play audio ruidoso
print("Audio filtrado:")
display(Audio(x_filtrada, rate=fs)) # play audio filtrado
```

Figura 44. Código en Python del Ejemplo 13: diseño y aplicación de filtro de promedio.

Y obtenemos:

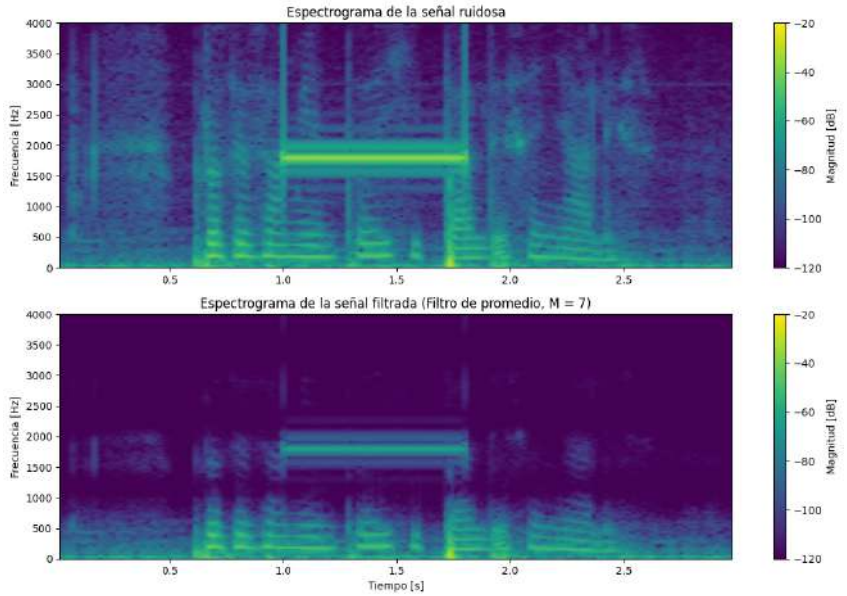


Figura 45. Espectrograma del Ejemplo 13: señal original (superior) vs señal filtrada (inferior), utilizando filtro de promedio con M=7.

La Figura 45 compara los espectrogramas de la señal ruidosa y de la señal filtrada mediante un filtro de promedio con $M = 7$. Observamos que la interferencia tonal ubicada alrededor de 1800 Hz disminuye después del filtrado, aunque no desaparece por completo. Sin embargo, también se atenúan otras componentes frecuenciales de la señal de voz, evidenciando que la reducción del ruido viene acompañada de una pérdida parcial de información útil.

Cuando reproducimos la señal filtrada, percibimos una voz más "ahogada" o apagada. Esto se debe a que el filtro no solo reduce la interferencia, sino que también atenúa componentes propias de la voz, especialmente en la región comprendida entre 500 y 1000 Hz, donde se concentra una parte importante de la información necesaria para su correcta percepción.

Finalmente, podemos guardar la señal filtrada, así:

```
write("audio_filtrado_promedio.wav", fs, x_filtrada.astype(np.float32))
```

Ejemplo 14. Filtrado con filtro FIR utilizando método ventaneo

Utilizamos un filtro FIR diseñado mediante el método de ventaneo, empleando una ventana Blackman. A diferencia del filtro de promedio, esta estrategia permite controlar de forma más directa la frecuencia de corte y obtener una respuesta en frecuencia más selectiva. En este caso, seleccionaremos un orden de filtro alto, con el propósito de lograr una transición más rápida entre la banda de paso y la banda de rechazo, y así atenuar con mayor fuerza la interferencia tonal ubicada alrededor de 1800 Hz.

Entonces, seleccionamos M = 100, y asignamos $f_c = 1200\ Hz$.

A diferencia del Capítulo 1, donde calculábamos explícitamente la ventana y luego la multiplicábamos por la respuesta al impulso ideal para obtener el filtro resultante, ahora utilizaremos la instrucción *firwin*, que realiza este procedimiento de forma automática y entrega directamente la respuesta al impulso del filtro diseñado.

```
from scipy.signal import firwin, filtfilt

fc = 1200
orden = 100 # Se denomina como N

x_ruidosa_filepath = "audio_noisy.wav" # Lectura del audio
x, fs = librosa.load(x_ruidosa_filepath, sr=None)


b = firwin(numtaps=orden+1, cutoff=fc, fs=fs, window='blackman')
a = [1]

x_filtrada_blackman = filtfilt(b, a, x)

f1, t1, Sxx1 = spectrogram(x, fs=fs, nperseg=256, noverlap=128) # espectrograma ruidosa
f2, t2, Sxx2 = spectrogram(x_filtrada_blackman, fs=fs, nperseg=256, noverlap=128) # espectrograma filtrada
vmin, vmax = -120, -20
fig, ax = plt.subplots(2, 1, figsize=(12, 8))

im1 = ax[0].pcolormesh(t1, f1, 10*np.log10(Sxx1 + 1e-12),
                       shading='gouraud', vmin=vmin, vmax=vmax)
ax[0].set_title('Espectrograma de la señal ruidosa'); ax[0].set_ylabel('Frecuencia [Hz]')
ax[0].set_ylim([0, 4000])
cbar1 = fig.colorbar(im1, ax=ax[0]); cbar1.set_label('Magnitud [dB]')

im2 = ax[1].pcolormesh(t2, f2, 10*np.log10(Sxx2 + 1e-12),
                       shading='gouraud', vmin=vmin, vmax=vmax)
ax[1].set_title('Espectrograma de la señal filtrada (Ventana Blackman, N = 100)')
ax[1].set_xlabel('Tiempo [s]'); ax[1].set_ylabel('Frecuencia [Hz]')
ax[1].set_ylim([0, 4000])
cbar2 = fig.colorbar(im2, ax=ax[1]); cbar2.set_label('Magnitud [dB]')

plt.tight_layout();plt.show()

print("Audio ruidoso:")
display(Audio(x, rate=fs)) # play audio ruidoso
print("Audio filtrado:")
display(Audio(x_filtrada_blackman, rate=fs)) # play audio filtrado
```

Figura 46. Código en Python del Ejemplo 14: diseño y aplicación de filtro con ventana Blackman.

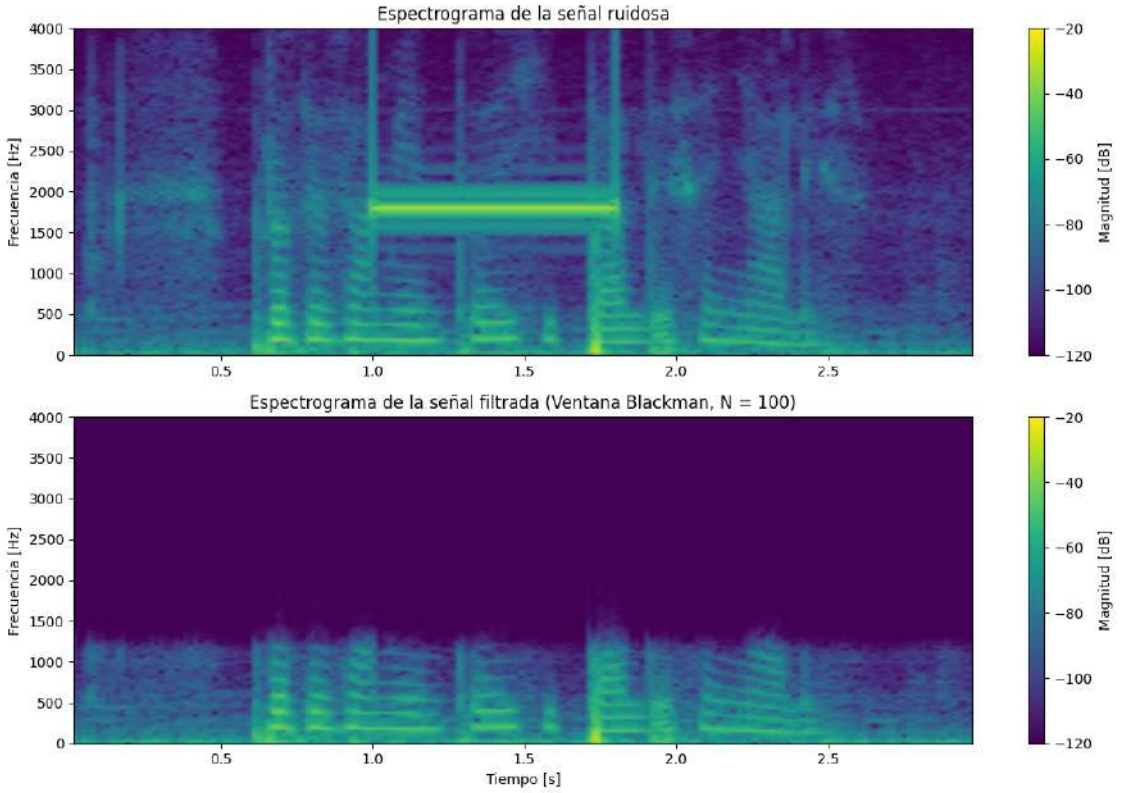


Figura 47. Espectrograma del Ejemplo 14: señal original (superior) vs señal filtrada (inferior), utilizando filtrado con ventana Blackman, N=100.

📌 **Observación importante:** al utilizar una ventana Blackman y un valor elevado de N, se logra una atenuación significativa de las componentes de alta frecuencia, especialmente por encima de 1200 Hz, lo que se refleja en la notable reducción del ruido observada en el espectrograma de la señal filtrada.

Adicionalmente, a diferencia de lo obtenido con el filtro de promedio, no se evidencian atenuaciones importantes dentro de la banda de paso, por lo que las componentes de interés de la señal de voz se conservan en mayor medida y se reduce el efecto de voz “ahogada”, característica de este tipo de filtros.

Ejemplo 15. Filtrado con filtro IIR Butterworth aplicando Transformada Bilineal

En este tercer ejemplo diseñamos un filtro IIR, específicamente un filtro Butterworth, aplicando la Transformada Bilineal. Es decir, partimos de un filtro definido en el dominio analógico, con función de transferencia (H(s)), y obtenemos un filtro digital con función de transferencia (H(z)), mediante una ecuación de mapeo entre los planos (s) y (z).

En este caso, utilizamos un filtro de orden (N=20), ya que este valor nos permite obtener una banda de transición angosta, es decir, una caída rápida entre la banda de paso y la banda de rechazo.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
from scipy.io import wavfile
from scipy.signal import spectrogram # Import spectrogram
from IPython.display import Audio, display # Import Audio and display

x_ruidosa_filepath = "audio_noisy.wav" # Lectura del audio  # Cambiar por el nombre del archivo
fc = 1200                              # Frecuencia de corte [Hz]
orden = 20                             # Orden del filtro Butterworth

x, fs = librosa.load(x_ruidosa_filepath, sr=None)
wc = 2 * np.pi * fc        # Frecuencia de corte analógica [rad/s]

b_a, a_a = signal.butter(orden, wc, btype="low",analog=True)
b_z, a_z = signal.bilinear(b_a, a_a, fs=fs)
x_filtrada_butterworth = signal.filtfilt(b_z, a_z, x)

f1, t1, Sxx1 = spectrogram(x, fs=fs, nperseg=256, noverlap=128) # espectrograma ruidosa
f2, t2, Sxx2 = spectrogram(x_filtrada_butterworth, fs=fs, nperseg=256, noverlap=128) # espectrograma filtrada
vmin, vmax = -120, -20
fig, ax = plt.subplots(2, 1, figsize=(12, 8))
im1 = ax[0].pcolormesh(t1, f1, 10*np.log10(Sxx1 + 1e-12),
                       shading='gouraud', vmin=vmin, vmax=vmax)
ax[0].set_title('Espectrograma de la señal ruidosa'); ax[0].set_ylabel('Frecuencia [Hz]')
ax[0].set_ylim([0, 4000])
cbar1 = fig.colorbar(im1, ax=ax[0]); cbar1.set_label('Magnitud [dB]')
im2 = ax[1].pcolormesh(t2, f2, 10*np.log10(Sxx2 + 1e-12),
                       shading='gouraud', vmin=vmin, vmax=vmax)
ax[1].set_title('Espectrograma de la señal filtrada (Butterworth, N = 20)')
ax[1].set_xlabel('Tiempo [s]'); ax[1].set_ylabel('Frecuencia [Hz]')
ax[1].set_ylim([0, 4000])
cbar2 = fig.colorbar(im2, ax=ax[1]); cbar2.set_label('Magnitud [dB]')

plt.tight_layout();plt.show()

print("Audio ruidoso:")
display(Audio(x, rate=fs)) # play audio ruidoso
print("Audio filtrado:")
display(Audio(x_filtrada_blackman, rate=fs)) # play audio filtrado
```

Figura 48. Código en Python del Ejemplo 15: diseño y aplicación de filtro Butterworth con Transformada Bilineal.

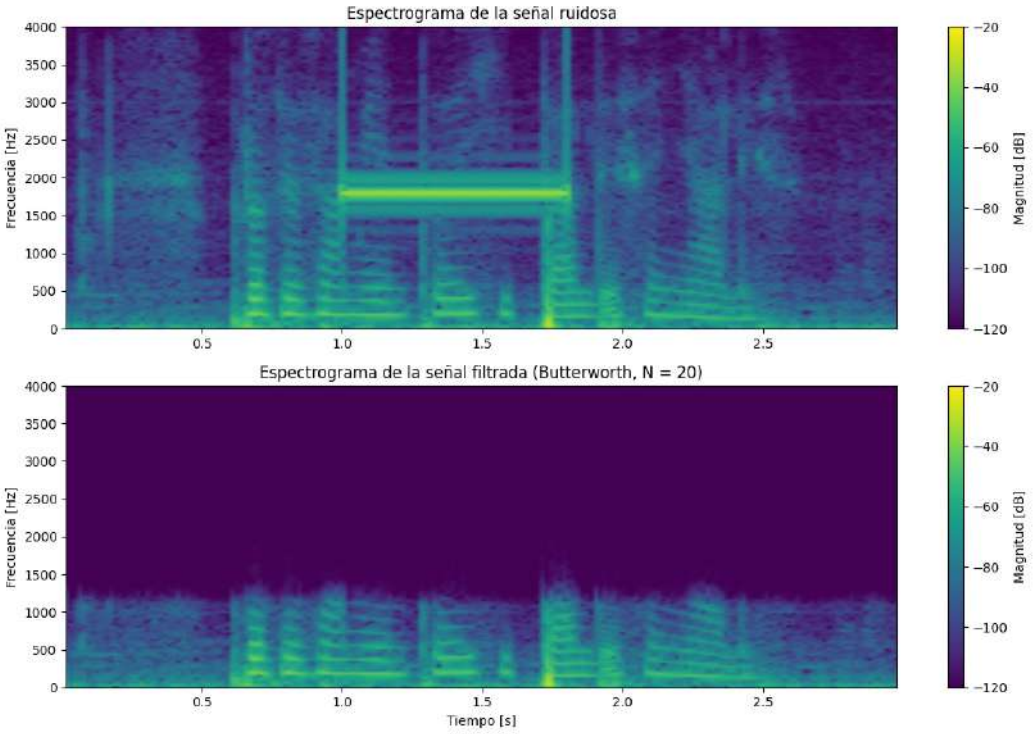


Figura 49. Espectrograma del Ejemplo 15: señal original (superior) vs señal filtrada (inferior), utilizando filtro Butterworth con Transformada Bilineal, N=20.

Podemos apreciar que el resultado obtenido con este filtro **IIR Butterworth de orden 20** es similar al alcanzado con el filtro **FIR diseñado mediante ventana Blackman de orden 100**. Sin embargo, una diferencia importante entre ambos enfoques es que los filtros IIR pueden lograr una respuesta altamente selectiva utilizando órdenes considerablemente menores. Esto se traduce en un menor número de coeficientes y operaciones, reduciendo el costo computacional asociado al proceso de filtrado.

Ejemplo 16. Filtrado mediante filtro Leaky

En este último ejemplo utilizaremos un **filtro Leaky**, uno de los filtros recursivos más simples que pueden implementarse en tiempo discreto. Su

ecuación entrada-salida está dada por:

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

donde λ es un parámetro comprendido entre 0 y 1 que controla el nivel de suavizado aplicado a la señal.

A diferencia de los filtros anteriores, el filtro *Leaky* no se diseña a partir de especificaciones de frecuencia de corte o banda de transición. En cambio, su comportamiento depende directamente del valor de λ. Cuando λ se aproxima a cero, la salida es muy similar a la señal original; por el contrario, cuando λ se aproxima a uno, el filtro produce un mayor suavizado y una mayor atenuación de las componentes de alta frecuencia.

Vamos a utilizar un valor de $\lambda = 0.9$, ya que valores cercanos a uno producen un comportamiento pasa-bajos más pronunciado. De esta forma, las componentes de alta frecuencia, incluido el tono interferente presente en la señal, son atenuadas en mayor medida que las componentes de baja frecuencia asociadas a la voz.

Sin embargo, antes de implementar el filtro, es conveniente obtener su función de transferencia. Partimos de la ecuación entrada-salida del filtro Leaky:

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

Reorganizando términos:

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

y aplicando TZ:

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

Factorizando $Y(z)$:

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

Por tanto, la función de transferencia del filtro Leaky es:

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

Comparando esta expresión con la forma general de un filtro digital,

$$H(z) = \frac{b_0 + b_1 z^{-1} + \cdots + b_M z^{-M}}{a_0 + a_1 z^{-1} + \cdots + a_N z^{-N}},$$

obtenemos que los coeficientes del filtro corresponden a:

$$b = [\,1-\lambda\,]$$

A partir del análisis anterior, nuestro código en Python queda:

```
import numpy as np
import matplotlib.pyplot as plt
import librosa
from scipy.signal import lfilter, spectrogram
from scipy.io.wavfile import write

x_ruidosa_filepath = "audio_noisy.wav"

x, fs = librosa.load(x_ruidosa_filepath, sr=None)

lambda_ = 0.98 # comprendido entre 0 y 1. Más cercano a 1 mayor el suavizado de la señal
b_leaky = [1 - lambda_]
a_leaky = [1, -lambda_]

x_filtrada_leaky = signal.filtfilt(b_leaky, a_leaky, x)

f1, t1, Sxx1 = spectrogram(x, fs=fs, nperseg=256, noverlap=128) # espectrograma ruidosa
f2, t2, Sxx2 = spectrogram(x_filtrada_leaky, fs=fs, nperseg=256, noverlap=128) # espectrograma filtrada
vmin, vmax = -120, -20
fig, ax = plt.subplots(2, 1, figsize=(12, 8))
im1 = ax[0].pcolormesh(t1, f1, 10*np.log10(Sxx1 + 1e-12),
                       shading='gouraud', vmin=vmin, vmax=vmax)
ax[0].set_title('Espectrograma de la señal ruidosa'); ax[0].set_ylabel('Frecuencia [Hz]')
ax[0].set_ylim([0, 4000])
cbar1 = fig.colorbar(im1, ax=ax[0]); cbar1.set_label('Magnitud [dB]')
im2 = ax[1].pcolormesh(t2, f2, 10*np.log10(Sxx2 + 1e-12),
                       shading='gouraud', vmin=vmin, vmax=vmax)
ax[1].set_title('Espectrograma de la señal filtrada (Leaky, lambda = 0.9)')
ax[1].set_xlabel('Tiempo [s]'); ax[1].set_ylabel('Frecuencia [Hz]')
ax[1].set_ylim([0, 4000])
cbar2 = fig.colorbar(im2, ax=ax[1]); cbar2.set_label('Magnitud [dB]')

plt.tight_layout();plt.show()

print("Audio ruidoso:")
display(Audio(x, rate=fs)) # play audio ruidoso
print("Audio filtrado:")
display(Audio(x_filtrada_blackman, rate=fs)) # play audio filtrado
```

Figura 50. Código en Python del Ejemplo 16: filtro Leaky, $\lambda = 0.98$.

los cuales son precisamente los coeficientes utilizados en la implementación en Python. Para este ejemplo se emplea $\lambda = 0.98$, valor que produce un comportamiento pasa-bajos pronunciado y permite atenuar de forma significativa las componentes de alta frecuencia presentes en la señal.

Obteniendo,

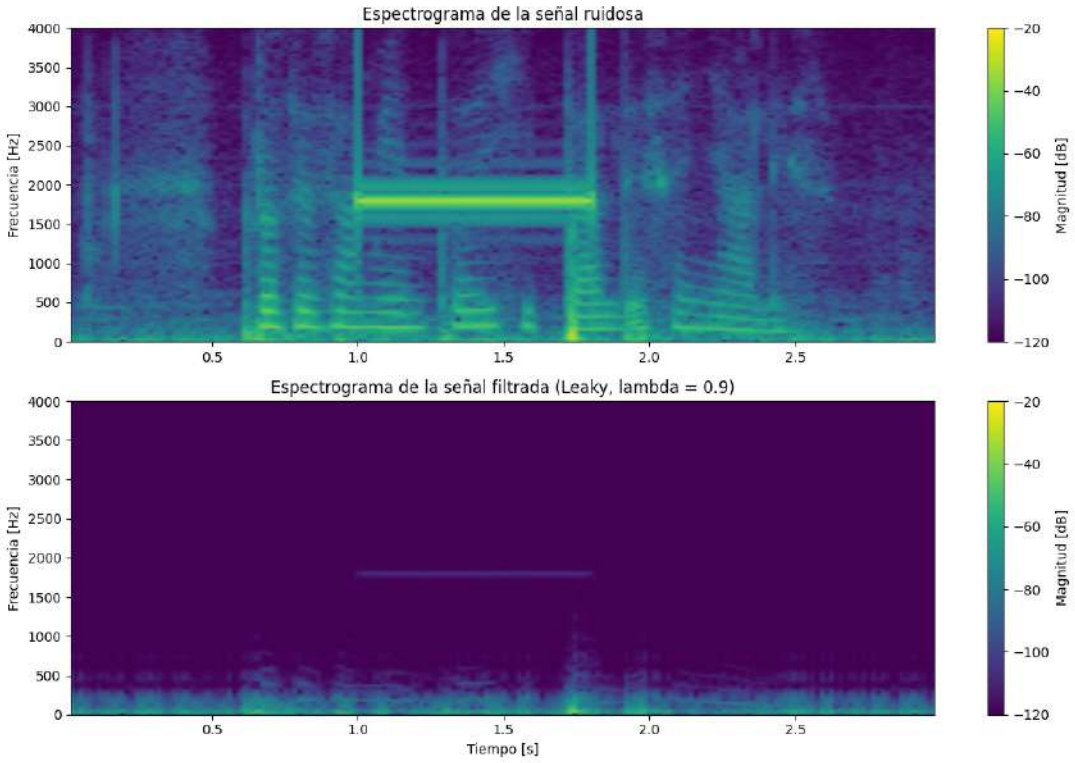


Figura 51. Espectrograma del Ejemplo 16: señal original (superior) vs señal filtrada (inferior), utilizando filtro Leaky, $\lambda = 0.98$.

Si comparamos este resultado con los filtros en los ejemplos anteriores, es decir, cuando filtramos la señal ruidosa con filtro de promedio, Blackman y Butterworth, podemos observar que con el filtro Leaky ($\lambda = 0.98$) obtenemos una atenuación mucho más agresiva de las componentes de frecuencia media y alta,

concentrando la energía casi exclusivamente por debajo de 500 Hz. No obstante, se conserva aún una pequeña parte de la energía del tono ubicado en 1800 Hz.

Ejemplo 17. Análisis comparativo de cuatro tipos de filtro para el caso de estudio: filtro de promedio, filtro con ventana Blackman, filtro Butterworth con Transformada Bilineal, y filtro Leaky

Típicamente, en otro libro, y en otro momento de mi vida, te habría propuesto evaluar la calidad de las señales filtradas utilizando como referencia el audio original sin ruido, presentado en el Ejemplo 11. Para ello, podríamos emplear métricas como la relación señal-ruido (SNR), el error cuadrático medio (MSE), la correlación entre señales (todas ellas calculadas en el dominio del tiempo) o incluso indicadores basados en los espectrogramas, como el índice de similitud estructural (SSIM).

Y no quiere decir que este enfoque sea incorrecto. De hecho, he utilizado este tipo de estrategias en diferentes proyectos de investigación, por ejemplo, en aplicaciones relacionadas con la inserción imperceptible de información dentro de señales de voz.

Sin embargo, en un contexto real rara vez disponemos de una versión “limpia” de la señal. En la mayoría de los casos, la única información disponible es la señal contaminada y las distintas alternativas de procesamiento que podrían aplicarse para mejorarla. Aun así, el ingeniero debe tomar una decisión y seleccionar la solución más adecuada.

Entonces surge una pregunta fundamental:
¿Cómo podemos comparar diferentes filtros cuando no disponemos de una señal de referencia?

Una primera opción consiste en evaluar qué tan bien cada filtro eliminó el tono de 1800 Hz, ya que ese era precisamente el objetivo del proceso de filtrado. Para ello, podemos analizar la energía residual observada en el espectrograma alrededor de dicha frecuencia. Cuanto menor sea esta energía, mayor habrá sido la capacidad del filtro para atenuar la interferencia.

Sin embargo, no podemos olvidar que el propósito no es únicamente eliminar el ruido, sino también preservar el mensaje de voz. De poco serviría eliminar por completo el tono si, al mismo tiempo, desaparece una parte importante de la energía asociada a la señal útil. Por esta razón, la conservación del contenido espectral de la voz constituye un segundo criterio de evaluación igualmente importante.

Criterio 1: Energía residual del tono de 1800 Hz

Tomamos una banda estrecha alrededor del tono, por ejemplo:
$\mathbf{1750} \leq f \leq \mathbf{1850}$ Hz

y calculamos:

$$E_{\text{tono}} = \sum_{t} \sum_{f=1750}^{1850} S(t, f)$$

Mientras menor sea E_{tono}, mejor filtrado del ruido.
Para ello, utilizamos el siguiente código en Python:

```python
import numpy as np
from scipy.signal import spectrogram

def energia_residual_tono(x, fs, f_tono=1800, ancho_banda=50):
    """
    Calcula la energía residual alrededor de un tono interferente
    usando el espectrograma de la señal.

    Parámetros:
    x : señal de audio
    fs : frecuencia de muestreo
    f_tono : frecuencia del tono interferente
    ancho_banda : ancho a cada lado de la frecuencia central

    Retorna:
    energia_tono : energía acumulada en la banda del tono
    """

    f, t, Sxx = spectrogram(x, fs=fs, nperseg=512, noverlap=256)
    f_min = f_tono - ancho_banda
    f_max = f_tono + ancho_banda
    idx_banda = (f >= f_min) & (f <= f_max)
    energia_tono = np.sum(Sxx[idx_banda, :])
    return energia_tono

y_promedio, fs = librosa.load("audio_promedio.wav", sr=None)
y_blackman, fs = librosa.load("audio_blackman.wav", sr=None)
y_butter, fs = librosa.load("audio_butterworth.wav", sr=None)
y_leaky, fs = librosa.load("audio_leaky.wav", sr=None)

E_promedio = energia_residual_tono(y_promedio, fs)
E_blackman = energia_residual_tono(y_blackman, fs)
E_butter = energia_residual_tono(y_butter, fs)
E_leaky = energia_residual_tono(y_leaky, fs)

print("Energía residual del tono de 1800 Hz")
print(f"Filtro de promedio: {E_promedio:.6e}")
print(f"Filtro Blackman: {E_blackman:.6e}")
print(f"Filtro Butterworth: {E_butter:.6e}")
print(f"Filtro Leaky: {E_leaky:.6e}")
```

Figura 52. Código en Python del Ejemplo 17:
criterio de evaluación de energía residual.

Y obtenemos,

Energía residual del tono de 1800 Hz
Filtro de promedio: 5.432817e-05
Filtro Blackman: 6.927825e-13
Filtro Butterworth: 6.287068e-13
Filtro Leaky: 1.518678e-09

Estos resultados muestran que los filtros Blackman y Butterworth presentan la menor energía residual alrededor de 1800 Hz, por lo que logran la mayor atenuación de la interferencia.

Criterio 2: Energía conservada de la señal de voz

Tomamos la banda donde queremos conservar la mayor parte de la voz, por ejemplo:

$$0 \leq f \leq 1200 \text{ Hz}$$

y calculamos:

$$E_{\text{voz}} = \sum_{t} \sum_{f=0}^{1200} S(t,f)$$

Mientras mayor sea E_{voz}, mejor preservación de la señal útil.

Para ello, utilizamos el siguiente código en Python:

```python
import numpy as np
from scipy.signal import spectrogram

def energia_banda_voz(x, fs, f_min=0, f_max=1200):
    """
    Calcula la energía conservada en la banda útil de la voz
    a partir del espectrograma de la señal.

    Parámetros:
    x : señal de audio
    fs : frecuencia de muestreo
    f_min : frecuencia mínima de la banda útil
    f_max : frecuencia máxima de la banda útil

    Retorna:
    energia_voz : energía acumulada en la banda de voz
    """

    f, t, Sxx = spectrogram(x, fs=fs, nperseg=512, noverlap=256)
    idx_banda = (f >= f_min) & (f <= f_max)
    energia_voz = np.sum(Sxx[idx_banda, :])
    return energia_voz

y_promedio, fs = librosa.load("audio_promedio.wav", sr=None)
y_blackman, fs = librosa.load("audio_blackman.wav", sr=None)
y_butter, fs = librosa.load("audio_butterworth.wav", sr=None)
y_leaky, fs = librosa.load("audio_leaky.wav", sr=None)

Evoz_promedio = energia_banda_voz(y_promedio, fs)
Evoz_blackman = energia_banda_voz(y_blackman, fs)
Evoz_butter = energia_banda_voz(y_butter, fs)
Evoz_leaky = energia_banda_voz(y_leaky, fs)

print("Energía conservada en la banda útil de la voz")
print(f"Filtro de promedio: {Evoz_promedio:.6e}")
print(f"Filtro Blackman: {Evoz_blackman:.6e}")
print(f"Filtro Butterworth: {Evoz_butter:.6e}")
print(f"Filtro Leaky: {Evoz_leaky:.6e}")
```

Figura 53. Código en Python del Ejemplo 17: criterio de evaluación de conservación de energía.

Y obtenemos,

Energía conservada en la banda útil de la voz
Filtro de promedio: 7.453849e-03
Filtro Blackman: 8.685923e-03
Filtro Butterworth: 8.682669e-03
Filtro Leaky: 7.247906e-04

En este caso, nuevamente los mejores resultados corresponden a los filtros Blackman y Butterworth, con valores muy cercanos entre sí. En una tercera posición se encuentra el filtro de promedio, mientras que el filtro Leaky presenta la menor conservación de energía en la banda útil de la voz.

Ahora surge una pregunta natural: ¿cómo podemos combinar ambos criterios?

Una opción sencilla consiste en asignar puntuaciones por puesto, como en una competencia olímpica: medalla de oro, plata y bronce. La medalla de oro recibe 3 puntos, la de plata 2 puntos y la de bronce 1 punto, para cada criterio evaluado. La idea es que no gane necesariamente el filtro que mejor elimina el tono, sino aquel que logra el mejor balance entre atenuar la interferencia y conservar la información útil de la voz.

Para el criterio 1, la puntuación queda así:

- Medalla de oro: filtros Blackman y Butterworth, debido a que sus resultados son muy similares entre sí. **3 puntos**.
- Medalla de plata: filtro Leaky. **2 puntos**.
- Medalla de bronce: filtro de promedio. **1 punto**.

Para el criterio 2, la puntuación queda así:

- Medalla de oro: filtros Blackman y Butterworth, nuevamente con resultados muy cercanos entre sí. **3 puntos**.
- Medalla de plata: filtro de promedio. **2 puntos**.
- Medalla de bronce: filtro Leaky. **1 punto**.

Por tanto, la puntuación final es:

- Medalla de oro: filtros Blackman y Butterworth. **6 puntos**.
- Medalla de plata: filtros de promedio y Leaky. **3 puntos**.

Esto indica que, para este caso de estudio, los filtros Blackman y Butterworth permitieron la mejor calidad global, pues atenuaron de manera significativa el tono de 1800 Hz y, al mismo tiempo, conservaron gran parte de la energía útil de la señal de voz.

Sin embargo, si ambos filtros ofrecen un desempeño similar, aparece una nueva pregunta de ingeniería:

¿cómo desempatamos?

Teniendo en cuenta que el filtro Blackman fue diseñado con $N = 100$, mientras que el filtro Butterworth utilizó un orden $N = 20$, el filtro Butterworth alcanza un excelente desempeño con un menor costo computacional. Por esta razón, el filtro Butterworth es el ganador dado que logra el mejor equilibrio entre calidad de la señal filtrada y eficiencia computacional.

Bueno, y ahora viene una imagen que espero que no revises únicamente mientras cursas esta asignatura de PDS, ni tampoco solo durante la primera lectura de este libro si no eres mi estudiante. Espero que vuelvas a ella cuantas veces lo consideres necesario. Más que una simple figura, representa una metodología de comparación y selección de soluciones. Aunque en este capítulo se encuentra aplicada al problema de filtrado de una señal de voz, los pasos que la conforman pueden adaptarse a otros problemas de filtrado e incluso a diferentes aplicaciones de procesamiento de audio.

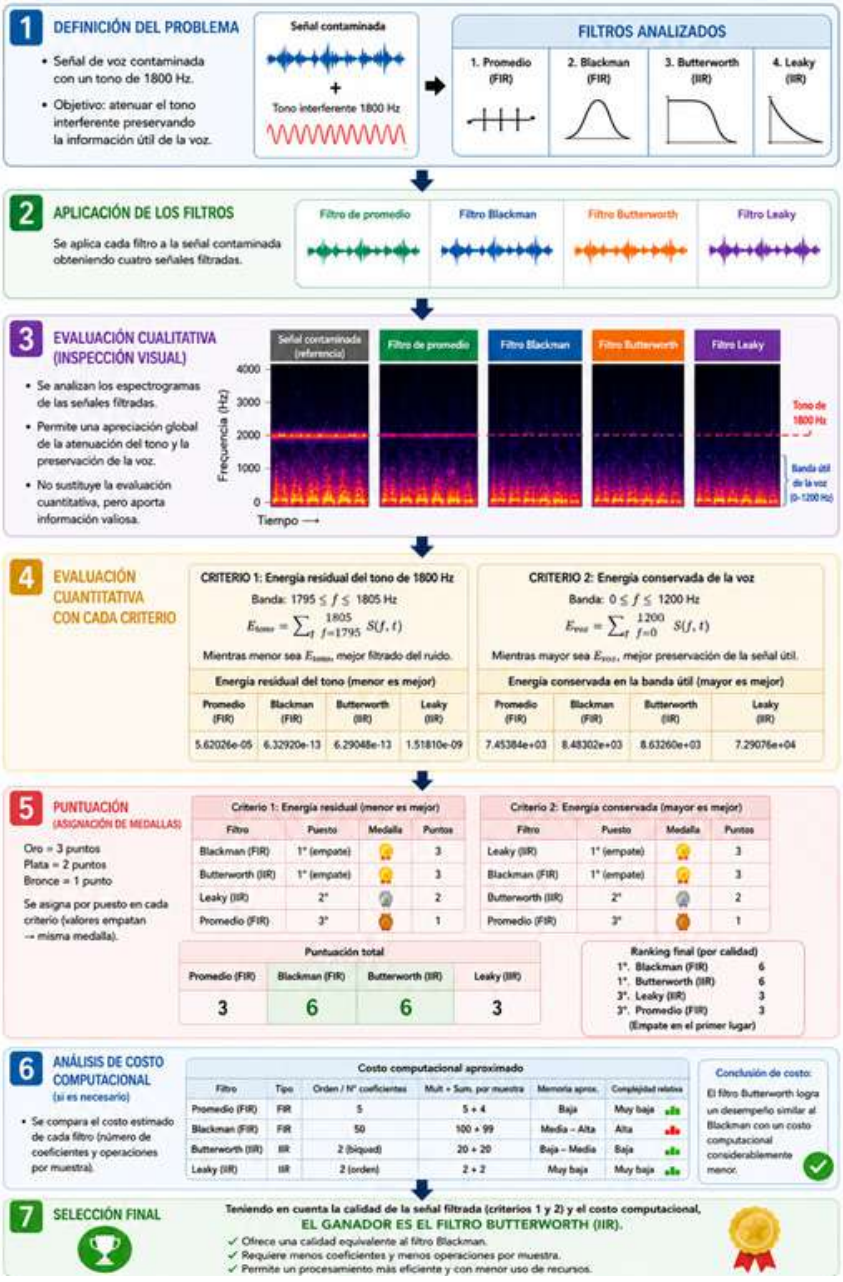


Figura 54. Metodología propuesta para la evaluación, comparación y selección de soluciones de filtrado de una señal de voz. Aplicación al caso de estudio presentado en este capítulo.

Mi invitación es a que no te quedes únicamente con los resultados obtenidos en este caso de estudio, sino que te inspires en el procedimiento seguido para abordar nuevos problemas, definir criterios de evaluación, comparar alternativas y seleccionar la solución más adecuada para cada contexto.

CAPÍTULO III

IA PARA LOCALIZACIÓN DE INTERFERENCIAS Y FILTRADO ESPECTRAL

Cuando escuchamos hablar de Inteligencia Artificial (IA), con frecuencia pensamos en algoritmos capaces de aprender automáticamente a partir de los datos. Sin embargo, no todos los métodos de aprendizaje funcionan de la misma manera. Dos de las estrategias más utilizadas son el aprendizaje supervisado y el aprendizaje no supervisado.

- **Aprendizaje supervisado:** requiere disponer de parejas entrada–salida. En nuestro caso, esto significaría contar con señales de voz contaminadas con ruido y sus correspondientes versiones limpias. A partir de estos ejemplos, el modelo aprende a identificar patrones relacionados con la presencia de la interferencia y a generar una versión mejorada de la señal. Una vez finalizado el entrenamiento, el modelo puede utilizarse para procesar nuevos audios que nunca había visto anteriormente.
- **Aprendizaje no supervisado:** no requiere disponer de señales limpias. El algoritmo trabaja únicamente con los datos disponibles e intenta descubrir patrones, agrupaciones o comportamientos anómalos presentes en ellos.

Para el caso de estudio de este capítulo, correspondiente al filtrado de un tono interferente presente en una señal de voz, no disponemos de una base

de datos suficientemente grande de señales limpias y contaminadas que permita entrenar adecuadamente un modelo supervisado. Por esta razón, utilizaremos una estrategia de aprendizaje no supervisado.

La IA no filtra la señal, localiza la interferencia

Antes de continuar, es importante aclarar cuál será el papel de la IA dentro de nuestra solución (Figura 55).

- Partimos del mismo audio contaminado con ruido tonal utilizado en el Capítulo 2. El primer paso consiste en transformar la señal desde el dominio del tiempo al dominio tiempo-frecuencia mediante el cálculo de su espectrograma. En este caso, el espectrograma no se utiliza únicamente como una herramienta de visualización, sino como el espacio de trabajo donde operará nuestro bloque de Inteligencia Artificial.

- A continuación, la IA analizará la información espectral con el objetivo de identificar automáticamente las componentes asociadas a la interferencia tonal. Para ello utilizaremos dos algoritmos de aprendizaje no supervisado: **K-means** y **DBSCAN**.

- Una vez localizada la región espectral asociada al ruido, construiremos una máscara de atenuación correspondiente a un filtro *notch* (rechaza-banda altamente selectivo). Esta máscara será aplicada directamente sobre el espectrograma para reducir la energía de la interferencia.

- Finalmente, mediante la transformada inversa correspondiente, reconstruiremos la señal en el dominio del tiempo, obteniendo una nueva versión de la señal de voz con una menor presencia del tono interferente.

En resumen, la IA responde a la pregunta: **¿Dónde está el ruido?** mientras que el Procesamiento Digital de Señales responde a la pregunta: **¿Cómo lo eliminamos?**

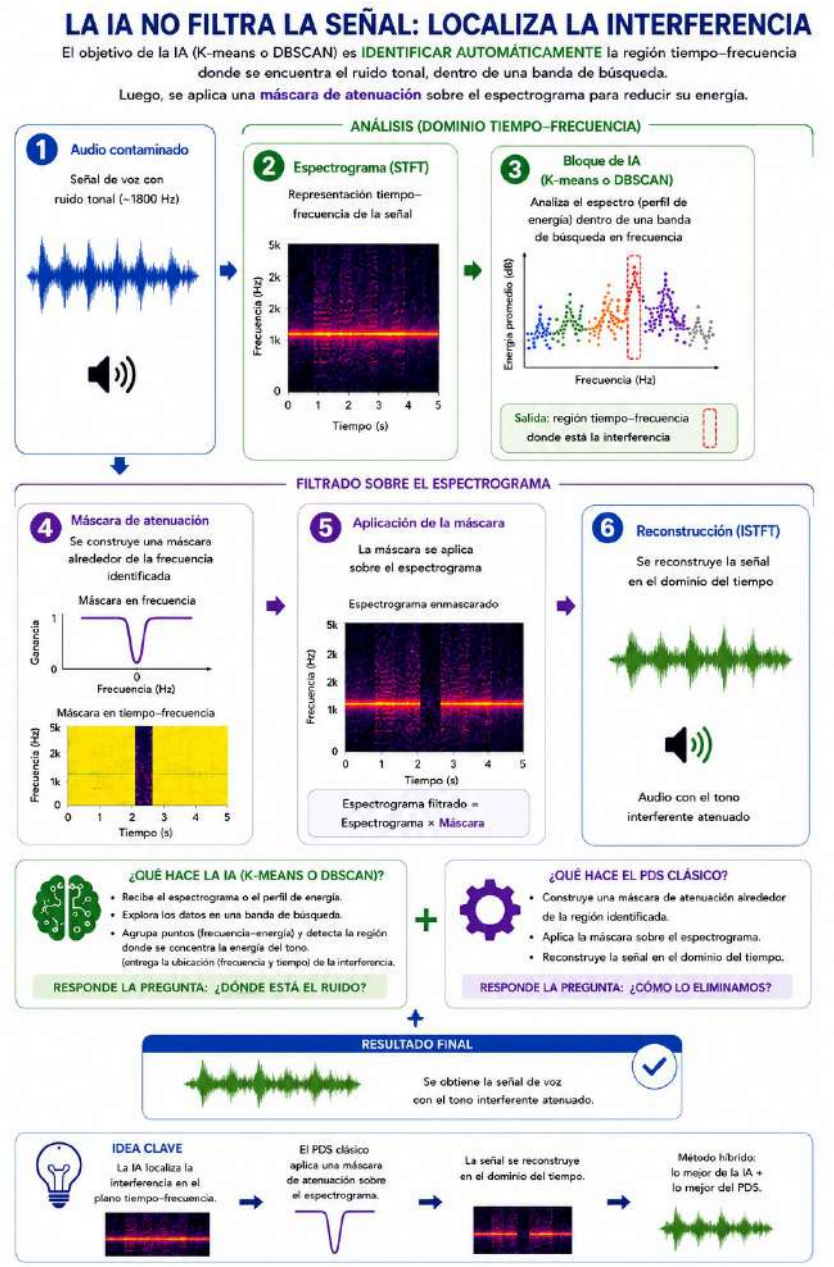


Figura 55. Integración de la IA no supervisada en el filtrado de ruido tonal en señal de voz.

Esta combinación permite aprovechar las fortalezas de ambos enfoques: la capacidad de la IA para descubrir patrones en los datos y la capacidad de los métodos clásicos para modificar selectivamente el contenido espectral de la señal.

A continuación, comprenderemos cada uno de los dos métodos propuestos para el bloque de localización de ruido y su correspondiente solución.

3.1 Algoritmo K-means (aprendizaje no supervisado)

El algoritmo K-means tiene como objetivo particionar un conjunto de datos en K grupos, de tal forma que los elementos de cada grupo sean lo más similares posible entre sí y lo más diferentes posible respecto a los demás grupos.

La idea es sencilla. Supongamos que disponemos de un conjunto de datos y queremos organizarlos en K grupos diferentes. El algoritmo analiza las características de cada dato y los asigna al grupo que considere más parecido. Como resultado, se obtienen K agrupaciones, cada una representada por un valor central denominado **centroide**.

En aplicaciones tradicionales, *K-means* puede utilizarse para segmentar clientes, agrupar documentos o identificar patrones de comportamiento. En procesamiento digital de señales, también puede emplearse para encontrar regiones del espectrograma con características similares.

Ejemplo 18. Algoritmo K-means aplicado al filtrado de ruido tonal presente en señal de voz

En nuestra implementación, K-means agrupa los puntos del espectrograma ***utilizando únicamente su energía espectral*** como característica de entrada. El algoritmo calcula dos centroides, uno para cada grupo, y de esta forma dependiendo de la proximidad de uno u otro, cada punto se clasifica en uno de los dos grupos.

La hipótesis de trabajo es que la energía asociada al tono de ruido es diferente de la energía asociada a la señal de voz. Como el tono concentra gran parte de su energía en una frecuencia específica, mientras que la voz distribuye su energía en un rango amplio de frecuencias, esperamos que ambos comportamientos puedan ser separados mediante K-means.

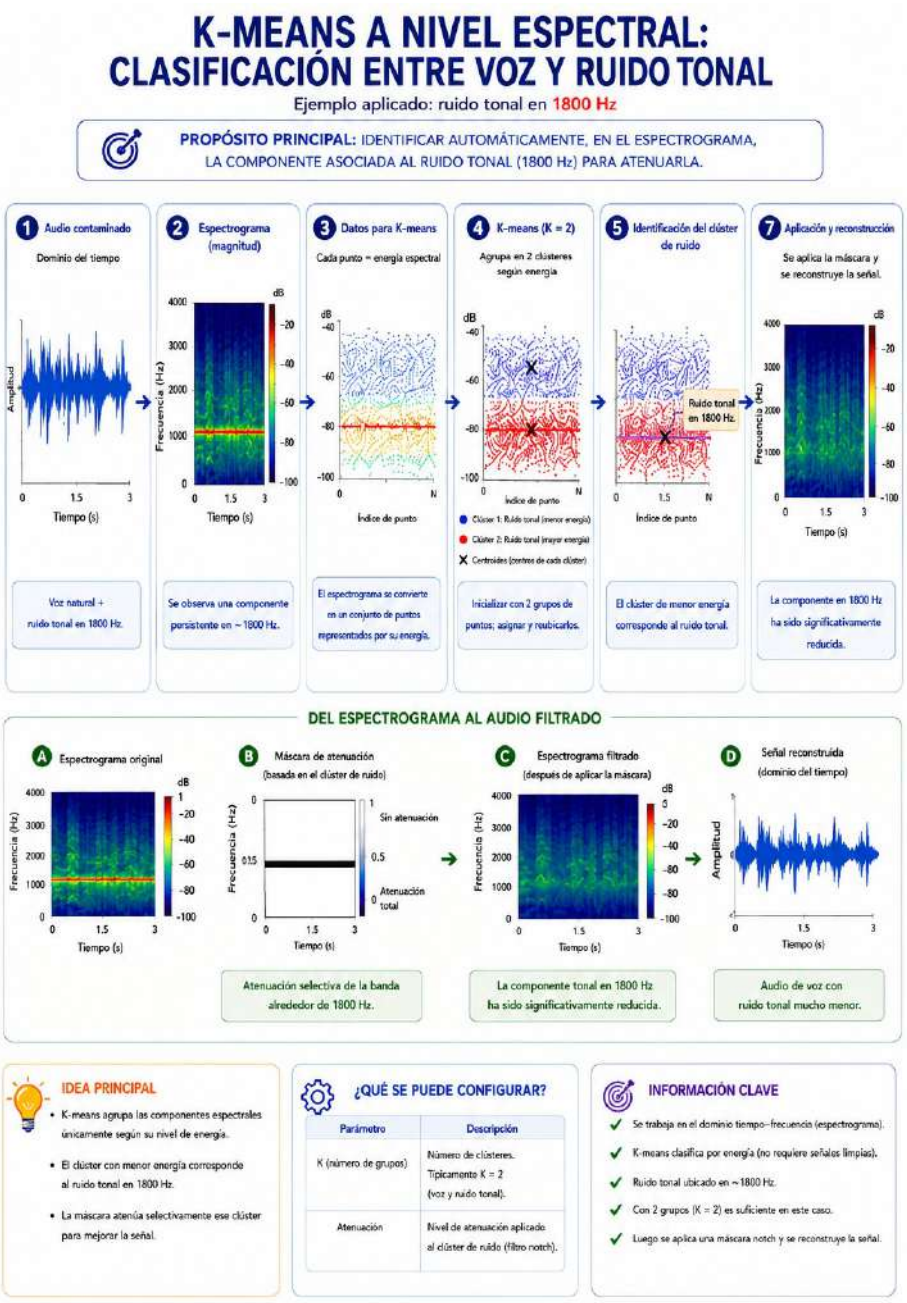


Figura 56. Aplicación de K-means para la eliminación de tono a partir del espectrograma de una señal de voz.

Los pasos son los siguientes:

- Paso 1: Partimos de una señal de voz contaminada con un ruido tipo tono.
- Paso 2: Calculamos el espectrograma de la señal.
- Paso 3: Utilizamos la energía de cada punto del espectrograma como entrada para *K-means*.
- Paso 4: Aplicamos *K-means* con K=2. El algoritmo agrupa automáticamente los puntos con niveles de energía similares.
- Paso 5: Identificamos el grupo con mayor energía, asumiendo inicialmente que corresponde al ruido tonal.
- Paso 6: Construimos una máscara de atenuación para reducir la energía de dicho grupo y conservar el resto de las componentes espectrales.
- Paso 7: Aplicamos la máscara sobre el espectrograma y reconstruimos la señal filtrada en el dominio del tiempo.

Una vez comprendido el procedimiento, la siguiente inquietud es:

¿qué parámetros podemos ajustar?

En esta implementación existen dos parámetros principales que podemos ajustar:

- **Número de grupos (K):** en nuestro caso utilizamos $K = 2$, bajo la hipótesis de que un grupo corresponde principalmente a la señal de voz y el otro al ruido tonal.
- **Nivel de atenuación:** determina cuánto se reduce la energía del grupo identificado como posible interferencia.

Vamos entonces con el código:

```python
import numpy as np
import matplotlib.pyplot as plt
import soundfile as sf
from scipy.signal import stft, istft, spectrogram
from sklearn.cluster import KMeans

x, fs = sf.read("audio_noisy.wav")
f, t, Zxx = stft(x, fs=fs, window="hann", nperseg=1024, noverlap=768)
S = np.abs(Zxx)
S_db = 20*np.log10(S + 1e-12)
X = S_db.reshape(-1, 1)

K = 2
kmeans = KMeans(n_clusters=K, random_state=0, n_init=10)
labels = kmeans.fit_predict(X)
labels_2D = labels.reshape(S_db.shape)

centros = kmeans.cluster_centers_.flatten()
cluster_ruido = np.argmax(centros)

print("Centroides encontrados:")
for i, c in enumerate(centros):
    print(f"Grupo {i}: {c:.2f} dB")
print(f"Grupo seleccionado como ruido: {cluster_ruido}")

mascara = np.ones_like(S)
mascara[labels_2D == cluster_ruido] = 0.05
Zxx_filtrado = Zxx * mascara
_, x_filtrada_kmeans = istft(Zxx_filtrado, fs=fs, window="hann",
                             nperseg=1024, noverlap=768)
x_filtrada_kmeans = x_filtrada_kmeans[:len(x)]
x_filtrada_kmeans = x_filtrada_kmeans / np.max(np.abs(x_filtrada_kmeans))
```

Figura 57. Código en Python del Ejemplo 18: algoritmo K-means para el filtrado de ruido tipo tono en señal de voz.

** Nota aclaratoria: el código anterior no incluye la parte de dibujo de los espectrogramas, por temas de espacio. Se puede utilizar el mismo texto correspondiente al de la Figura 50.*

Sin embargo, existe una limitación importante que aún no hemos abordado. En esta implementación, la agrupación se realiza únicamente a partir de la energía espectral; es decir, no se utiliza de forma explícita ni la ubicación temporal ni la ubicación frecuencial de cada componente. Como consecuencia, componentes de voz con niveles de energía similares a los del tono podrían

terminar siendo agrupadas dentro del mismo clúster, afectando el resultado del filtrado.

Pero antes de sacar conclusiones, observemos cómo queda el espectrograma de la señal filtrada.

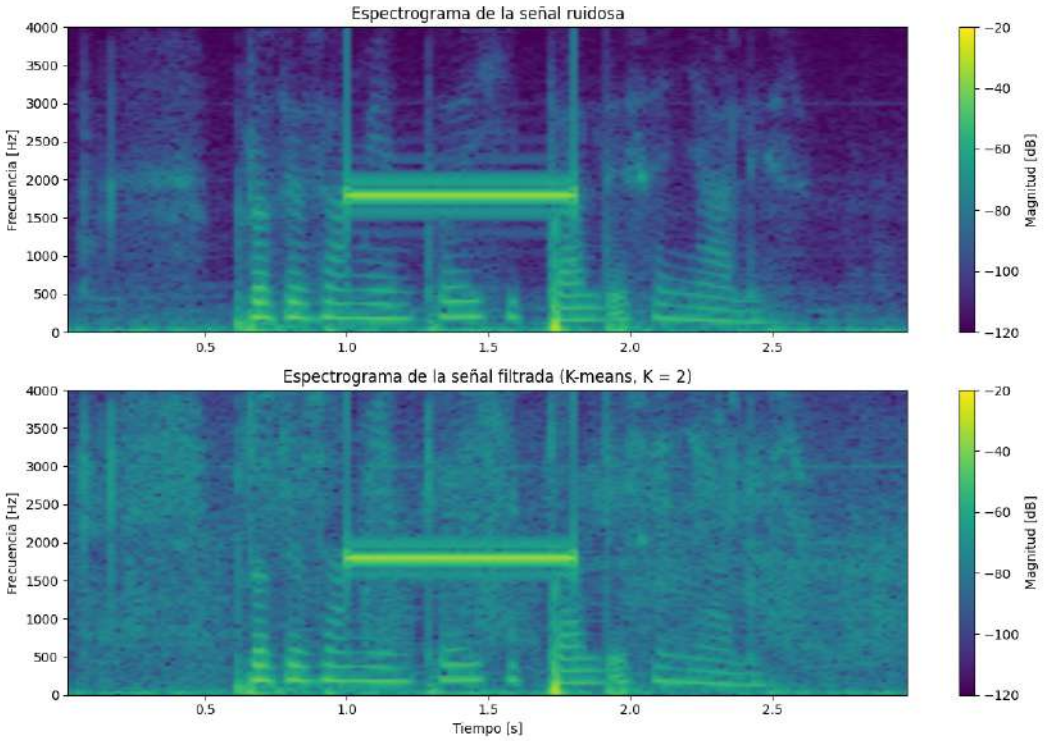


Figura 58. Espectrograma del Ejemplo 18: algoritmo K-means para el filtrado de ruido tipo tono en señal de voz.

Centroides encontrados:
Grupo 0: -97.68 dB
Grupo 1: -72.60 dB
Grupo seleccionado como ruido: 1

Al observar el espectrograma de la señal filtrada, encontramos un resultado diferente al esperado. Los centroides obtenidos por *K-means* fueron **−97.68 dB** para el Grupo 0 (azul oscuro) y **−72.60 dB** para el Grupo 1 (azul

verdoso). Como el Grupo 1 presenta mayor energía, fue seleccionado como posible ruido y posteriormente atenuado.

Es por esta razón que el fondo azul oscuro presente en el espectrograma original prácticamente desaparece en el espectrograma de la señal filtrada. Sin embargo, el tono no fue eliminado, ya que quedó clasificado dentro del mismo grupo que muchas de las componentes de la señal de voz. Esto tiene sentido si observamos que la señal de voz también presenta componentes de alta energía, especialmente en las frecuencias bajas, alcanzando niveles comparables a los del tono de ruido.

Así que, aunque K-means logró identificar dos centroides, separar los datos en dos grupos y atenuar uno de ellos, una clasificación basada únicamente en la energía espectral no resulta suficiente para diferenciar adecuadamente entre ruido tonal y señal de voz. Se requiere incorporar información adicional que permita describir mejor el comportamiento de la interferencia dentro del espectrograma.

Por esta razón, a continuación, exploraremos una estrategia alternativa que utilizará simultáneamente información de frecuencia y energía, permitiendo identificar con mayor precisión el tono que deseamos eliminar.

3.2 Algoritmo DBSCAN (aprendizaje no supervisado)

¿Qué aprendimos de la solución anterior con K-means?

Que agrupar únicamente por energía espectral no fue suficiente para separar correctamente el tono correspondiente a ruido de la señal de voz. Aunque el algoritmo logró identificar dos grupos con diferentes niveles de energía, el tono permaneció dentro del mismo grupo que muchas componentes útiles de la voz.

¿Y entonces qué podemos hacer?

Utilizaremos una segunda técnica de aprendizaje no supervisado denominada **DBSCAN** (*Density-Based Spatial Clustering of Applications with Noise*).

Ejemplo 19. Algoritmo DBSCAN aplicado al filtrado de ruido tonal presente en señal de voz

A diferencia de *K-means*, **DBSCAN** no requiere especificar previamente el número de grupos que se desea encontrar (es decir, pueden ser más de dos y el ruido pertenecer a uno de ellos). En su lugar, analiza la distribución de los datos y busca regiones donde existe una alta concentración de puntos, formando automáticamente agrupaciones o *clusters*. Adicionalmente, los puntos que no pertenecen claramente a ningún grupo son clasificados como ruido o anomalías.

Esta característica resulta especialmente adecuada para nuestro caso de estudio, ya que no conocemos cuántos patrones espectrales diferentes existen dentro de la señal y, además, nos interesa detectar componentes que se comporten de forma distinta al resto.

La Figura 59 presenta una explicación conceptual de DBSCAN aplicada a nuestro caso de estudio.

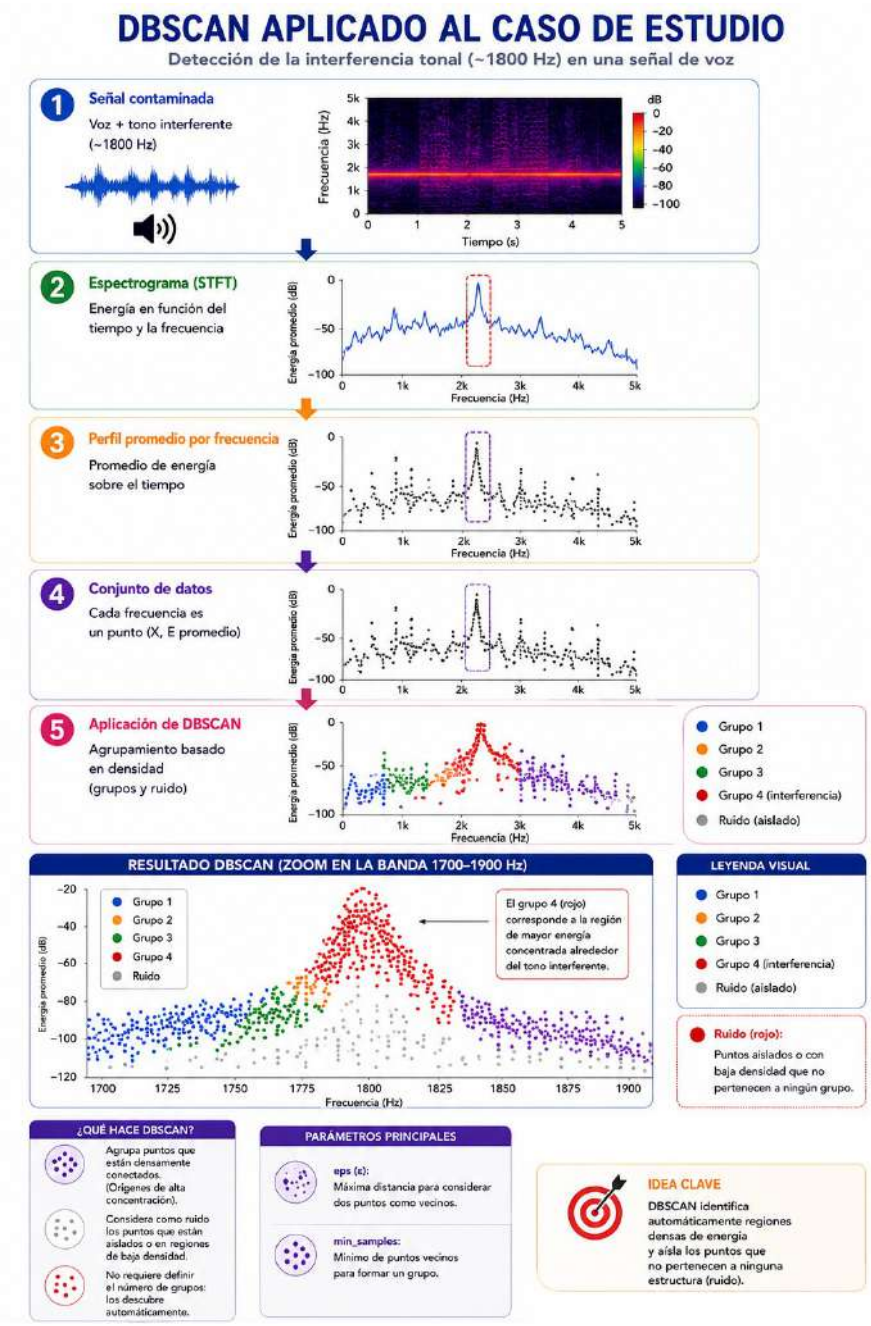


Figura 59. Aplicación de DBSCAN para la eliminación de tono a partir del espectrograma de una señal de voz.

Los pasos del método son los siguientes:

- **Paso 1:** Partimos de una señal de voz contaminada con un ruido tipo tono.
- **Paso 2:** Calculamos el espectrograma de la señal para representar la distribución de energía en función del tiempo y la frecuencia.
- **Paso 3:** Calculamos el perfil promedio de energía por frecuencia, promediando la energía espectral a lo largo del tiempo. De esta forma obtenemos una representación más compacta de la señal.
- **Paso 4:** Construimos el conjunto de datos para DBSCAN. Cada frecuencia se representa mediante un punto cuyas características son la frecuencia y la energía promedio asociada a ella.
- **Paso 5:** Aplicamos DBSCAN para identificar automáticamente grupos de puntos con comportamientos similares. A diferencia de K-means, no es necesario indicar previamente cuántos grupos deben encontrarse.
- **Paso 6:** Analizamos los grupos obtenidos e identificamos aquel que concentra la mayor energía dentro de la banda de interés, ya que corresponde a un posible tono interferente.
- **Paso 7:** Estimamos la frecuencia del tono a partir del punto de máxima energía dentro del grupo seleccionado.
- **Paso 8:** Construimos una máscara de atenuación alrededor de la frecuencia estimada y la aplicamos sobre el espectrograma.
- **Paso 9:** Finalmente, reconstruimos la señal en el dominio del tiempo obteniendo una nueva versión con una menor presencia del tono interferente.

Es importante mencionar que DBSCAN no conoce qué componentes corresponden a voz y cuáles a ruido. Su función consiste únicamente en **identificar agrupaciones dentro de los datos**.

La interpretación de los resultados y la decisión sobre qué componentes deben ser atenuadas siguen dependiendo del conocimiento previo que tengamos sobre la señal y la interferencia. Por esta razón, se define una banda de búsqueda donde se espera encontrar el ruido, concentrando el análisis en la región espectral de interés y evitando confusiones con componentes propias de la voz.

El aprendizaje no supervisado no elimina la necesidad de conocer el problema; simplemente elimina la necesidad de disponer de ejemplos etiquetados.

A continuación, el código para DBSCAN:

```python
import numpy as np
import matplotlib.pyplot as plt
import soundfile as sf
from scipy.signal import stft, istft, spectrogram
from sklearn.cluster import DBSCAN
from sklearn.preprocessing import StandardScaler

x, fs = sf.read("audio_noisy.wav")
f, t, Zxx = stft(x, fs=fs, window="hann", nperseg=1024, noverlap=768)
S = np.abs(Zxx)
S_db = 20*np.log10(S + 1e-12)

# Perfil promedio de energía por frecuencia
perfil_frecuencia = np.mean(S_db, axis=1)

# Analizar solo una banda donde puede estar el tono
idx_busqueda = (f >= 500) & (f <= 4000)

f_roi = f[idx_busqueda]
E_roi = perfil_frecuencia[idx_busqueda]

# Datos para DBSCAN: frecuencia + energía
X = np.column_stack((f_roi, E_roi))
X_scaled = StandardScaler().fit_transform(X)

# DBSCAN
dbscan = DBSCAN(eps=0.30, min_samples=5)
labels = dbscan.fit_predict(X_scaled)

print("Grupos encontrados:", np.unique(labels))

# Identificar grupo candidato a ruido:
# tomamos el grupo con mayor energía promedio, excluyendo etiqueta -1
grupos = np.unique(labels)
grupos_validos = grupos[grupos != -1]


if len(grupos_validos) > 0:
    energia_grupos = []
    for g in grupos_validos:
        energia_grupos.append(np.mean(E_roi[labels == g]))

    grupo_ruido = grupos_validos[np.argmax(energia_grupos)]
    idx_grupo_ruido = labels == grupo_ruido
    idx_max = np.argmax(E_roi)
    f_ruido = f_roi[idx_max]
    print(f"Frecuencia interferente estimada: {f_ruido:.2f} Hz")

else:
    print("No se encontraron grupos válidos para el ruido.")

print(f"Frecuencia interferente estimada: {f_ruido:.2f} Hz")

# Máscara tipo notch guiada por DBSCAN
#ancho_banda = 80
ancho_banda = 100
#atenuacion = 0.005
atenuacion = 0.001

mascara = np.ones_like(S)

idx_ruido = (f >= f_ruido - ancho_banda) & (f <= f_ruido + ancho_banda)
mascara[idx_ruido, :] = atenuacion

Zxx_filtrado = Zxx * mascara

_, x_filtrada_dbscan = istft(Zxx_filtrado, fs=fs, window="hann",
                             nperseg=1024, noverlap=768)

x_filtrada_dbscan = x_filtrada_dbscan[:len(x)]
x_filtrada_dbscan = x_filtrada_dbscan / np.max(np.abs(x_filtrada_dbscan))

sf.write("audio_filtrado_dbscan.wav", x_filtrada_dbscan, fs)
```

```python
# Espectrogramas
f1, t1, Sxx1 = spectrogram(x, fs=fs, nperseg=256, noverlap=128)
f2, t2, Sxx2 = spectrogram(x_filtrada_dbscan, fs=fs, nperseg=256, noverlap=128)

vmin, vmax = -120, -20
fig, ax = plt.subplots(2, 1, figsize=(12, 8))

im1 = ax[0].pcolormesh(t1, f1, 10*np.log10(Sxx1 + 1e-12),
                       shading='gouraud', vmin=vmin, vmax=vmax)
ax[0].set_title('Espectrograma de la señal ruidosa')
ax[0].set_ylabel('Frecuencia [Hz]')
ax[0].set_ylim([0, 4000])
cbar1 = fig.colorbar(im1, ax=ax[0])
cbar1.set_label('Magnitud [dB]')

im2 = ax[1].pcolormesh(t2, f2, 10*np.log10(Sxx2 + 1e-12),
                       shading='gouraud', vmin=vmin, vmax=vmax)
ax[1].set_title('Espectrograma de la señal filtrada (DBSCAN)')
ax[1].set_xlabel('Tiempo [s]')
ax[1].set_ylabel('Frecuencia [Hz]')
ax[1].set_ylim([0, 4000])
cbar2 = fig.colorbar(im2, ax=ax[1])
cbar2.set_label('Magnitud [dB]')
plt.tight_layout()
plt.show()

# Visualización de DBSCAN
plt.figure(figsize=(8, 4))
plt.scatter(f_roi, E_roi, c=labels, s=20)
plt.axvline(f_ruido, linestyle='--', label=f'f estimada = {f_ruido:.1f} Hz')
plt.xlabel('Frecuencia [Hz]')
plt.ylabel('Energía promedio [dB]')
plt.title('Agrupamiento DBSCAN sobre frecuencia y energía')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Figura 60. Código en Python del Ejemplo 19: algoritmo DBSCAN para identifica ruido tonal en señal de voz.

Inicialmente, vamos a suponer que no le acotamos al algoritmo la zona de búsqueda del tono y de decimos que **idx_busqueda = (f >= 0) & (f <= 4000)**, entonces obtenemos el siguiente agrupamiento sobre frecuencia y energía:

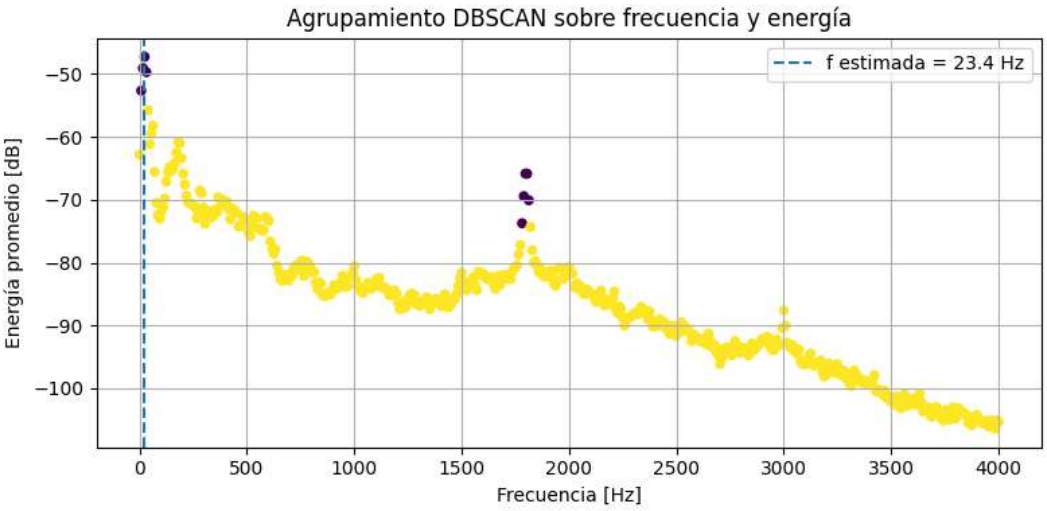


Figura 61. Agrupamiento DBSCAN del Ejemplo 19 utilizando una banda de búsqueda entre 0 y 4000 Hz. El algoritmo estima una frecuencia interferente cercana a 23 Hz.

¿Qué podemos identificar en la Figura 61?

Observamos que las componentes de muy baja frecuencia presentan niveles de energía incluso mayores que los asociados al tono de ruido. Este resultado ayuda a comprender por qué *K-means* no funcionó adecuadamente en el ejemplo anterior: al utilizar únicamente la energía como criterio de agrupamiento, las componentes de voz con alta energía terminan dominando el proceso de clasificación. Sin embargo, un conocedor de Procesamiento Digital de Señales identificaría que el segundo máximo local de energía, ubicado alrededor de 1800 Hz, corresponde al tono interferente que deseamos eliminar.

Simplemente, vamos a decir que la voz principalmente se encuentra entre 0 y 500 Hz, y que esa zona no la vamos a afectar, entonces: **idx_busqueda = (f >= 500) & (f <= 4000)**, y obtenemos:

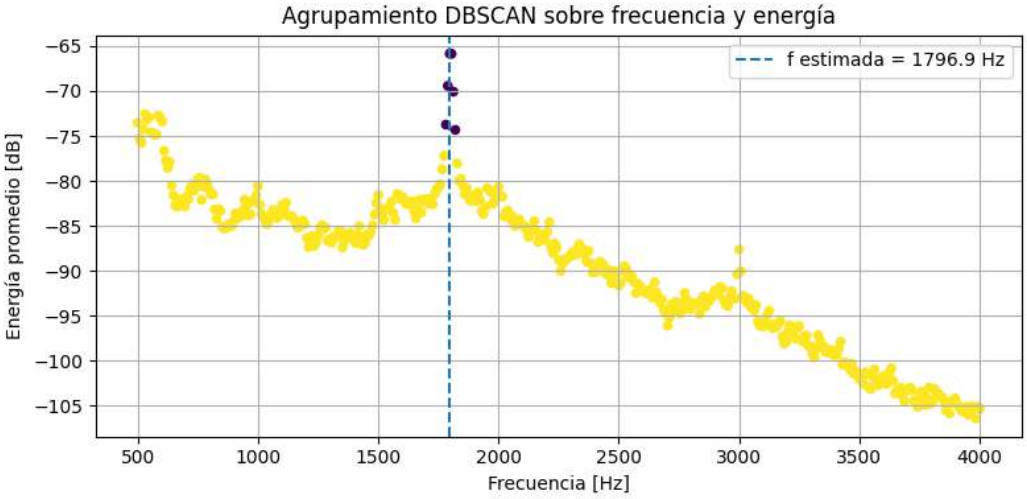


Figura 62. Agrupamiento DBSCAN del Ejemplo 19 utilizando una banda de búsqueda entre 500 y 4000 Hz. El algoritmo estima una frecuencia de interferencia de 1797 Hz, aproximadamente.

En este caso, el algoritmo DBSCAN estimó que la frecuencia correspondiente al ruido es de 1796.9 Hz, valor muy cercano al real, que es de 1800 Hz.

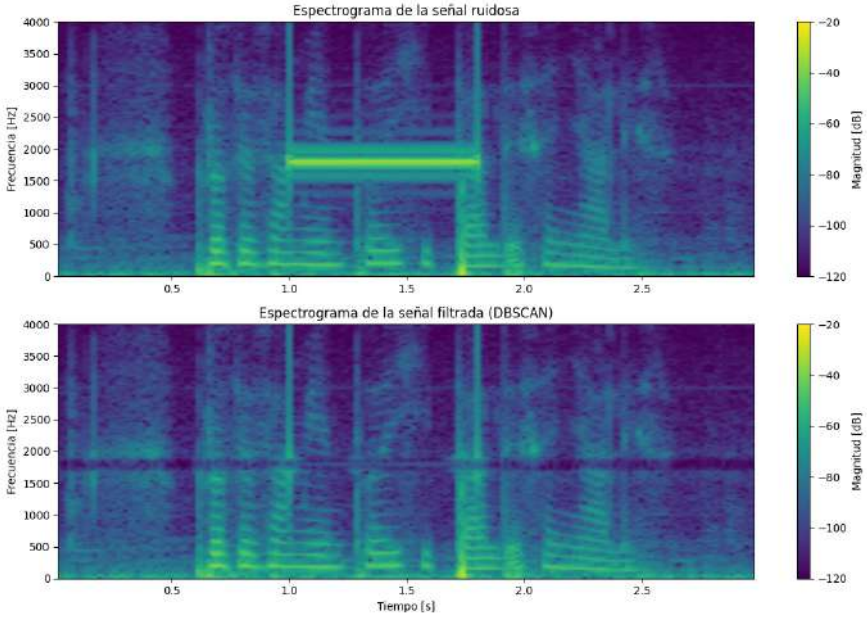


Figura 63. Espectrograma del Ejemplo 19: algoritmo DBSCAN para el filtrado de ruido tipo tono en señal de voz.

Observamos que el tono ubicado en 1800 Hz fue significativamente atenuado. La energía residual del tono se redujo hasta 1.17×10^{-6}, mientras que la energía conservada en la banda útil de la voz fue de 1.26×10^{-2}. Aunque la atenuación no alcanza los niveles obtenidos por los filtros Butterworth y Blackman, el tono residual deja de ser perceptualmente molesto y se conserva una mayor cantidad de energía útil de la señal de voz.

3.3 Análisis comparativo de las soluciones obtenidas con PDS clásico y con IA

A lo largo de los Capítulos 2 y 3 hemos explorado dos enfoques muy diferentes para la atenuación de un ruido tonal presente en una señal de voz.

En el primer caso, correspondiente al Procesamiento Digital de Señales (PDS) clásico, utilizamos dos filtros FIR (promedio y Blackman) y dos filtros IIR (Butterworth y Leaky), todos ellos aplicados directamente sobre la señal de voz en el dominio del tiempo.

En el segundo caso, utilizamos técnicas de aprendizaje no supervisado para identificar automáticamente la ubicación espectral del tono interferente. Posteriormente, construimos una máscara de atenuación equivalente a un filtro *notch* (rechaza-banda altamente selectivo), aplicada directamente sobre el espectrograma de la señal. Finalmente, reconstruimos la señal filtrada en el dominio del tiempo mediante la transformada inversa correspondiente.

Es importante destacar una diferencia fundamental entre ambos enfoques. En el caso de los filtros pasa-bajos utilizados en el Capítulo 2, no era necesario conocer con exactitud la frecuencia del tono interferente. Bastaba con seleccionar una frecuencia de corte adecuada (1200 Hz en nuestro caso) para garantizar que el tono quedara fuera de la banda de paso. Por esta razón, se habrían obtenido resultados similares si el tono hubiese estado ubicado en 1500 Hz, 1600 Hz, 2000 Hz o en cualquier frecuencia superior.

Por el contrario, en el enfoque basado en IA y enmascaramiento espectral sí es necesario conocer con precisión la ubicación del tono, ya que la máscara debe aplicarse específicamente sobre la región espectral asociada a la interferencia. La diferencia es que, en este caso, dejamos que sea la IA quien realice dicha identificación de manera automática.

La Figura 64 resume las principales diferencias entre ambos enfoques.

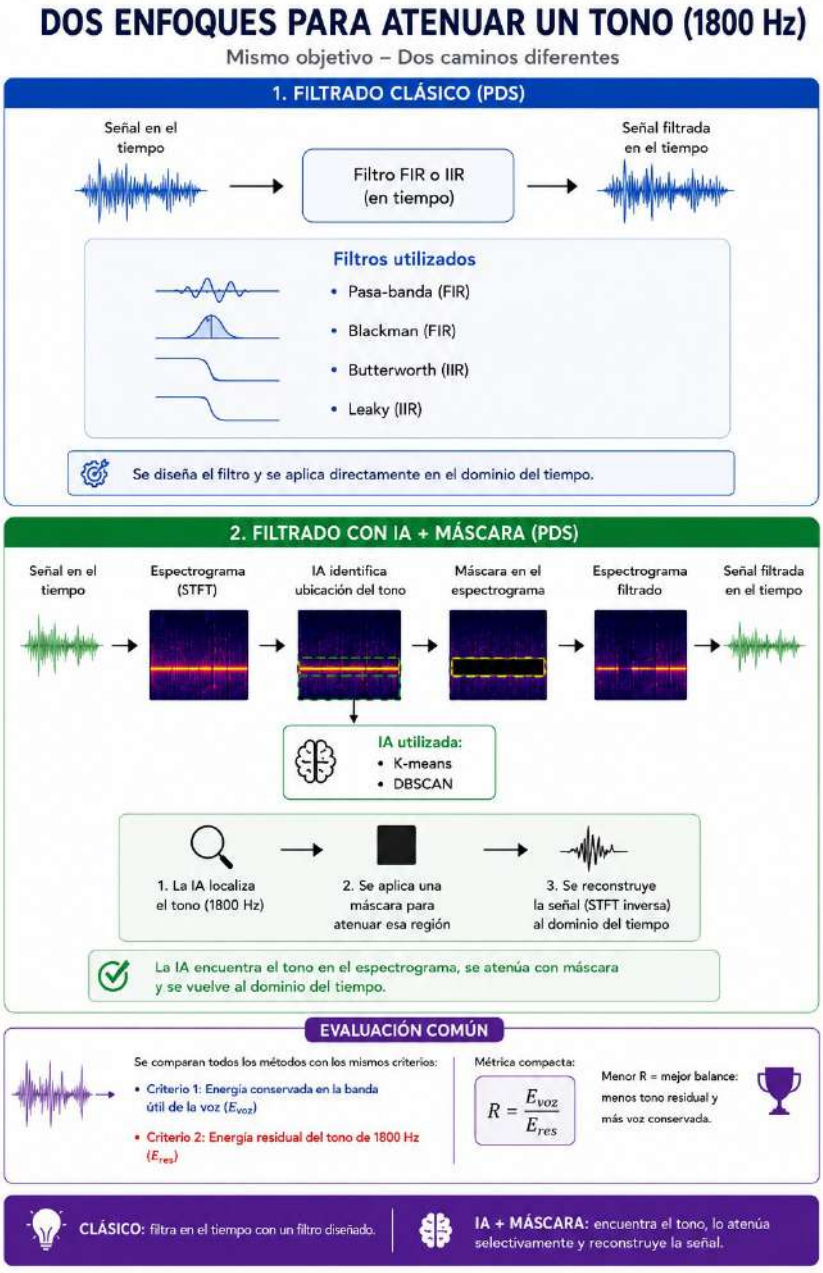


Figura 64. Comparación entre el filtrado de ruido tonal mediante filtros digitales clásicos y el enfoque basado en IA para localización de la interferencia y filtrado mediante máscara espectral.

Ejemplo 20. Comparación cuantitativa de filtrado de voz con ruido tonal utilizando filtros FIR, IIR y máscara con localización de tono con IA

Para realizar una comparación objetiva entre las seis soluciones encontradas, partiremos inicialmente con los dos criterios de evaluación presentados en el Capítulo 2. De esta forma, evaluaremos conjuntamente las cuatro soluciones basadas en PDS clásico y las dos soluciones basadas en aprendizaje no supervisado.

Una vez más, resulta importante recordar que los filtros clásicos operan directamente sobre la señal de voz en el dominio del tiempo, mientras que los métodos basados en IA operan sobre una representación tiempo-frecuencia de la señal (espectrograma), utilizando posteriormente una máscara de atenuación antes de reconstruir la señal filtrada.

La Tabla 1 presenta el consolidado de las seis soluciones evaluadas:

Tabla 1. Comparación de señales filtrada en términos de energía residual del tono y energía conservada de la voz.

Método	Energía residual del tono	Energía conservada de voz
Promedio	5.432817e-05	7.453849e-03
Blackman	**6.927825e-13**	8.685923e-03
Butterworth	**6.287068e-13**	8.682669e-03
Leaky	1.518678e-09	7.247906e-04
K-means	2.574687e-02	8.688575e-03
DBSCAN	1.165503e-06	**1.258411e-02**

Al analizar la energía residual del tono, observamos que los mejores resultados corresponden a los filtros **Butterworth** y **Blackman**, con valores del orden de 10^{-13}. Esto indica que ambos filtros lograron eliminar prácticamente la totalidad de la energía asociada al tono interferente de 1800 Hz. El filtro

Leaky también presentó una atenuación significativa, mientras que el filtro de **promedio** obtuvo un desempeño intermedio. Por el contrario, **K-means** fue el método menos efectivo para reducir la energía del tono, mientras que **DBSCAN** logró una reducción considerable, aunque sin alcanzar el nivel de los mejores filtros clásicos.

Si ahora analizamos la energía conservada en la banda útil de la voz, encontramos un resultado diferente. El método que preservó una mayor cantidad de energía útil fue **DBSCAN**, seguido por **K-means**, **Blackman** y **Butterworth**, cuyos resultados fueron muy similares entre sí. El filtro de **promedio** presentó una pérdida moderada de energía útil, mientras que el filtro **Leaky** fue el que más degradó la señal de voz.

Estos resultados muestran que no existe un ganador absoluto cuando se evalúan los dos criterios de forma independiente. Algunos métodos son más efectivos eliminando el tono, mientras que otros preservan mejor la energía de la señal de voz. Por esta razón, resulta conveniente definir una métrica que permita integrar ambos criterios en un único valor de desempeño.

Por lo que, la comparación final la realizaremos con la siguiente métrica:

Relación entre energía residual del tono y energía conservada de la señal de voz (R)

Vamos a aplicar la siguiente ecuación:

$$R = \frac{E_{tono}}{E_{voz}}$$

donde:

- E_{tono} corresponde a la energía residual del tono.
- E_{voz} corresponde a la energía conservada en la banda útil de la voz.

Un valor pequeño de R indica simultáneamente una baja energía residual del tono y una alta conservación de la señal útil, representando un mejor compromiso entre ambos objetivos.

Al aplicar R, obtenemos:

- **Butterworth** → 7.24×10^{-11}
- **Blackman** → 7.98×10^{-11}
- **Leaky** → 2.10×10^{-6}
- **DBSCAN** → 9.26×10^{-5}
- **Promedio** → 7.29×10^{-3}
- **K-means** → 2.962.96

Podríamos decir que, para las condiciones de nuestro experimento, el ganador fue el filtro **Butterworth**, seguido muy de cerca por el filtro **Blackman** (el primero de tipo IIR y el segundo de tipo FIR). Sin embargo, esto no significa que estos dos filtros sean "absolutamente mejores" que los demás para eliminar ruido tonal en señales de voz. Lo único que podemos afirmar es que, con los parámetros de diseño utilizados en este caso de estudio, fueron las soluciones que obtuvieron los mejores resultados.

Me explico. Los resultados obtenidos con **DBSCAN** podrían mejorar si se modifican parámetros como el nivel de atenuación o el ancho de banda de la máscara aplicada sobre el espectrograma. De forma similar, si hubiésemos seleccionado otros valores de diseño para los filtros **Butterworth** o **Blackman** (por ejemplo, órdenes menores), es muy probable que la diferencia respecto a sus competidores hubiese sido menor.

Por esta razón, es importante comprender que el desempeño de un filtro no depende únicamente de su tipo (FIR, IIR o basado en IA), sino también de los parámetros utilizados durante su diseño e implementación. En consecuencia, la selección adecuada de dichos parámetros puede tener un impacto tan importante sobre la calidad de la señal filtrada como la propia elección del método de filtrado.

3.4 ¿Qué aprendimos en esta primera sección?

A lo largo de esta primera sección recorrimos un camino que va mucho más allá del estudio de algunos algoritmos de procesamiento digital de señales. Comenzamos comprendiendo los fundamentos del procesamiento digital

de señales y el funcionamiento de filtros digitales clásicos. Posteriormente, aplicamos estos conceptos para resolver un problema real de eliminación de ruido en señales de voz. Finalmente, abordamos exactamente el mismo problema desde una perspectiva diferente, incorporando técnicas de IA para localizar automáticamente la interferencia antes de aplicar el procesamiento digital tradicional.

Uno de los principales aprendizajes de esta sección es que **los métodos clásicos y la Inteligencia Artificial no son enfoques opuestos, sino herramientas complementarias**. Los algoritmos clásicos permiten diseñar soluciones basadas en el conocimiento del problema y del comportamiento de la señal, mientras que la IA aprende patrones directamente a partir de los datos cuando resulta difícil definir reglas explícitas.

En este caso de estudio, la IA no reemplazó al procesamiento digital de señales. Por el contrario, permitió localizar automáticamente la región afectada por el ruido, mientras que los filtros digitales continuaron desempeñando el papel fundamental de eliminar dicha interferencia. Este tipo de integración representa una tendencia cada vez más frecuente en la ingeniería moderna: combinar modelos basados en conocimiento con modelos basados en datos para construir soluciones más robustas y eficientes.

Así que, permíteme resumir lo más importante de esta sección:

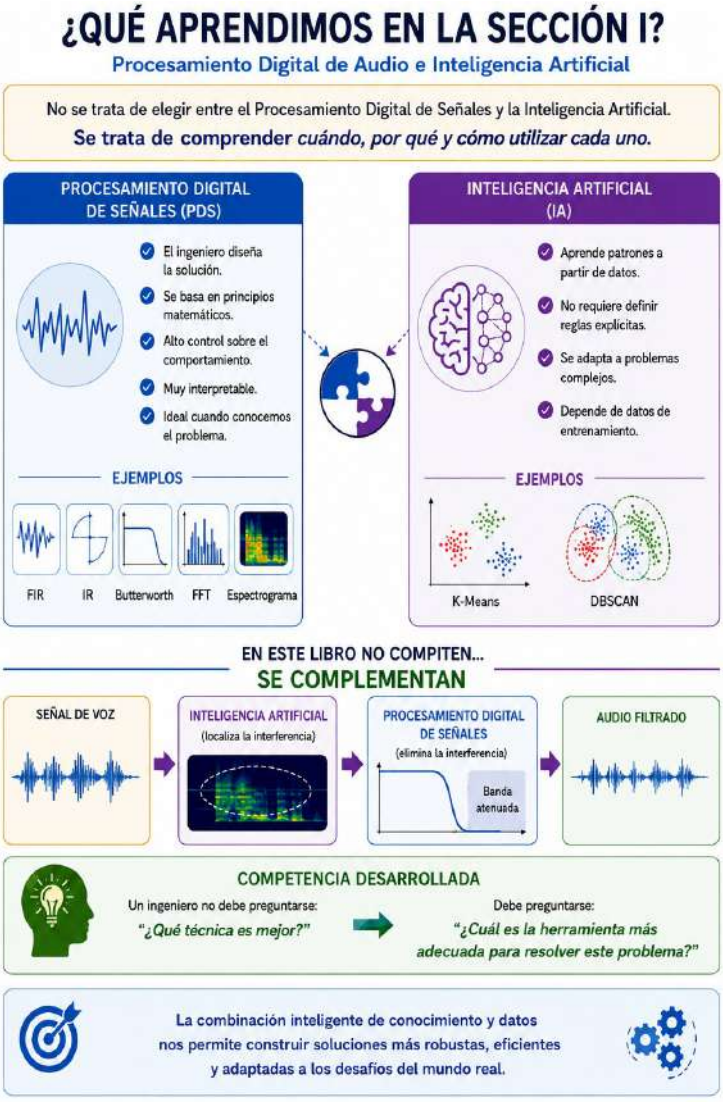


Figura 65. Síntesis de los principales aprendizajes de la Sección I.

Si esta primera sección te deja una idea, espero que sea la siguiente:

La ingeniería no consiste en elegir entre los métodos clásicos o la Inteligencia Artificial, sino en comprender cuándo, por qué y cómo utilizar cada uno de ellos.

Los algoritmos evolucionan con el tiempo, pero la capacidad para analizar un problema y seleccionar la herramienta más adecuada seguirá siendo una de las competencias más valiosas para cualquier ingeniero.

SECCIÓN II

PROCESAMIENTO DIGITAL DE IMÁGENES E INTELIGENCIA ARTIFICIAL

Nueva sección, nuevo tipo de señal: en este caso, **señales bidimensionales**, específicamente imágenes. Y te preguntarás:

¿Por qué es necesaria una sección aparte para este tipo de señales?

La respuesta es sencilla. Aunque muchas de las ideas fundamentales del procesamiento digital siguen siendo las mismas, las técnicas utilizadas cambian considerablemente. El tipo de ruido ya no es el mismo que encontramos en las señales de audio, y también cambian los filtros y herramientas que nos permitirán mejorar la calidad de la imagen. Además, aparecen conceptos que no desempeñaban un papel relevante en el procesamiento de audio, pero que resultan fundamentales en el procesamiento digital de imágenes, como los histogramas, la ecualización, la detección de bordes, las operaciones morfológicas y la segmentación.

En algunos programas de ingeniería, los cursos de procesamiento de señales unidimensionales (1D) y bidimensionales (2D) se estudian por separado. Sin embargo, hacerlo de esta manera puede hacer que se pierda una visión más amplia del procesamiento digital de señales. Aunque las técnicas específicas sean diferentes, muchos de los conceptos estudiados en la sección anterior continúan presentes: representación digital, eliminación de ruido, filtrado,

evaluación de resultados e incluso la Inteligencia Artificial como una herramienta para resolver problemas reales.

En esta sección recorreremos tres etapas complementarias. En el **Capítulo 4** estudiaremos los fundamentos del procesamiento digital de imágenes, comprendiendo cómo se representan, mejoran y analizan las imágenes digitales.

En el **Capítulo 5** aplicaremos estos conceptos para resolver un problema real de segmentación mediante técnicas clásicas de procesamiento digital de imágenes.

Finalmente, en el **Capítulo 6** abordaremos el mismo problema utilizando una arquitectura **U-Net**, mostrando cómo la Inteligencia Artificial es capaz de aprender automáticamente la tarea de segmentación a partir de ejemplos previamente etiquetados.

Más que presentar dos soluciones diferentes, el objetivo será comprender dos formas de abordar un mismo problema. La primera, basada en reglas diseñadas explícitamente por el ingeniero; la segunda, basada en modelos de aprendizaje profundo capaces de aprender dichas reglas directamente a partir de los datos. Al finalizar esta sección no solo conocerás ambas estrategias, sino que también estarás en capacidad de comprender sus ventajas, limitaciones y los escenarios en los que resulta más conveniente utilizar cada una de ellas.

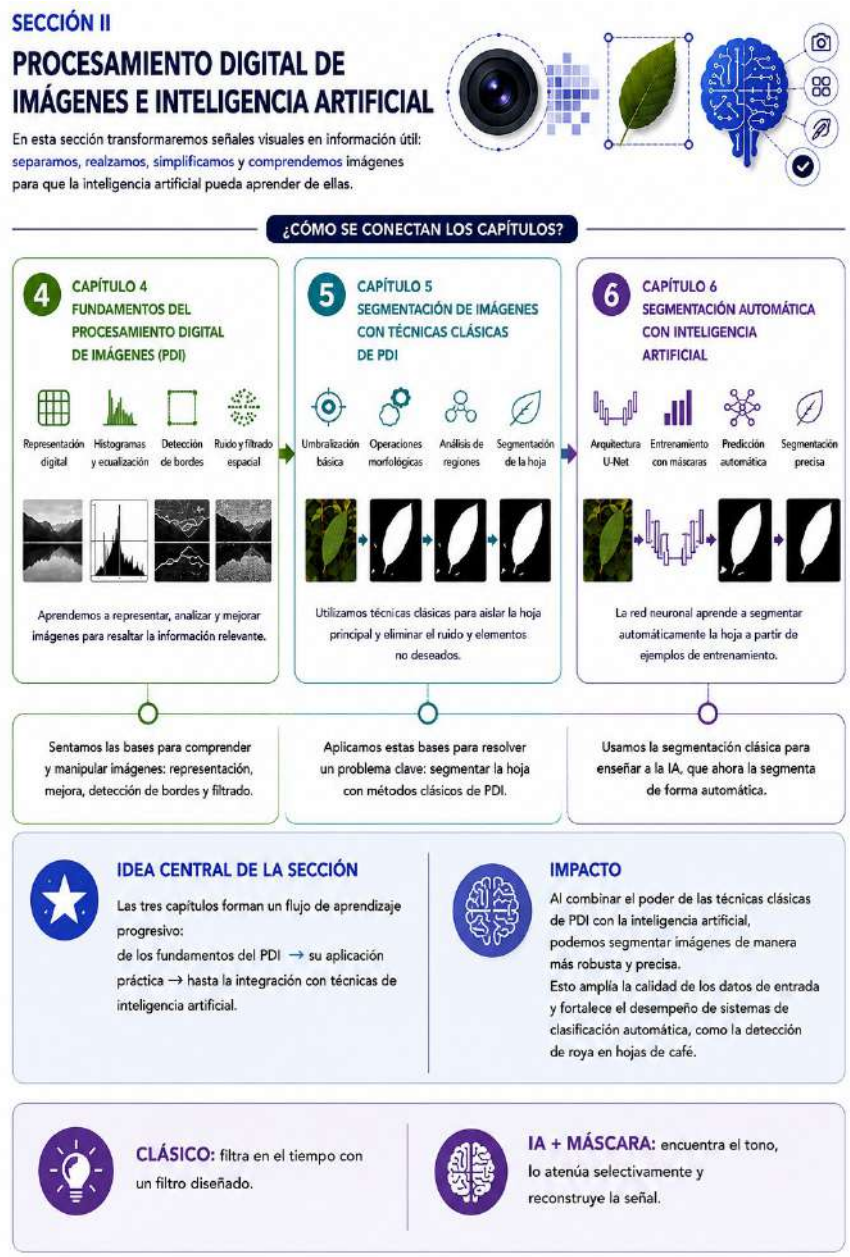


Figura 66. Sección II: desde los fundamentos del procesamiento digital de imágenes (PDI), pasando por la segmentación mediante técnicas clásicas, hasta la segmentación automática con Inteligencia Artificial mediante una arquitectura U-Net.

La **Figura 66** resume el recorrido que seguiremos durante esta sección. Comenzaremos comprendiendo los fundamentos del procesamiento digital de imágenes y las principales herramientas que nos permitirán mejorar, analizar y extraer información de una imagen. Posteriormente, aplicaremos estos conceptos para resolver un problema real de segmentación utilizando técnicas clásicas de procesamiento digital de imágenes. Finalmente, abordaremos exactamente el mismo problema desde una perspectiva completamente diferente, utilizando Inteligencia Artificial y una arquitectura U-Net capaz de aprender automáticamente a segmentar una imagen.

Espero que esta sección no solo te permita aprender nuevas técnicas. También deseo que, si ya estudiaste procesamiento digital de imágenes hace algunos años, tengas la oportunidad de **redescubrir** muchos de esos conceptos desde una perspectiva diferente, integrándolos con herramientas modernas como Python y la Inteligencia Artificial. A veces, volver sobre un tema no significa repetir lo aprendido, sino comprenderlo con una mayor profundidad.

Si al terminar estas páginas logras comprender que un mismo problema puede resolverse tanto mediante reglas diseñadas por un ingeniero como mediante modelos capaces de aprender directamente de los datos, y además disfrutas ese proceso de aprendizaje, entonces el objetivo de esta sección se habrá cumplido.

CAPÍTULO IV

FUNDAMENTOS DEL PROCESAMIENTO DIGITAL DE IMÁGENES

Los tres primeros capítulos del libro estuvieron enfocados en señales unidimensionales (1D), y específicamente los Capítulos 2 y 3 en señales de voz. Este tipo de señales dependen típicamente del tiempo y pueden representarse como una secuencia ordenada de muestras.

Ahora, en la Sección II trabajaremos con imágenes, las cuales corresponden a señales bidimensionales (2D). A diferencia de las señales de audio, las imágenes describen información espacial y, en la mayoría de las aplicaciones, no dependen explícitamente del tiempo.

Es decir, mientras una señal 1D puede interpretarse como una secuencia de muestras, una imagen puede representarse como una matriz cuyos elementos contienen información de intensidad o color.

Dependiendo del tipo de imagen, los valores almacenados en cada elemento de la matriz pueden tener diferentes formatos. En las imágenes más comunes, cada píxel se almacena utilizando 8 bits sin signo (*uint8*), permitiendo representar valores enteros en el rango de 0 a 255. Sin embargo, también existen imágenes binarias, imágenes de 16 bits e incluso imágenes en punto flotante utilizadas en aplicaciones científicas y médicas.

La Figura 67 presenta una comparación entre las principales características de las señales unidimensionales y las señales bidimensionales.

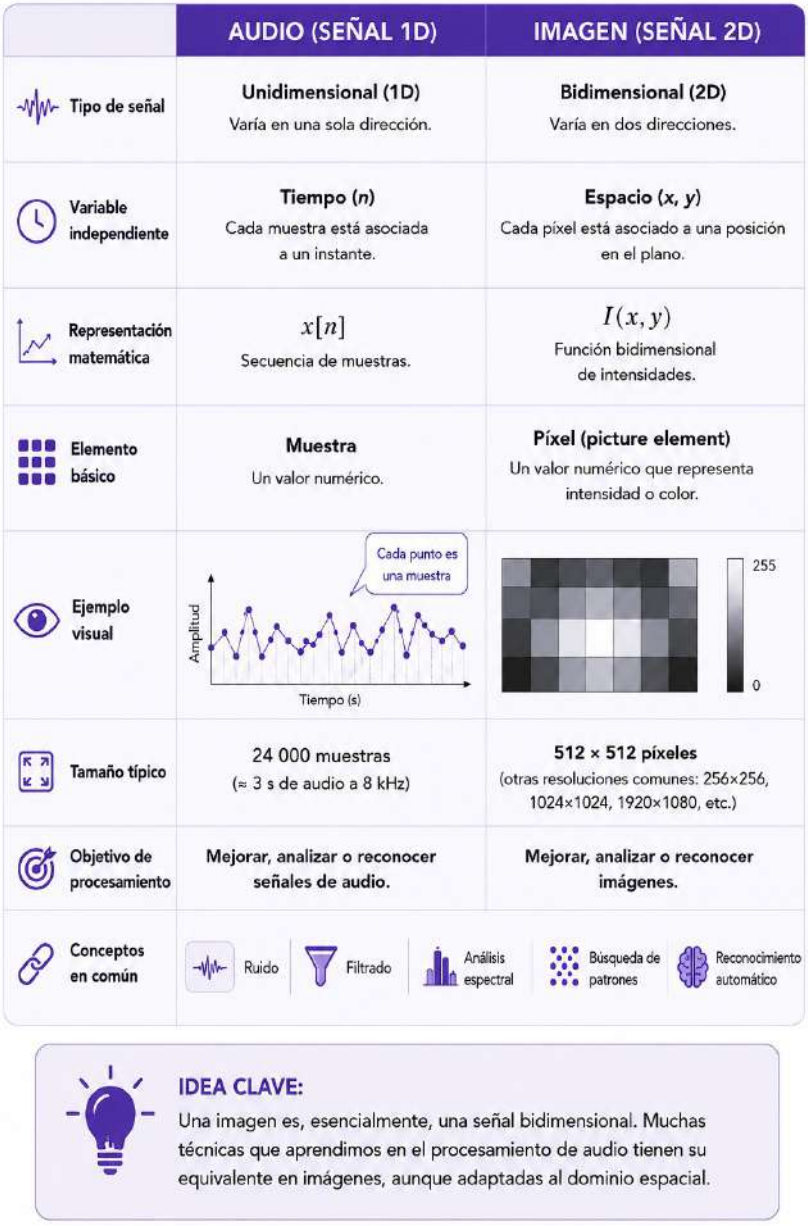


Figura 67. Comparación señal de audio vs. imagen: tipo de señal, variable independiente, representación matemática, elemento básico, ejemplo visual, tamaño típico, objetivo de procesamiento.

En el caso de las imágenes, la representación matemática más común es:

$$I(x,y)$$

donde:

- x corresponde a una coordenada espacial horizontal.
- y corresponde a una coordenada espacial vertical.

Cada par (x,y) identifica un píxel dentro de la imagen y el valor asociado representa su intensidad o color.

4.1 Resolución espacial y profundidad de bits

Existen dos conceptos fundamentales que permiten caracterizar una imagen digital: la **resolución espacial** y la **profundidad de bits**.

Comencemos con la **profundidad de bits**. Una imagen puede almacenar diferente cantidad de información en cada píxel dependiendo del número de bits utilizados para representarlo. Por ejemplo, podemos encontrar imágenes de 1 bit por píxel, 8 bits por píxel o 24 bits por píxel.

En el primer caso, tenemos imágenes binarias (*Black and White*, B/W), donde cada píxel puede tomar únicamente uno de dos valores posibles: 0 (negro) o 1 (blanco).

También, encontramos las imágenes en escala de grises, que habitualmente utilizan 8 bits por píxel. Esto significa que cada píxel puede tomar valores entre:

$$00000000_2 = 0$$

y

$$11111111_2 = 255$$

De esta forma, además del negro y el blanco, disponemos de 254 tonalidades intermedias de gris que permiten representar transiciones mucho más suaves en la imagen.

Finalmente, tenemos las imágenes a color en modelo RGB (*Red, Green, Blue*). En este caso, la imagen está compuesta por tres canales independientes:

- Canal rojo (*Red*).
- Canal verde (*Green*).
- Canal azul (*Blue*).

Cada canal utiliza típicamente 8 bits, para un total de:

$$8 + 8 + 8 = 24 \text{ bits por píxel}$$

La combinación de las intensidades de estos tres canales permite representar millones de colores diferentes (exactamente 2^{24} colores diferentes).

La Figura 68 presenta una comparación de los tres tipos de imágenes que abordaremos en esta sección. Cabe aclarar que existen otros tipos de imágenes, como las imágenes multiespectrales e hiperespectrales, que quedan por fuera del alcance de este libro.

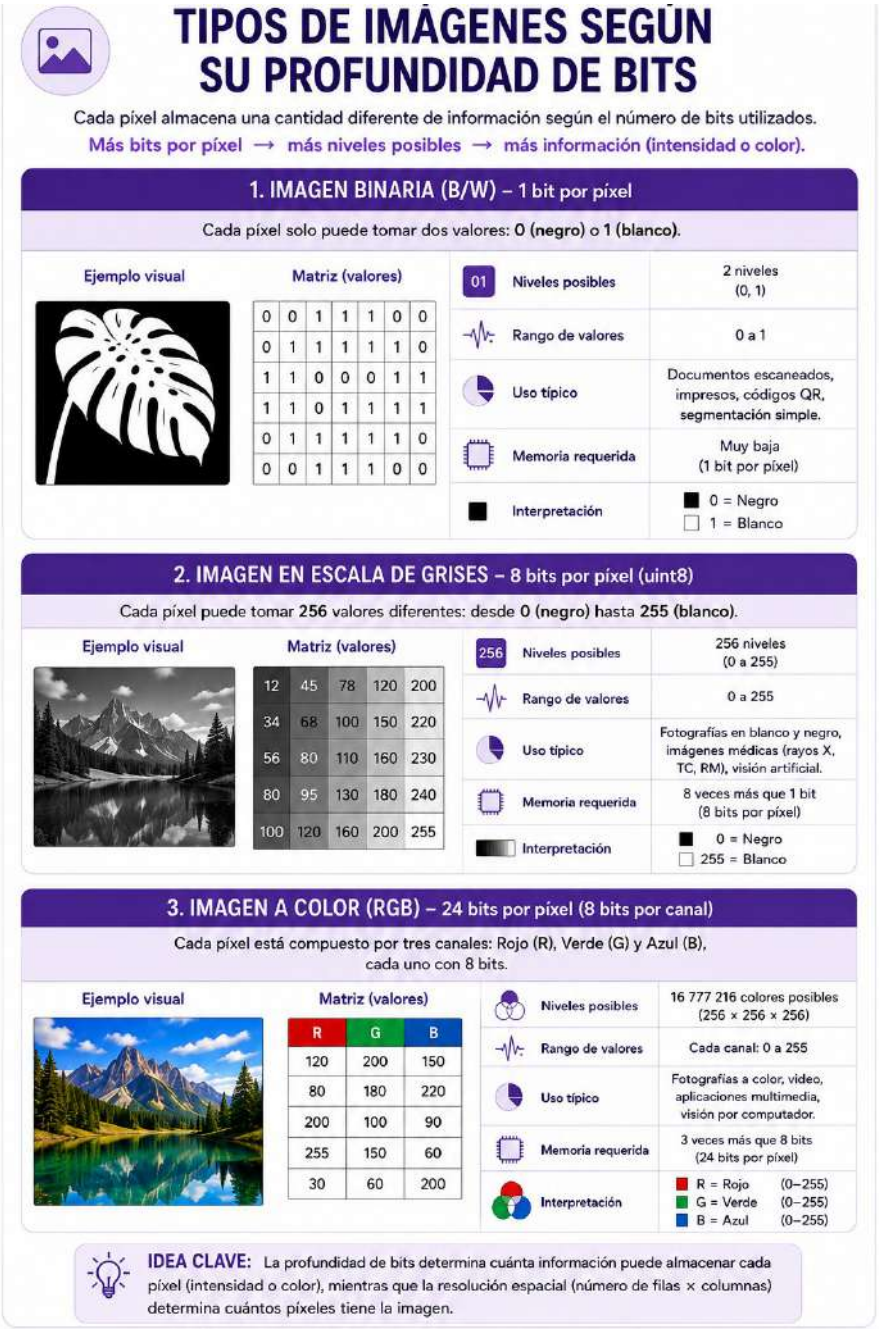


Figura 68. Tipos de imágenes según su profundidad de bits: binaria (B/W), en escala de grises y a color (RGB).

Un aspecto importante que quiero resaltar corresponde al aumento en los requerimientos de almacenamiento a medida que se incrementa la profundidad de bits. Por ejemplo, si una imagen RGB posee las mismas dimensiones que una imagen en escala de grises, su “peso” será aproximadamente tres veces mayor, dado que contiene tres canales (rojo, verde y azul) en lugar de uno solo. De forma similar, una imagen RGB requerirá aproximadamente veinticuatro veces más memoria que una imagen binaria, ya que cada píxel se representa mediante 24 bits en lugar de 1 bit (y digo *aproximadamente* porque, además de los píxeles, también se almacena información adicional como encabezados y metadatos).

En consecuencia, una mejor representación visual y una mayor cantidad de información por píxel traen consigo un costo adicional en términos de almacenamiento, transmisión y procesamiento. Como veremos más adelante, este compromiso entre calidad visual y recursos computacionales es una consideración frecuente en aplicaciones de procesamiento digital de imágenes. En ingeniería, como ocurre en muchos otros problemas, obtener más información suele implicar consumir más recursos.

Y ahora, pasemos al concepto de **resolución espacial**, el cual nos indica cuántos píxeles componen una imagen.

La resolución espacial suele expresarse mediante el número de filas y columnas de la matriz que representa la imagen. Por ejemplo, una imagen de:

512×512

posee 512 filas y 512 columnas, para un total de:

$512 \times 512 = 262{,}144$ píxeles.

De manera similar, una imagen Full HD tiene una resolución de:

1920×1080 lo que equivale a más de dos millones de píxeles.

Cuantos más píxeles tengamos disponibles, más pequeños serán los detalles que podremos observar. Sin embargo, como ya vimos al hablar de la profundidad de bits, este aumento en la calidad también tiene un costo asociado: se requiere más memoria para almacenar la imagen y más recursos computacionales para procesarla.

La Figura 69 presenta una comparación de una misma imagen utilizando diferentes resoluciones espaciales. Observa cómo, a medida que disminuye el número de píxeles, la imagen comienza a perder detalle y las transiciones se vuelven más abruptas.



Figura 69. Efecto de la resolución espacial sobre la calidad visual de la imagen.

Existe una analogía interesante con las señales de audio estudiadas en capítulos anteriores. En una señal de voz utilizábamos la frecuencia de muestreo (f_s) para determinar cuántas muestras se almacenaban por segundo (a mayor f_s, el audio "sonaba mejor"). En una imagen ocurre algo similar: la resolución espacial determina en cuántas filas y columnas se divide la imagen.

En ambos casos, aumentar el número de muestras o píxeles permite representar mejor la información original. Sin embargo, esta mejora tiene un costo asociado: se requiere más memoria para almacenar los datos y más recursos computacionales para procesarlos.

Ejemplo 21. Efecto de la resolución espacial y la profundidad de bits de la imagen

Supongamos que tenemos una imagen de:

512×512

Esto significa que la imagen contiene:

$512 \times 512 = 262,144 512 \times 512 = 262,144$ píxeles

Si la imagen es binaria, cada píxel requiere 1 bit:

$262,144 \times 1 = 262,144$ bits lo que equivale a:

$32,768$ bytes ≈ 32 KB

Nota: *En este libro utilizaremos las equivalencias tradicionales empleadas en informática, donde 1 KB = 1024 bytes y 1 MB = 1024 KB.*

Si la imagen está en escala de grises, cada píxel requiere 8 bits:

$262,144 \times 8 = 2'097,152$ bits es decir:

$262,144$ bytes ≈ 256 KB

Finalmente, si la imagen es RGB, cada píxel requiere 24 bits:

$262,144 \times 24 = 6'291,456$ bits

lo que equivale a:

$786,432$ bytes ≈ 768 KB

Podemos observar que la resolución espacial permanece igual en los tres casos. Lo que cambia es la profundidad de bits. Por esta razón, una imagen RGB de 512×512 píxeles requiere aproximadamente tres veces más memoria que una imagen en escala de grises y veinticuatro veces más que una imagen binaria.

Este ejemplo nos permite ver que el tamaño de una imagen depende de dos factores: cuántos píxeles tiene y cuántos bits se usan para representar cada píxel.

$$\text{Tamaño en bits} = \text{filas} \times \text{columnas} \times \text{bits por píxel}$$

Aunque se trata de un ejemplo simplificado que no considera encabezados ni mecanismos de compresión, permite comprender cómo la resolución espacial y la profundidad de bits impactan directamente el tamaño de almacenamiento requerido.

4.2 Formatos de imagen y compresión

Hasta este punto hemos asumido que las imágenes se almacenan exactamente como las hemos descrito: píxel por píxel y utilizando todos los bits necesarios para representar la información. De hecho, los tamaños calculados en el Ejemplo 21 corresponden a esta situación ideal, donde no existe ningún tipo de compresión.

Sin embargo, hagamos una pausa y pensemos en algo cotidiano. ¿Cuántas fotografías tienes almacenadas actualmente en tu celular? ¿Cien? ¿Mil? ¿Tal vez varias decenas de miles? Ahora imagina que cada una de ellas ocupara varios megabytes. Muy rápidamente nos quedaríamos sin espacio de almacenamiento.

Y el problema no termina ahí. Las imágenes no solamente se almacenan; también se envían por WhatsApp, se publican en redes sociales, se cargan en páginas web y se transmiten continuamente a través de Internet. Si cada fotografía mantuviera siempre su tamaño original, el espacio de almacenamiento se llenaría rápidamente (más).

Por esta razón, se desarrollaron diferentes métodos de compresión de imágenes cuyo objetivo es reducir el tamaño de los archivos sin afectar excesivamente su apariencia visual. Sin embargo, no todos producen el mismo efecto sobre la imagen. En términos generales, podemos clasificarlos en dos grandes categorías: compresión *lossless* y compresión *lossy*.

- **Compresión sin pérdida de información (*lossless*):** permite recuperar exactamente la imagen original después de descomprimirla.
- **Compresión con pérdida de información (*lossy*):** reduce aún más el tamaño del archivo eliminando parte de la información visual que se considera poco relevante para la percepción humana.

Entre los formatos más conocidos de compresión *lossless* encontramos PNG, GIF, TIFF y BMP. Por su parte, entre los formatos lossy destacan JPEG/JPG, WebP y HEIF/HEIC.

La Figura 70 presenta una comparación general de ambos enfoques.

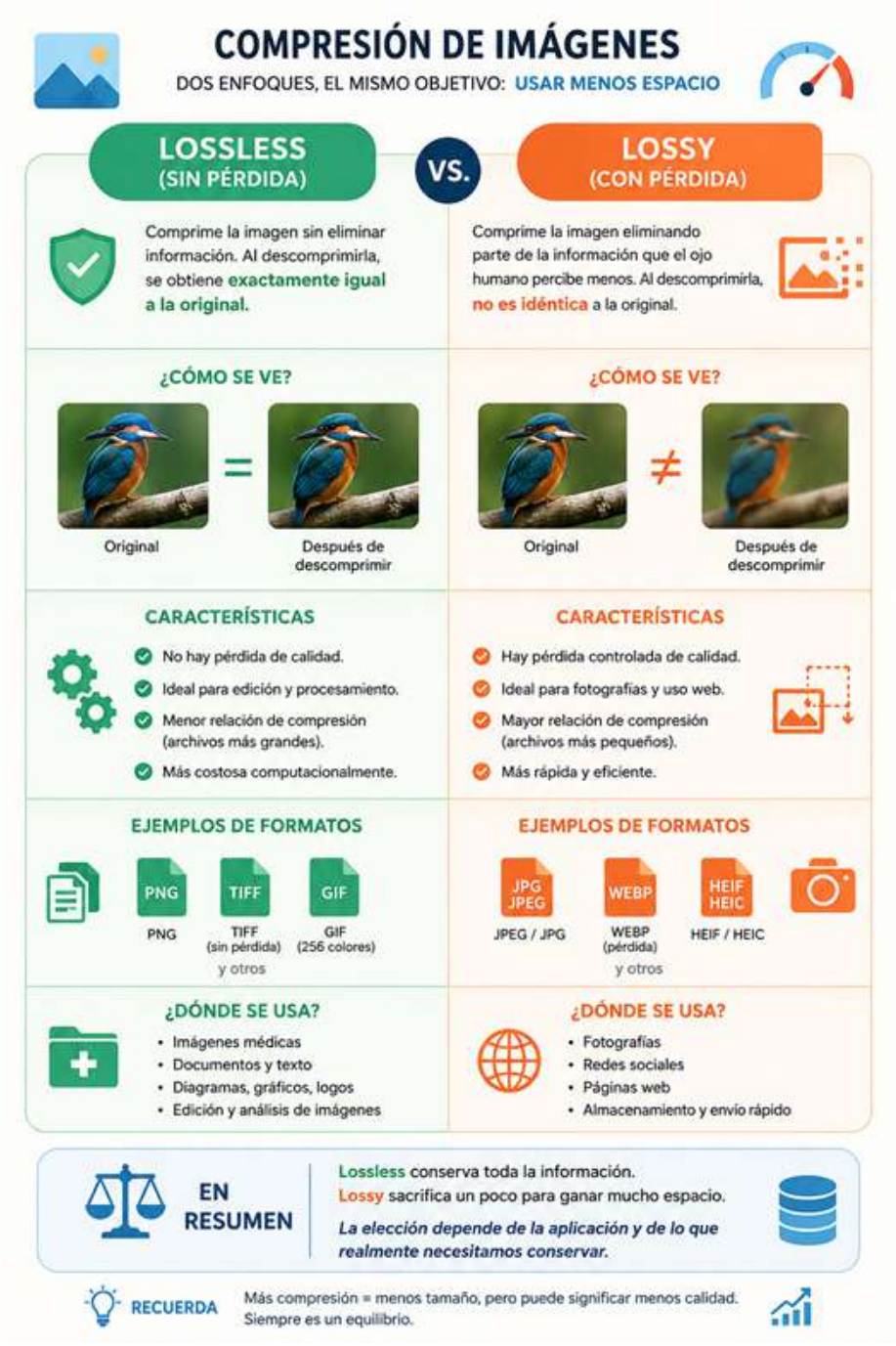


Figura 70. Compresión de imágenes: formatos *lossless* y *lossy*.

No podemos afirmar que una alternativa sea siempre mejor que la otra. La elección depende en gran medida de la aplicación. Por ejemplo, en imágenes médicas suele ser indispensable utilizar formatos *lossless*, ya que preservan todos los detalles de la imagen, algunos de los cuales pueden ser fundamentales para identificar una patología o realizar un diagnóstico adecuado.

Por el contrario, cuando se requiere almacenar o transmitir grandes cantidades de imágenes, suelen preferirse formatos *lossy*, ya que permiten reducir significativamente el tamaño de los archivos. En muchos casos, la pérdida de información es tan pequeña que resulta prácticamente imperceptible para el ojo humano.

En consecuencia, la selección del formato de imagen representa un compromiso entre calidad visual, capacidad de almacenamiento y velocidad de transmisión, una situación que encontraremos con frecuencia a lo largo de este libro.

4.3 Histograma y ecualización

Hasta este punto hemos estudiado cómo se almacena una imagen, cuántos píxeles la componen, cuántos bits utiliza cada uno de ellos y qué formatos existen para almacenarla. Sin embargo, aún no hemos analizado una pregunta fundamental: ¿cómo se distribuyen los valores de intensidad dentro de una imagen?

La respuesta la encontramos en el histograma. De manera intuitiva, es una representación gráfica de la distribución de los píxeles de una imagen. En otras palabras, indica cuántos píxeles existen para cada uno de los niveles de intensidad o color presentes en la imagen. Por ejemplo, en una imagen en escala de grises de 8 bits, los píxeles pueden tomar valores entre 0 y 255. El histograma contará cuántos píxeles poseen intensidad 0, cuántos poseen intensidad 1, cuántos poseen intensidad 2 y así sucesivamente hasta llegar a 255.

Matemáticamente, el histograma puede representarse como:

$h(i)$ donde:

i corresponde al nivel de intensidad.

$h(i)$ representa la cantidad de píxeles que poseen dicha intensidad.

La Figura 71 ilustra la relación entre una imagen en escala de grises y su correspondiente histograma.

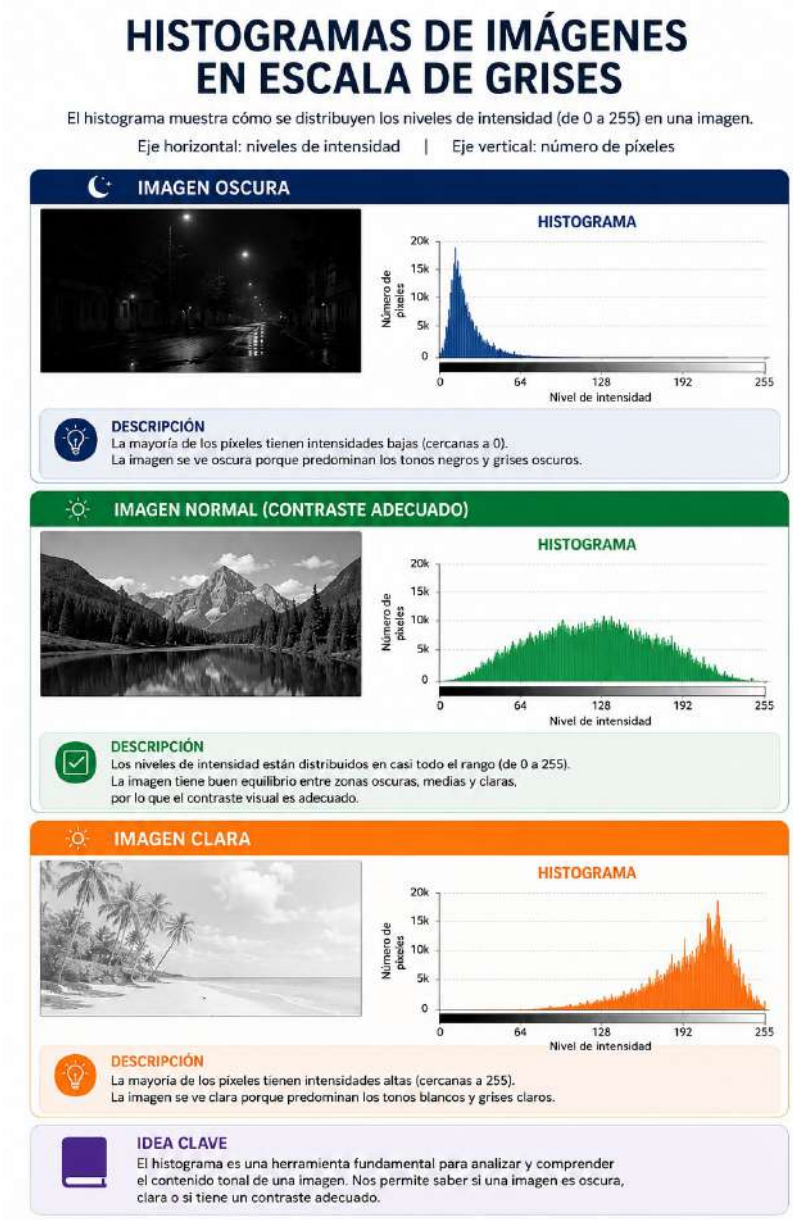


Figura 71. Ejemplos de histogramas de imágenes con diferentes características: oscura (izquierda), contraste adecuado (central) y clara (derecha).

Es importante aclarar que dos imágenes completamente diferentes pueden tener histogramas similares. Dado que, el histograma no describe la posición espacial de los objetos, sino únicamente la distribución global de intensidades presentes en la imagen.

Pero ¿cómo podemos interpretar un histograma?

Vamos a realizar este análisis con los tres ejemplos de la Figura 71. En la imagen de la izquierda, la distribución se concentra principalmente en niveles bajos de intensidad, dando lugar a una imagen oscura. En la imagen central, los niveles de intensidad ocupan una región más amplia del rango disponible, produciendo un contraste adecuado. Finalmente, en la imagen de la derecha, la distribución se desplaza hacia niveles altos de intensidad, generando una imagen clara.

Típicamente, una imagen oscura la obtenemos cuando tomamos una fotografía de noche sin utilizar flash. Por el contrario, una imagen clara puede aparecer cuando tomamos una fotografía en un día muy soleado. En ambos casos, el histograma nos permite identificar rápidamente estas características incluso antes de observar la imagen en detalle.

Ahora, ¿qué podemos hacer para mejorar la distribución de pixeles de la imagen?

La respuesta la encontramos en la **ecualización de histograma**, una de las técnicas más utilizadas para mejorar el contraste de una imagen.

La idea detrás de esta técnica es bastante sencilla. La ecualización redistribuye los niveles de intensidad de la imagen para aprovechar mejor el rango disponible. Como resultado, se mejora el contraste y se facilita la visualización de los detalles presentes en la imagen.

La Figura 72 ilustra de forma conceptual el efecto de la ecualización sobre una imagen y su correspondiente histograma.

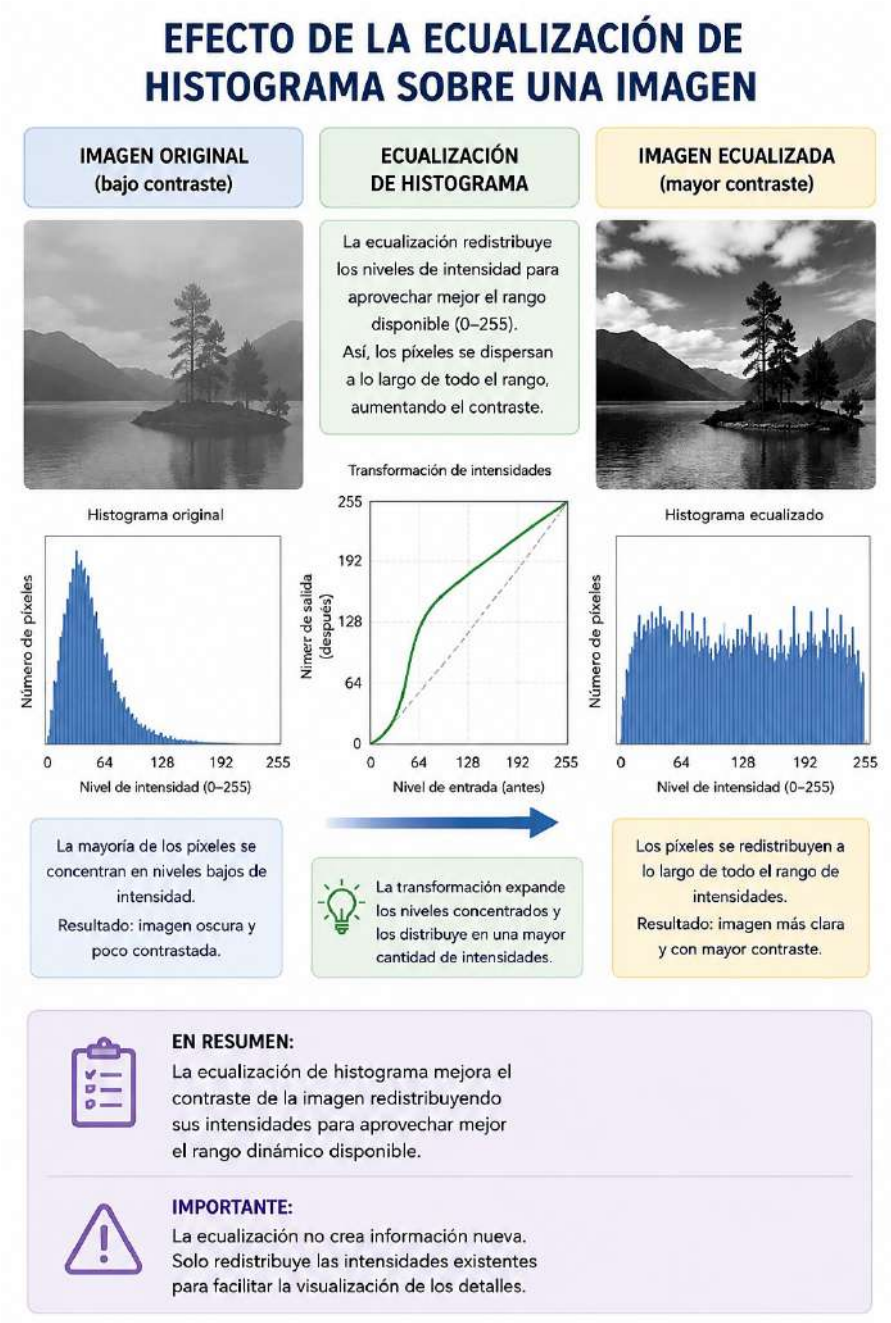


Figura 72. Efecto de la ecualización de histograma sobre una imagen.

Después de aplicar la ecualización, los detalles que antes resultaban difíciles de distinguir pueden hacerse más visibles, especialmente en imágenes con bajo contraste.

Esta técnica es ampliamente utilizada en aplicaciones como: procesamiento de imágenes médicas, análisis satelital, visión por computador, sistemas de inspección industrial y restauración de fotografías.

Sin embargo, la ecualización no siempre produce mejoras visuales. En algunas imágenes puede generar una apariencia artificial o inclusive amplificar ruido presente en regiones oscuras o brillantes. Por esta razón, su utilización debe evaluarse de acuerdo con la aplicación específica.

Ejemplo 22. Ecualización de imagen a escala de grises

A continuación, veremos un ejemplo en Python para ecualizar una imagen a escala de grises:

```python
import cv2
import numpy as np
import matplotlib.pyplot as plt

img = cv2.imread('/content/imagen_oscura.png', cv2.IMREAD_GRAYSCALE)
img_eq = cv2.equalizeHist(img_dark)

cv2.imwrite("imagen_oscura.png", img_dark)
cv2.imwrite("imagen_ecualizada.png", img_eq)

hist = cv2.calcHist([img], [0], None, [256], [0,256])
hist_eq = cv2.calcHist([img_eq], [0], None, [256], [0,256])

plt.figure(figsize=(10, 7))

# Imagen original
plt.subplot(2, 2, 1); plt.imshow(img, cmap="gray")
plt.title("Imagen original"); plt.axis("off")

# Imagen ecualizada
plt.subplot(2, 2, 2); plt.imshow(img_eq, cmap="gray")
plt.title("Imagen ecualizada"); plt.axis("off")

# Histograma original
plt.subplot(2, 2, 3); plt.plot(hist)
plt.title("Histograma original")
plt.xlabel("Nivel de intensidad")
plt.ylabel("Número de píxeles")
plt.xlim([0, 255])

# Histograma ecualizado
plt.subplot(2, 2, 4); plt.plot(hist_eq)
plt.title("Histograma ecualizado")
plt.xlabel("Nivel de intensidad")
plt.ylabel("Número de píxeles")
plt.xlim([0, 255])

plt.tight_layout()
plt.show()
```

Figura 73. Código en Python del Ejemplo 22: ecualización de una imagen a escala de grises.

La Figura 74 presenta el efecto de aplicar la ecualización de histograma sobre una imagen oscura, correspondiente a un paisaje de lago. A simple vista, podemos observar que la imagen original (superior izquierda) presenta niveles de intensidad muy bajos, lo que dificulta distinguir detalles en elementos como el árbol, el agua y las montañas del fondo. Esta situación también se refleja en el histograma original (inferior izquierda), cuya distribución se concentra principalmente en valores bajos de intensidad.

Después de aplicar la ecualización, la imagen presenta un mayor contraste y los detalles resultan más fáciles de observar. Esto se debe a que las intensidades se redistribuyeron y se tienen amplitudes en valores más altos de nivel de intensidad.

Una particularidad de los histogramas ecualizados es la aparición de “picos” en algunos niveles de intensidad. Por esta razón, es común que el histograma pase de una distribución suave a una distribución más irregular, con cambios abruptos entre niveles consecutivos.

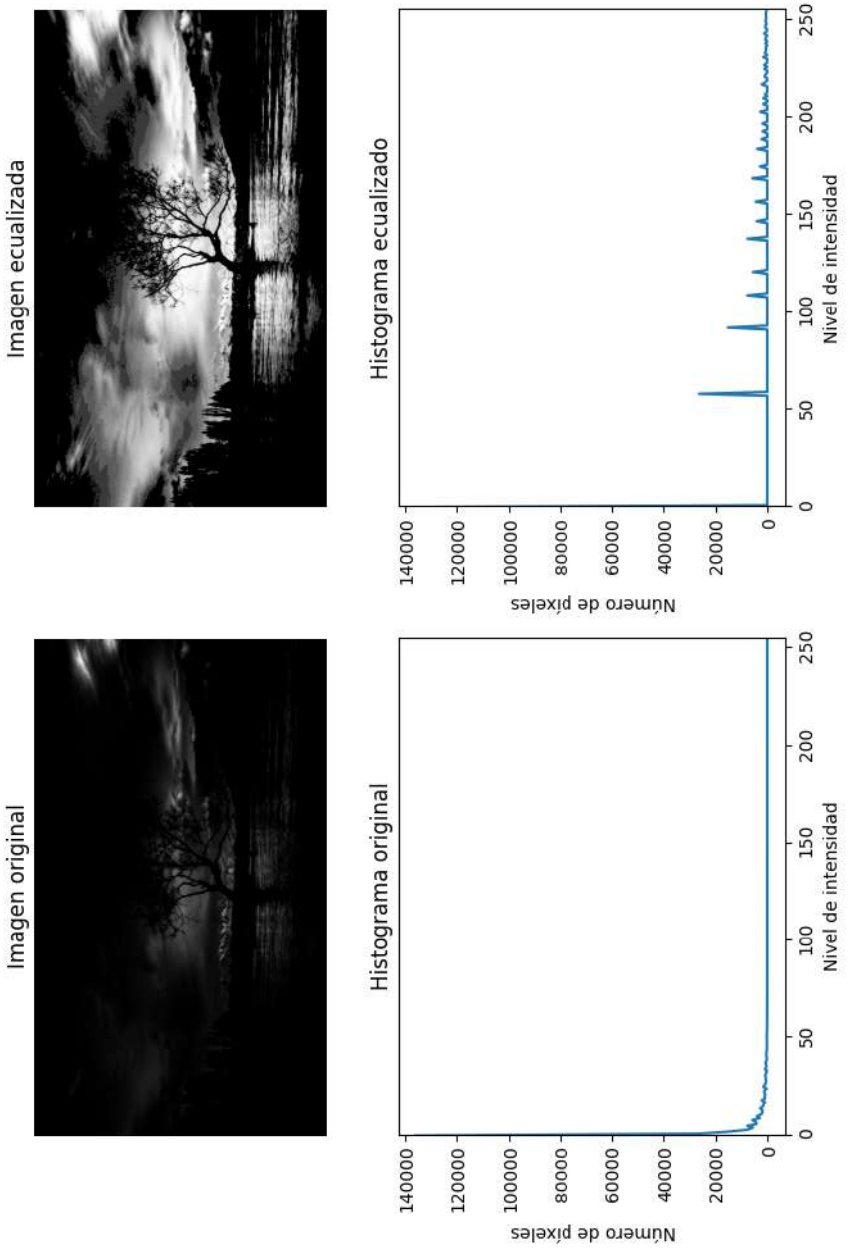


Figura 74. Ecualización de imagen a escala de grises (Ejemplo 22): imagen original vs. ecualizada (superior); histograma imagen original vs. histograma imagen ecualizada (inferior).

Ejemplo 23. Ecualización de imagen a color y modelo HSV

En el ejemplo anterior aplicamos la ecualización sobre una imagen en escala de grises. Sin embargo, en muchas aplicaciones resulta más común trabajar con imágenes a color. Y entonces, nos preguntamos:

¿Es posible aplicar la ecualización directamente sobre una imagen RGB?

La respuesta es sí, pero no suele ser la mejor alternativa. Aunque técnicamente es posible ecualizar cada canal de color por separado, esta práctica puede introducir cambios artificiales en la apariencia de la imagen. En consecuencia, los colores originales pueden alterarse y el resultado final puede verse poco natural.

Para evitar este inconveniente, en la práctica utilizanos el modelo de color **HSV (Hue, Saturation, Value)**, donde:

- **H (Hue):** representa el tono o color.
- **S (Saturation):** representa la saturación o pureza del color.
- **V (Value):** representa el brillo o intensidad.

La principal ventaja de este modelo es que separa la información de color de la información de brillo. De esta forma, podemos aplicar la ecualización únicamente sobre el canal **V**, mejorando el contraste de la imagen sin modificar significativamente sus colores originales.

La Figura 75 ilustra el proceso de ecualización de una imagen a color utilizando el modelo HSV.

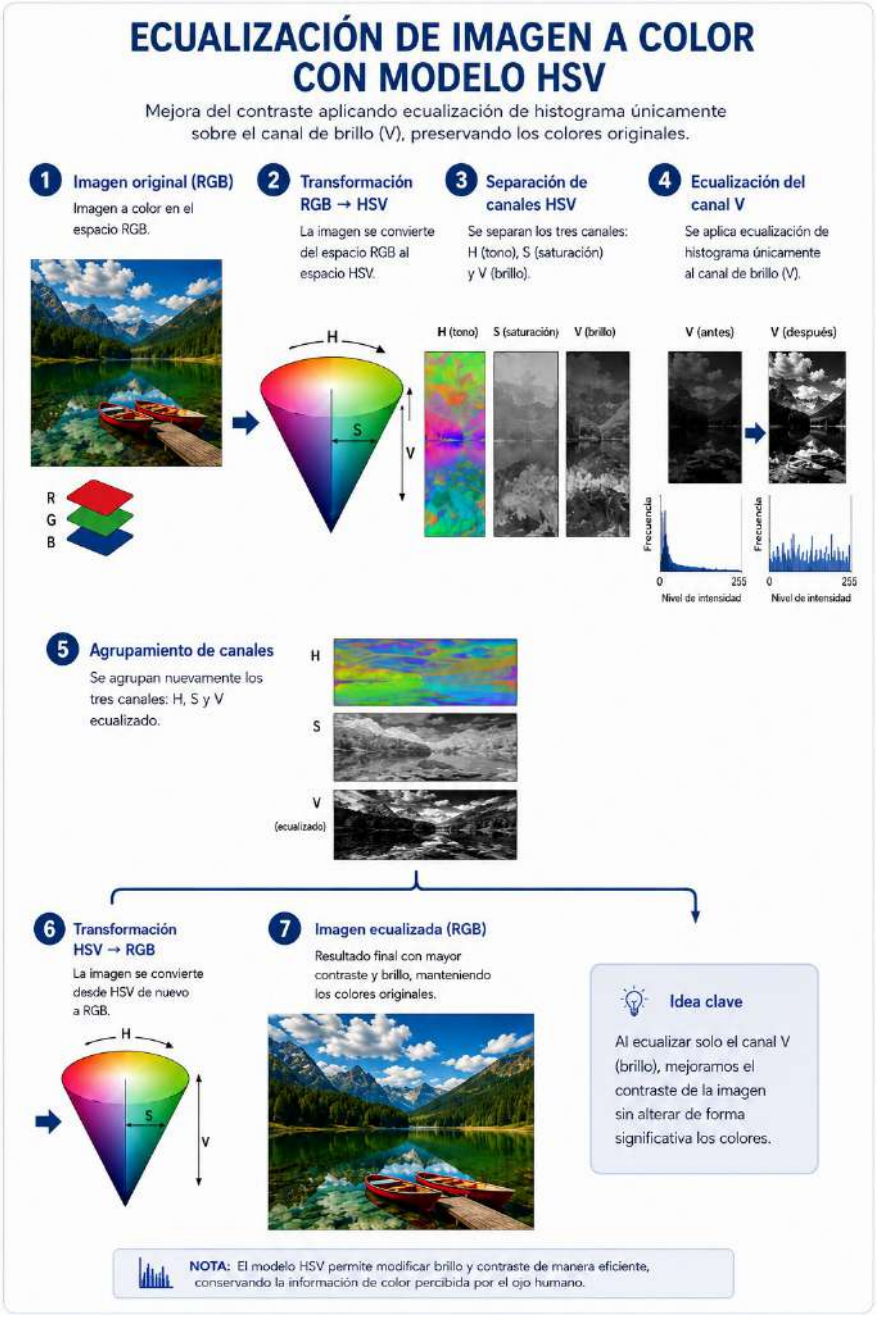


Figura 75. Proceso de ecualización de una imagen a color utilizando el modelo de color HSV.

A continuación, aplicaremos la ecualización sobre el canal V y compararemos el resultado con la imagen original.

```
import cv2
import matplotlib.pyplot as plt

imagen_bgr = cv2.imread("imagen_color.png") # Leer imagen en color
imagen_rgb = cv2.cvtColor(imagen_bgr, cv2.COLOR_BGR2RGB) # Convertir de BGR a RGB
imagen_hsv = cv2.cvtColor(imagen_bgr, cv2.COLOR_BGR2HSV) # Convertir de BGR a HSV
H, S, V = cv2.split(imagen_hsv) # Separar canales HSV
V_eq = cv2.equalizeHist(V) # Ecualizar únicamente el canal V
imagen_hsv_eq = cv2.merge([H, S, V_eq]) # Reconstruir imagen HSV
imagen_eq_rgb = cv2.cvtColor(imagen_hsv_eq, cv2.COLOR_HSV2RGB) # Convertir a RGB

# Histograma del canal 0 (R)
hist_original = cv2.calcHist([imagen_rgb], [0], None, [256], [0, 256])
hist_ecualizada = cv2.calcHist([imagen_eq_rgb], [0], None, [256], [0, 256])
plt.figure(figsize=(10, 8))

# Imagen original
plt.subplot(2, 2, 1);plt.imshow(imagen_rgb)
plt.title("Imagen original"); plt.axis("off")

# Imagen ecualizada
plt.subplot(2, 2, 2); plt.imshow(imagen_eq_rgb)
plt.title("Imagen ecualizada (HSV)"); plt.axis("off")

# Histograma original
plt.subplot(2, 2, 3); plt.plot(hist_original)
plt.title("Histograma original (Canal 0)"); plt.xlim([0, 256])
plt.xlabel("Nivel de intensidad"); plt.ylabel("Frecuencia")

# Histograma ecualizado
plt.subplot(2, 2, 4); plt.plot(hist_ecualizada)
plt.title("Histograma ecualizado (Canal 0)");plt.xlim([0, 256])
plt.xlabel("Nivel de intensidad"); plt.ylabel("Frecuencia")
```

Figura 76. Código en Python del Ejemplo 23: ecualización de imagen a color.

Obteniendo los siguientes resultados:

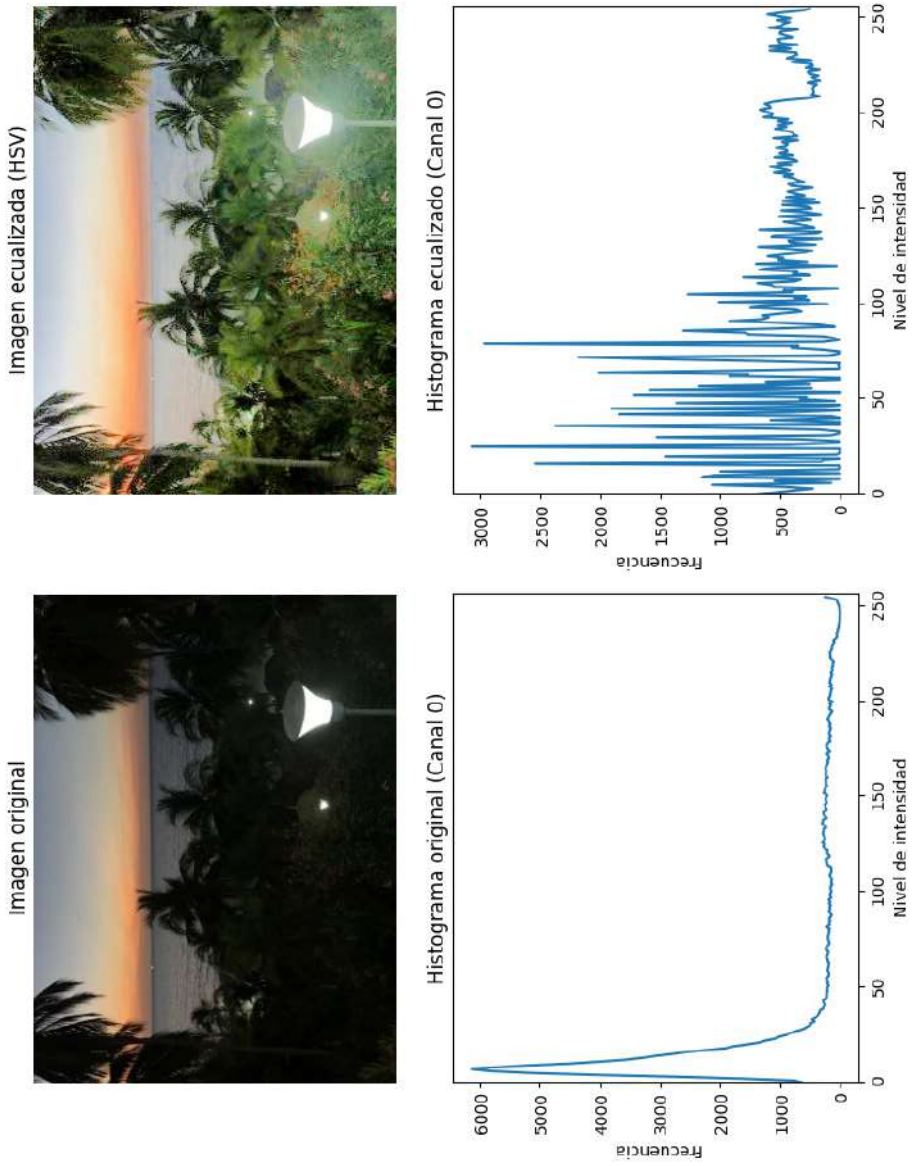


Figura 77. Ecualización de imagen a color (Ejemplo 23): imagen original vs. ecualizada (superior); histograma imagen original vs. histograma imagen ecualizada (inferior).

Como podemos observar, la imagen ecualizada presenta un mayor contraste, permitiendo apreciar con mayor claridad detalles que en la imagen original pasaban prácticamente desapercibidos, especialmente en las zonas más oscuras **(la puesta de sol ahora se ve mucho más hermosa)**. Esto se logra al redistribuir los niveles de intensidad del brillo de la imagen, aprovechando mejor el rango dinámico disponible.

Por otro lado, en la parte inferior comparamos el histograma del **canal R** antes y después de la ecualización. Podemos observar que el histograma original tiene una forma relativamente suave, con una alta concentración de píxeles en los niveles bajos de intensidad, característica de una imagen oscura. Después de la ecualización, el histograma cambia de forma y aparecen numerosos picos. Esto es completamente normal. La ecualización no busca generar una curva uniforme, sino redistribuir los niveles de intensidad para mejorar el contraste de la imagen. Por esta razón, algunos niveles de intensidad dejan de utilizarse, mientras que otros concentran un mayor número de píxeles, dando lugar a ese patrón de picos que ahora observamos.

4.4 Detección de bordes y segmentación

Hasta este momento hemos trabajado con técnicas orientadas a mejorar la apariencia de una imagen. Sin embargo, en muchas aplicaciones el objetivo no es únicamente obtener una imagen más agradable a la vista (por ejemplo, más clara), sino extraer información útil de ella. Pensemos en un vehículo autónomo que debe identificar el borde de la carretera, un sistema médico que necesita delimitar un órgano en una radiografía o un robot industrial que debe reconocer la posición de una pieza antes de manipularla.

Para ello aparecen dos conceptos fundamentales del procesamiento digital de imágenes: **la detección de bordes** y **la segmentación**. Aunque ambos buscan facilitar la interpretación de una imagen, sus objetivos son diferentes. La detección de bordes intenta localizar los cambios bruscos de intensidad que suelen indicar el límite entre dos regiones, mientras que la segmentación busca agrupar píxeles con características similares para separar los diferentes objetos presentes en la escena.

La siguiente figura presenta una breve comparación entre ambos conceptos.

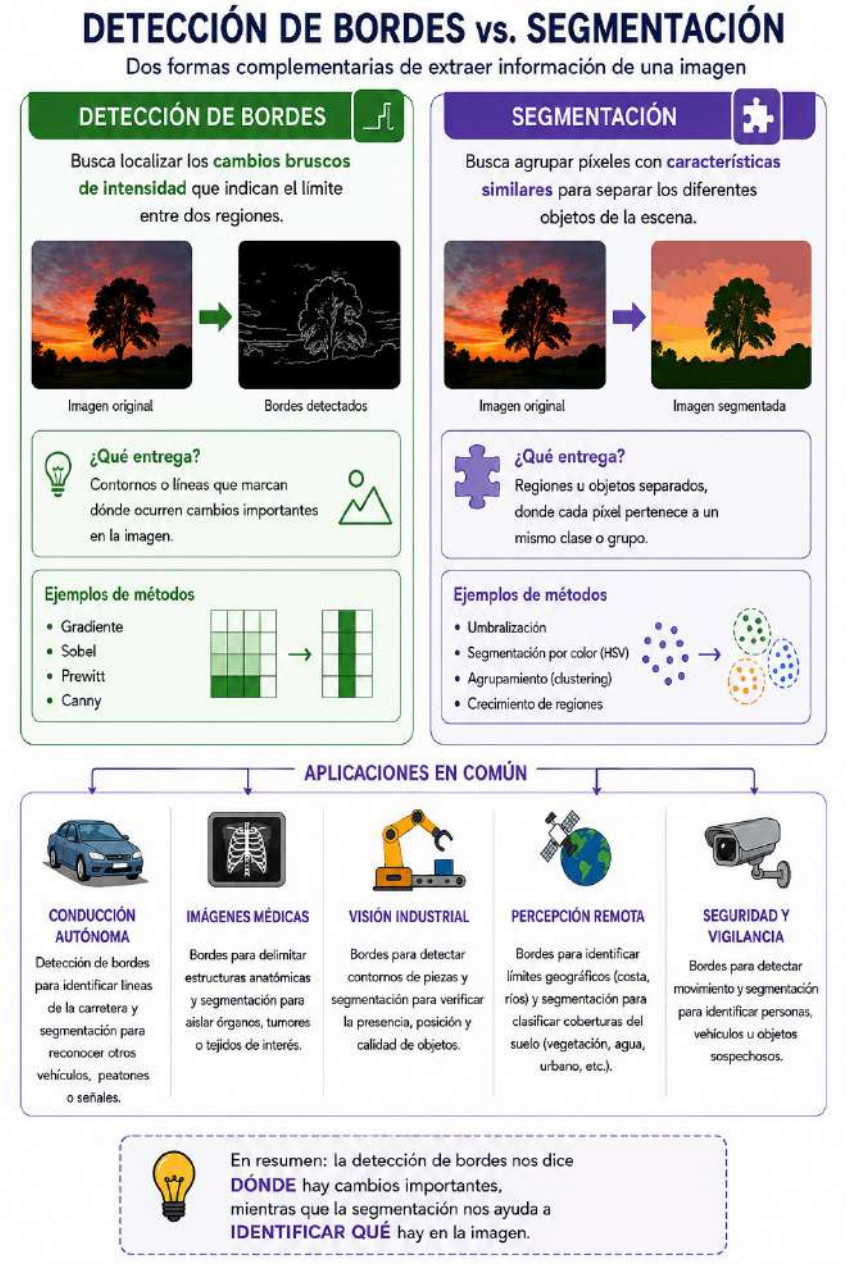


Figura 78. Comparación entre detección de bordes y segmentación.

Ahora, imagina que tenemos una fotografía como la que aparece en

la figura anterior: un árbol durante la puesta de sol. Si el objetivo es conocer la silueta del árbol, lo más importante será identificar dónde cambia bruscamente la intensidad entre el árbol y el cielo; es decir, detectar sus bordes. En cambio, si el objetivo es separar completamente el árbol del fondo para analizarlo o medir su tamaño, será necesario segmentar la imagen, asignando cada píxel al objeto o al fondo según sus características.

4.4.1 Detección de bordes

Una forma sencilla de identificar un borde consiste en comparar la intensidad de píxeles vecinos. Cuando la diferencia entre ellos es grande, es probable que exista un borde. Para ello se utilizan diferentes **máscaras** o **kernels**, los cuales detectan estos cambios mediante una convolución entre el kernel y la imagen (estrictamente, una correlación). Cada kernel está diseñado para resaltar cambios de intensidad en una dirección específica, como la horizontal, la vertical o la diagonal. La Figura 79 presenta algunos de los kernels clásicos utilizados para la detección de bordes.

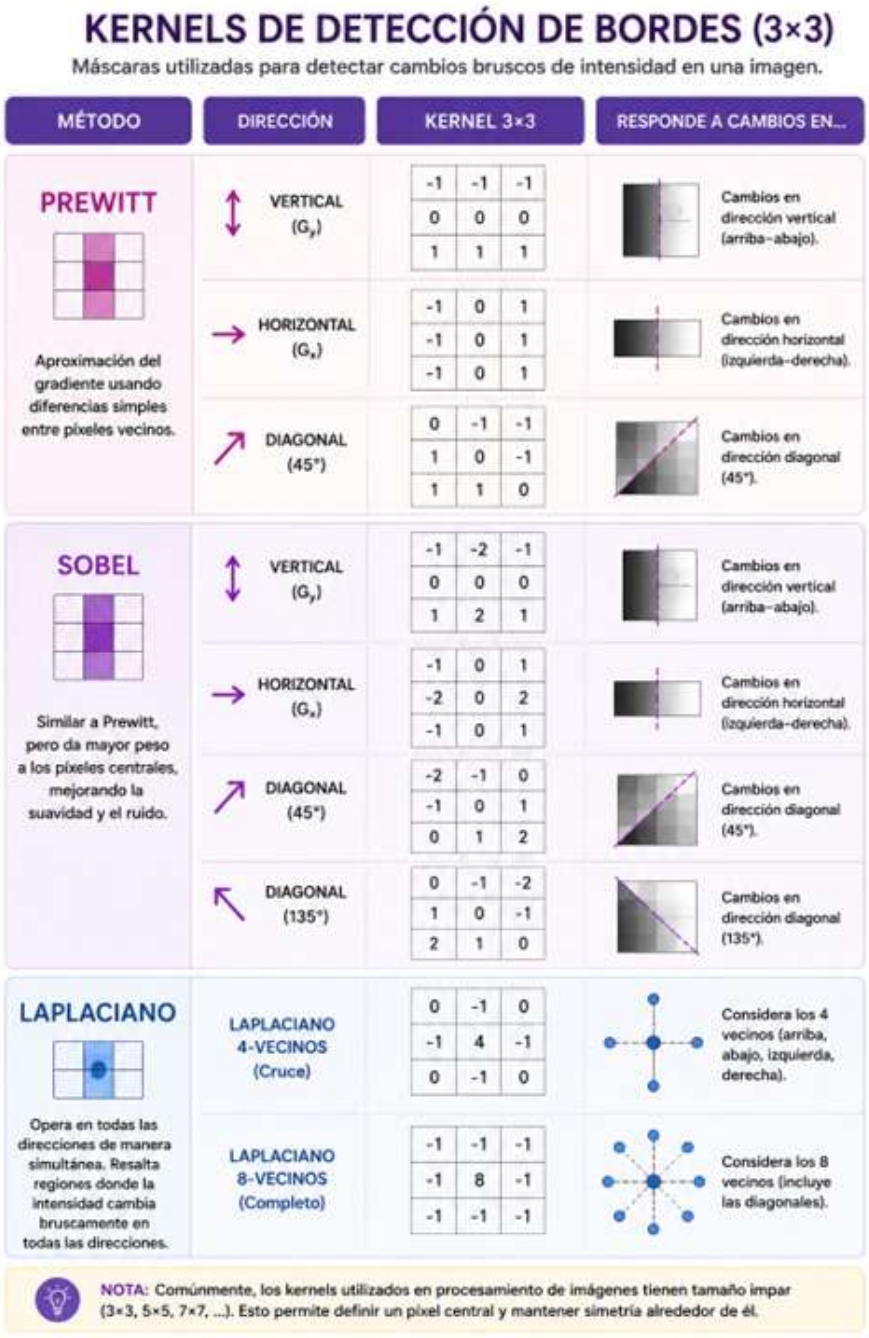


Figura 79. Kernels clásicos para detección de bordes: prewitt, sobel y laplaciano.

Los operadores **Prewitt** y **Sobel** permiten detectar cambios de intensidad en direcciones específicas, como la horizontal, la vertical o la diagonal. **Por ejemplo, para detectar bordes horizontales se utiliza un kernel compuesto por una fila de valores negativos, seguida de una fila de ceros y, finalmente, una fila de valores positivos.** La principal diferencia entre ellos es que Sobel asigna un mayor peso a los píxeles centrales, lo que lo hace ligeramente más robusto frente al ruido. Por su parte, el **operador Laplaciano** no detecta una dirección particular, sino que responde a cambios de intensidad en todas las direcciones, resaltando los contornos independientemente de su orientación.

¿Pero qué tienen en común los diferentes kernels?

Todos presentan un cambio de polaridad entre sus coeficientes y una suma total igual a cero, lo que les permite responder a cambios bruscos de intensidad. Sin embargo, en aplicaciones reales no suele aplicarse un único *kernel* de forma aislada. En su lugar, es común utilizar algoritmos más robustos, como el **método Canny**, el cual combina varias etapas de procesamiento para obtener bordes más precisos y continuos, reduciendo además la influencia del ruido. La Figura 80 resume su funcionamiento general.

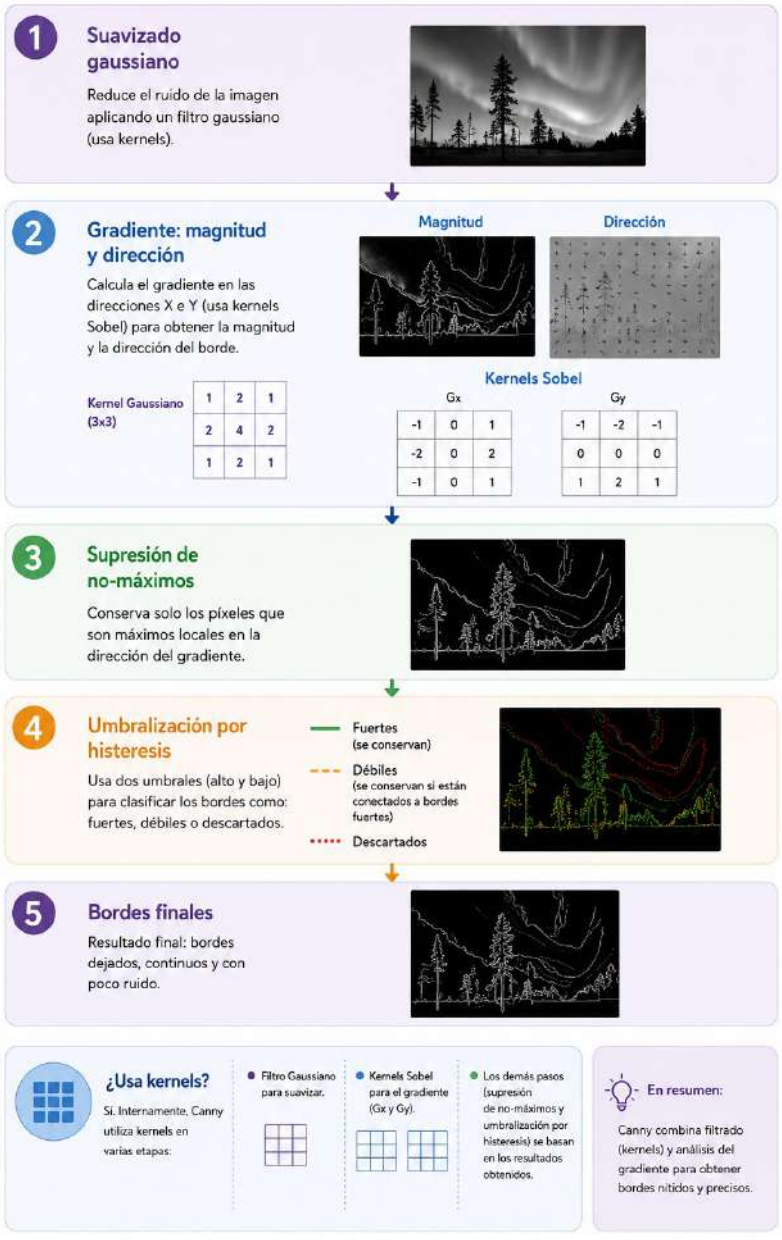


Figura 80. Algoritmo Canny: suavizado gaussiano, gradiente de magnitud y dirección, supresión de no-máximos, umbralización por histéresis y bordes finales.

Como podemos observar, el algoritmo **Canny** no se limita a aplicar un único kernel sobre la imagen. En su lugar, realiza una secuencia de etapas que incluyen el suavizado de la imagen, el cálculo del gradiente mediante **kernels** (como los operadores **Sobel** en las direcciones horizontal y vertical), la supresión de respuestas no máximas y la aplicación de umbrales para conservar únicamente los bordes más relevantes. El suavizado inicial se realiza mediante un kernel gaussiano, el cual estudiaremos al finalizar este capítulo.

Ejemplo 24. Detección de bordes con el algoritmo Canny

Vamos ahora a aplicar el algoritmo Canny para la detección de bordes en una imagen a color. En este ejemplo generaremos tres imágenes de salida, ya que utilizaremos tres pares de umbrales dentro del algoritmo: uno para una alta sensibilidad a los bordes, otro para una sensibilidad media y un último para una sensibilidad baja. A continuación, se presenta el código en Python.

```
import cv2
import matplotlib.pyplot as plt

imagen_bgr = cv2.imread("aurora_boreal.png") # Leer imagen
imagen_rgb = cv2.cvtColor(imagen_bgr, cv2.COLOR_BGR2RGB) # Convertir a RGB
imagen_gray = cv2.cvtColor(imagen_bgr, cv2.COLOR_BGR2GRAY) # Escala de grises

canny1 = cv2.Canny(imagen_gray, 20, 60) # Alta sensibilidad
canny2 = cv2.Canny(imagen_gray, 40, 120) # Sensibilidad media
canny3 = cv2.Canny(imagen_gray, 100, 240) # Baja sensibilidad

plt.figure(figsize=(10,8))

plt.subplot(2,2,1); plt.imshow(imagen_rgb)
plt.title("Imagen original"); plt.axis("off")

plt.subplot(2,2,2); plt.imshow(canny1, cmap="gray")
plt.title("Canny (20,60)"); plt.axis("off")

plt.subplot(2,2,3); plt.imshow(canny2, cmap="gray")
plt.title("Canny (40,120)"); plt.axis("off")

plt.subplot(2,2,4); plt.imshow(canny3, cmap="gray")
plt.title("Canny (100,240)"); plt.axis("off")

plt.tight_layout(); plt.show()
```

Figura 81. Segmentación por color utilizando el modelo HSV.

Obteniendo:

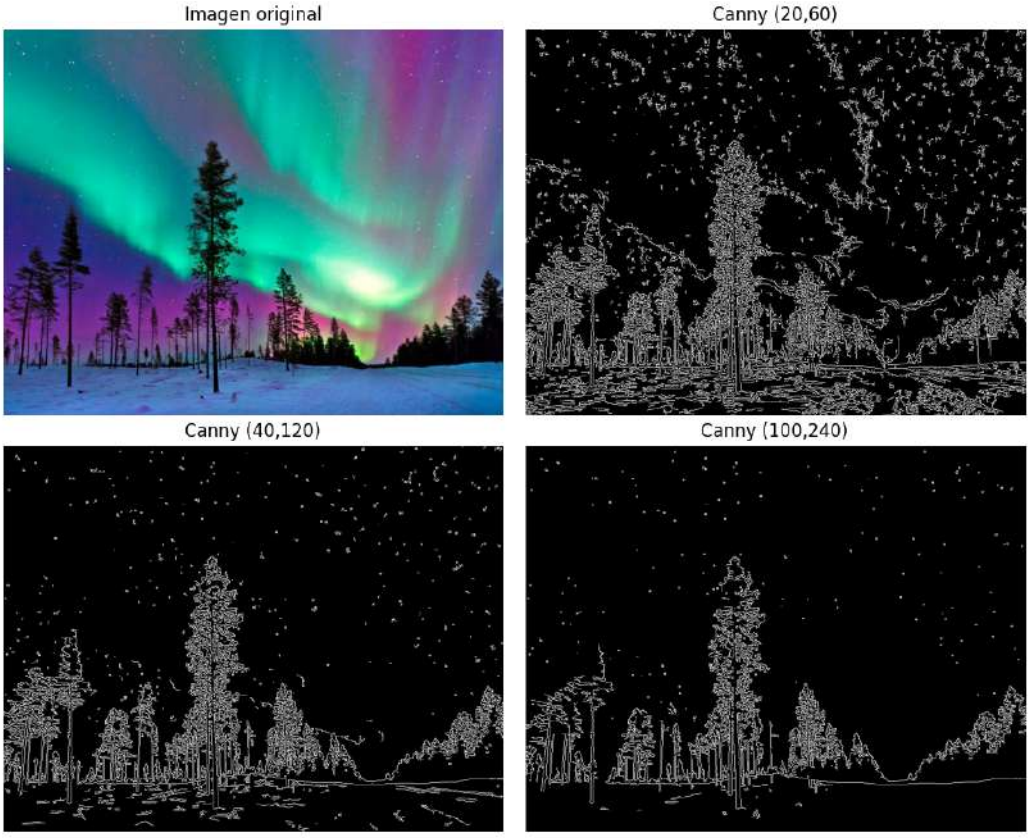


Figura 82. Imagen original (aurora boreal) y tres imágenes de detección de bordes utilizando el algoritmo Canny.

En la primera imagen de detección de bordes (superior derecha), correspondiente a los umbrales **20** y **60**, se observa una alta sensibilidad del algoritmo. Como resultado, se detectan numerosos bordes en la zona del bosque y también múltiples variaciones de intensidad presentes en la aurora boreal y en las estrellas, generando una imagen con una gran cantidad de detalles.

En la segunda imagen (inferior izquierda), obtenida con los umbrales **40** y **120**, disminuye el número de bordes detectados, conservándose principalmente los contornos más representativos del bosque y del horizonte. Finalmente, en la tercera imagen (inferior derecha), utilizando los umbrales **100** y **240**, únicamente

permanecen los bordes con mayor contraste, desapareciendo casi por completo las estructuras asociadas a la aurora boreal.

Como podemos observar, el resultado depende en gran medida de los dos umbrales utilizados por el algoritmo Canny: el **umbral inferior**, que determina la sensibilidad para detectar bordes débiles, y el **umbral superior**, que establece el nivel mínimo para considerar un borde como fuerte. Por esta razón, la selección de estos parámetros debe realizarse de acuerdo con las características de la imagen y el nivel de detalle que se desea conservar.

Ahora, pasaremos al concepto de segmentación.

4.4.2 Segmentación de color con el modelo HSV

Por otro lado, segmentar consiste en asignar cada píxel a una determinada región, facilitando así el análisis de los diferentes elementos presentes en la escena. Pensemos nuevamente en la fotografía del árbol durante la puesta de sol. Detectar los bordes nos permite conocer la silueta del árbol, mientras que segmentarlo significa separar todos los píxeles que pertenecen al árbol del resto de la imagen, obteniendo una nueva imagen donde el árbol aparece completamente aislado del fondo.

Existen diferentes criterios para realizar una segmentación. En algunos casos se utiliza la intensidad de los píxeles; en otros, el color, la textura o incluso información de profundidad. En este capítulo nos concentraremos en la **segmentación por color**, una de las técnicas más utilizadas por su simplicidad y eficiencia. Para lo cual, trabajamos con el modelo de color HSV (Figura 83).

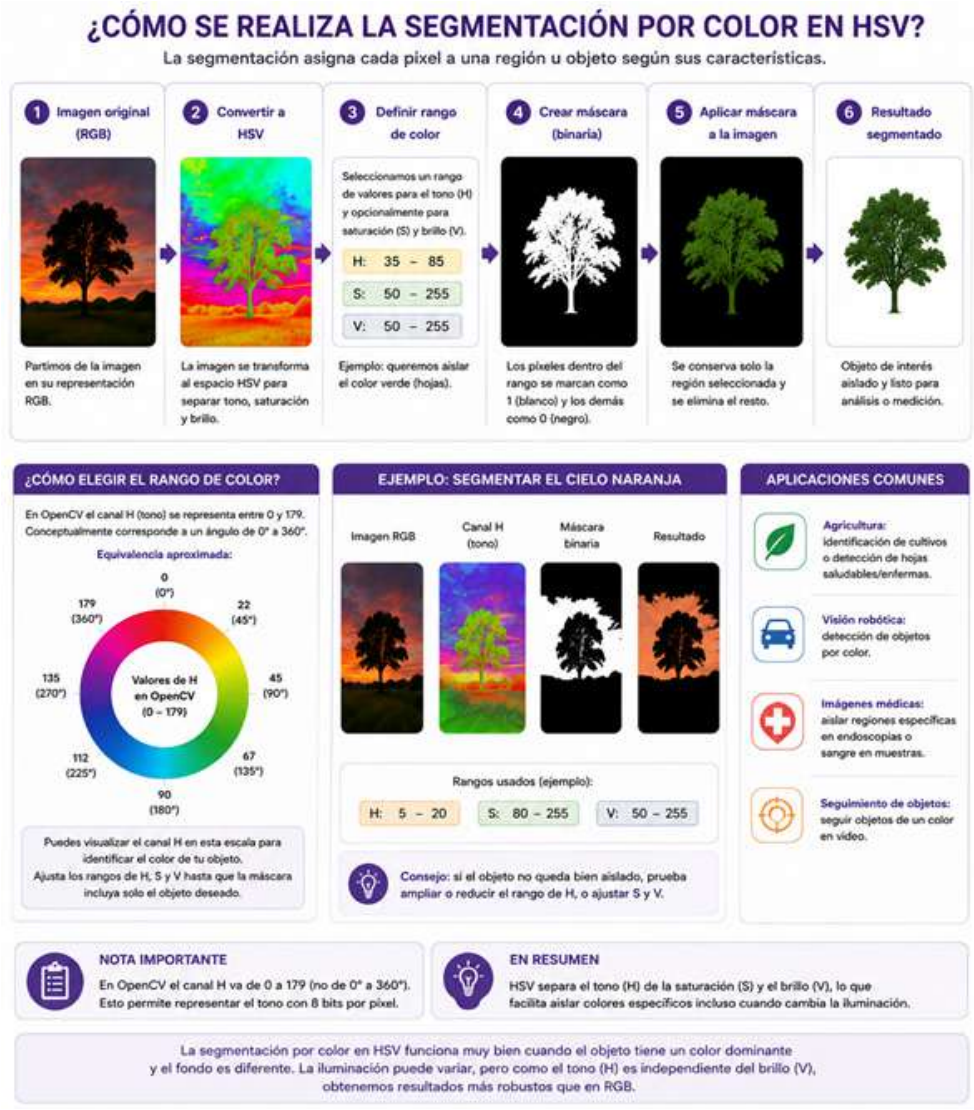


Figura 83. Segmentación por color utilizando el modelo HSV.

Supongamos que queremos segmentar el cielo con las tonalidades naranjas presentes durante la puesta de sol. Para ello definimos un rango de valores en el espacio HSV. El parámetro H (Hue) corresponde al tono o color y es el más importante en la segmentación por color. En este caso se selecciona el intervalo H = [5, 20], el cual abarca las tonalidades naranjas de la imagen (véase

la asignación de colores en la gráfica radial de la Figura 80). Los píxeles cuyo tono se encuentre fuera de este intervalo serán descartados.

Como recomendación, los valores mínimos de S (Saturation) y V (Value) no deberían ser demasiado bajos, ya que esto podría incluir píxeles poco saturados o con muy baja iluminación que no pertenecen al objeto de interés. En la práctica, estos parámetros los ajustamos de forma empírica según la imagen y las condiciones de iluminación.

En resumen, el parámetro **H** determina **qué color** se desea segmentar, mientras que **S** y **V** actúan como filtros adicionales que permiten eliminar regiones con poca saturación o escasa iluminación. Por esta razón, en la práctica normalmente ajustamos primero el rango de **H** y, posteriormente, refinamos los valores de **S** y **V** hasta obtener una segmentación adecuada.

Ejemplo 25. Segmentación de imagen con modelo HSV

En el **Ejemplo 24** vimos que el algoritmo **Canny** permite detectar bordes a partir de cambios de intensidad, resaltando principalmente los contornos de la zona del bosque. Sin embargo, en este ejemplo el propósito es completamente diferente: identificar la región de la imagen correspondiente a la **aurora boreal**, la cual presenta predominantemente tonalidades **verde turquesa (verde cian)**. Para ello utilizaremos el modelo **HSV**, definiendo un rango de valores para los parámetros **H**, **S** y **V** que permita segmentar dichas tonalidades.

Utilizaremos el siguiente código en Python:

```python
import cv2
import numpy as np
import matplotlib.pyplot as plt

imagen_bgr = cv2.imread("aurora_boreal.png") # Leer imagen
imagen_rgb = cv2.cvtColor(imagen_bgr, cv2.COLOR_BGR2RGB) # Convertir a RGB
imagen_hsv = cv2.cvtColor(imagen_bgr, cv2.COLOR_BGR2HSV) # Convertir a HSV

lim_inf = np.array([35, 60, 50]) # Límite inferior HSV
lim_sup = np.array([90, 255, 255]) # Límite superior HSV

mascara = cv2.inRange(imagen_hsv, lim_inf, lim_sup) # Crear máscara binaria
segmentada = cv2.bitwise_and(imagen_rgb, imagen_rgb, mask=mascara) # Aplicar máscara

plt.figure(figsize=(10, 8))
plt.subplot(2, 2, 1); plt.imshow(imagen_rgb)
plt.title("Imagen original"); plt.axis("off")
plt.subplot(2, 2, 2); plt.imshow(imagen_hsv[:, :, 0], cmap="hsv")
plt.title("Canal H"); plt.axis("off")
plt.subplot(2, 2, 3); plt.imshow(mascara, cmap="gray")
plt.title("Máscara HSV"); plt.axis("off")
plt.subplot(2, 2, 4); plt.imshow(segmentada)
plt.title("Aurora segmentada"); plt.axis("off")
plt.tight_layout(); plt.show()
```

Figura 84. Código en Python del Ejemplo 25: segmentación de imagen con HSV.

Antes de analizar los valores utilizados, observemos cómo se construyen los rangos de **H**, **S** y **V**. En el código definimos dos vectores: un **límite inferior** (lim_inf) y un **límite superior** (lim_sup). El primer elemento de cada vector corresponde al parámetro H (Hue), el segundo a S (Saturation) y el tercero a V (Value). Por lo tanto, el rango de cada parámetro se obtiene tomando el valor correspondiente del límite inferior y del límite superior. En este ejemplo se tiene:

- H = [35, 90]
- S = [60, 255]
- V = [50, 255]

Es decir, un píxel será seleccionado únicamente si su valor de **H** está entre **35 y 90**, su valor de **S** entre **60 y 255** y su valor de **V** entre **50 y 255**. Si alguno de estos tres parámetros queda por fuera de su intervalo, el píxel no hará parte de la segmentación.

En este ejemplo, el intervalo **H = [35, 90]** fue seleccionado porque corresponde a las tonalidades **verde turquesa** presentes en la aurora boreal. Recordemos que, en OpenCV, el canal **H** toma valores entre **0 y 179**, por lo que este rango abarca la mayor parte de las tonalidades verdes observadas en la imagen. Por su parte, el rango **S = [60, 255]** descarta píxeles con muy poca saturación, mientras que **V = [50, 255]** elimina regiones demasiado oscuras. De esta forma, la segmentación se concentra principalmente en las zonas de la aurora con un color verde bien definido.

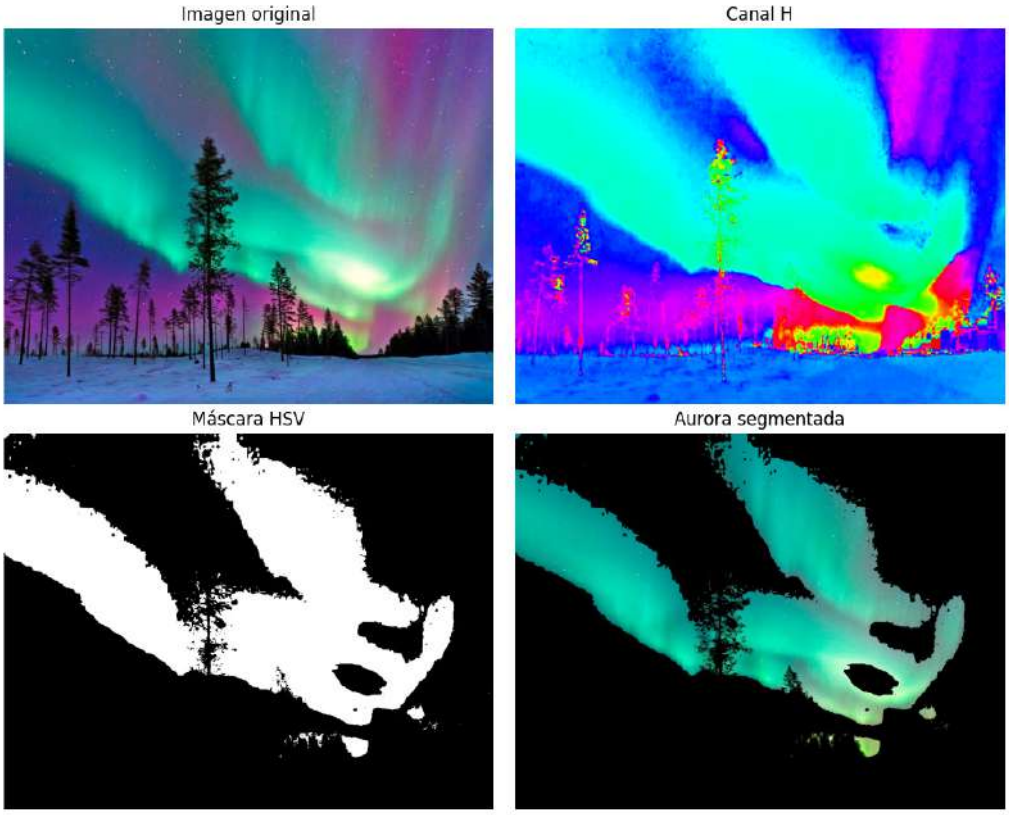


Figura 85. Imagen segmentada enfocada en la zona de aurora boreal.

La máscara binaria actúa como un filtro sobre la imagen original. Conceptualmente, puede interpretarse como una multiplicación píxel a píxel, donde los píxeles blancos de la máscara conservan el color original de la imagen, mientras que los píxeles negros se convierten en negro. Por lo cual, para nuestro ejemplo, solamente las zonas en blanco de la máscara HSV (inferior izquierda) al multiplicarse con la imagen original producen la imagen segmentada en la que aparece en negro la zona descartada y en verde turquesa la zona de interés.

Pero qué pasaría si seleccionamos de forma equivocada el rango para H. Vamos a ver el efecto con los siguientes umbrales:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

imagen_bgr = cv2.imread("aurora_boreal.png") # Leer imagen
imagen_rgb = cv2.cvtColor(imagen_bgr, cv2.COLOR_BGR2RGB) # Convertir a RGB
imagen_hsv = cv2.cvtColor(imagen_bgr, cv2.COLOR_BGR2HSV) # Convertir a HSV

# Rangos de H diferentes, manteniendo S y V constantes
lim_inf_1 = np.array([20, 60, 50]); lim_sup_1 = np.array([40, 255, 255])
lim_inf_2 = np.array([35, 60, 50]); lim_sup_2 = np.array([90, 255, 255])
lim_inf_3 = np.array([90, 60, 50]); lim_sup_3 = np.array([130, 255, 255])

mascara_1 = cv2.inRange(imagen_hsv, lim_inf_1, lim_sup_1) # Rango H bajo
mascara_2 = cv2.inRange(imagen_hsv, lim_inf_2, lim_sup_2) # Rango H adecuado
mascara_3 = cv2.inRange(imagen_hsv, lim_inf_3, lim_sup_3) # Rango H alto

seg_1 = cv2.bitwise_and(imagen_rgb, imagen_rgb, mask=mascara_1)
seg_2 = cv2.bitwise_and(imagen_rgb, imagen_rgb, mask=mascara_2)
seg_3 = cv2.bitwise_and(imagen_rgb, imagen_rgb, mask=mascara_3)
```

Figura 86. Código en Python del Ejemplo 25: segmentación de imagen con HSV, utilizando tres umbrales diferentes.

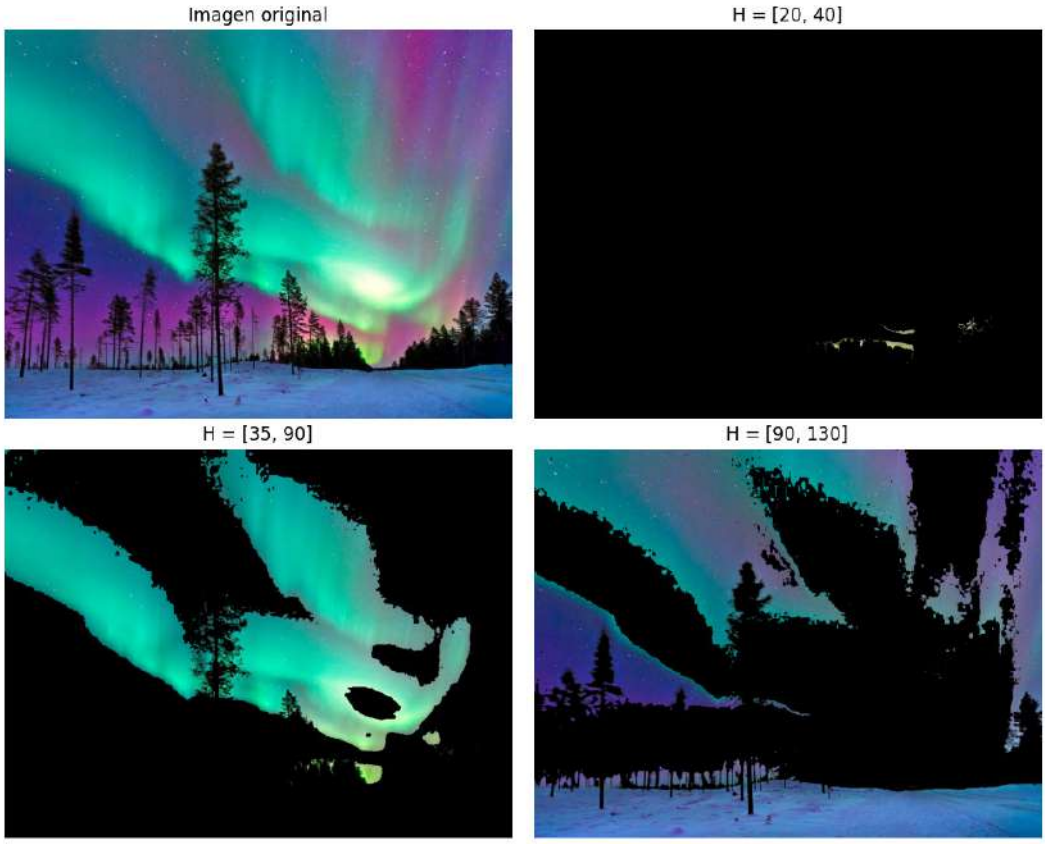


Figura 87. Imagen segmentada: comparación de resultados con tres umbrales diferentes.

La Figura 87 muestra el efecto de modificar únicamente el rango del parámetro **H (Hue)**, manteniendo constantes los valores de **S** y **V**. Como podemos observar, una selección inadecuada del rango de **H** puede alterar significativamente el resultado de la segmentación.

En la imagen superior derecha se utilizó el intervalo **H = [20, 40]**. Este rango no corresponde a las tonalidades predominantes de la aurora boreal, por lo que prácticamente toda la imagen es descartada y únicamente permanecen unas pocas regiones aisladas. En la imagen inferior izquierda se empleó el intervalo **H = [35, 90]**, el cual coincide con las tonalidades **verde turquesa**

de la aurora. Como resultado, la mayor parte de la aurora es correctamente segmentada, mientras que el resto de la escena se elimina.

Finalmente, en la imagen inferior derecha se utilizó el intervalo **H = [90, 130]**. En este caso, el algoritmo comienza a seleccionar tonalidades diferentes al verde turquesa, incorporando regiones que no pertenecen al objeto de interés y perdiendo simultáneamente parte de la aurora. Este ejemplo demuestra que la correcta selección del rango de **H** es el aspecto más importante de la segmentación por color en el espacio HSV.

Al comparar los resultados del Ejemplo 24 y del Ejemplo 25, nos damos cuenta de que la detección de bordes responde a la pregunta "¿dónde cambian los píxeles?", mientras que la segmentación responde a la pregunta "¿qué píxeles pertenecen al objeto de interés?". Por esta razón, la detección de bordes es especialmente útil para identificar contornos y formas, mientras que la segmentación permite separar regiones completas para su posterior análisis, medición o reconocimiento.

Ejemplo 26. El rojo, un caso especial en HSV

Y para finalizar la temática de segmentación por color, resumiremos algunos de los rangos de color más utilizados dentro del modelo HSV, los cuales servirán como referencia al momento de diseñar máscaras de segmentación. La Figura 88 presenta algunos de estos rangos.

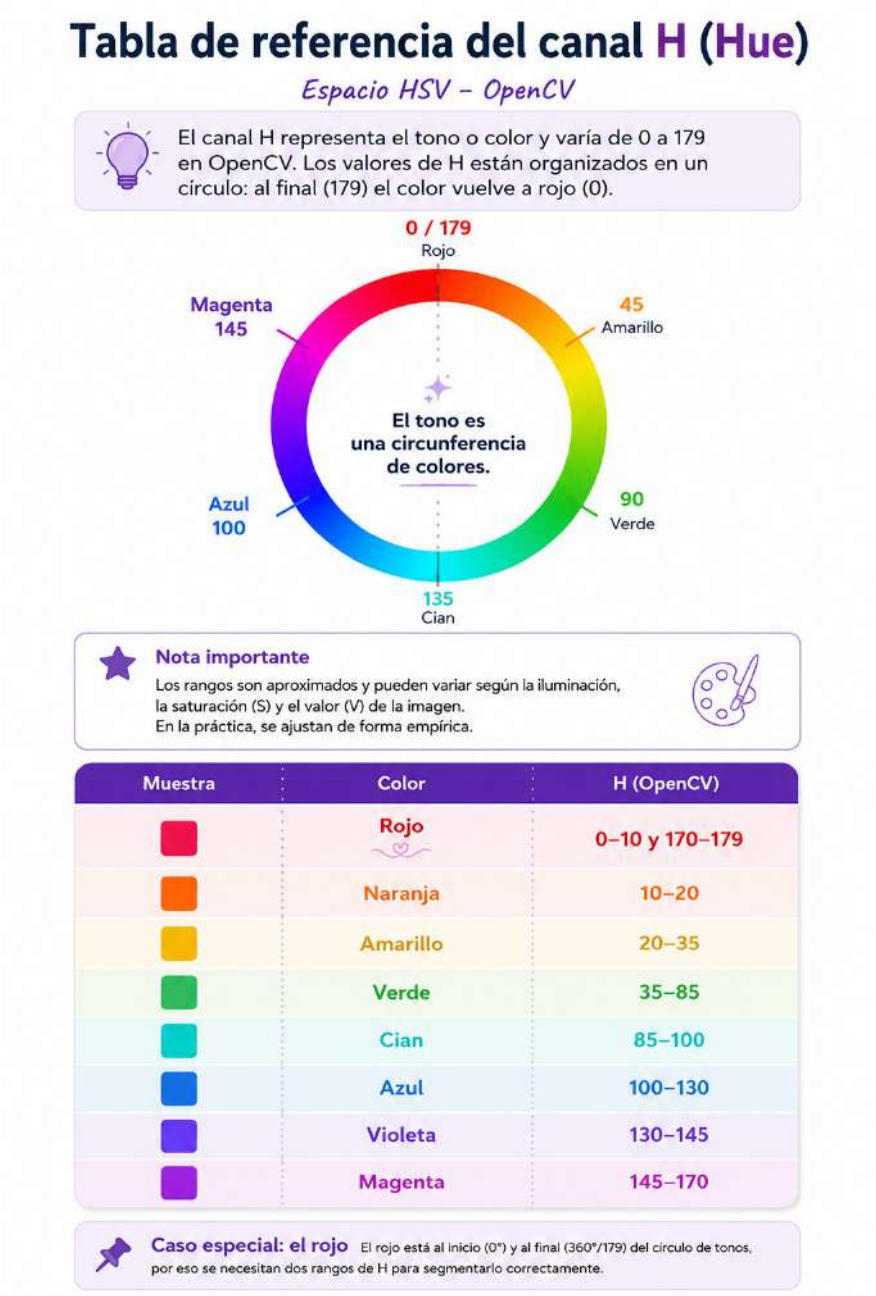


Figura 88. Rangos de referencia del canal **H (Hue)** para algunos colores en OpenCV.

Como podemos observar, el color **rojo** constituye un caso especial, ya que se encuentra ubicado tanto al inicio como al final del círculo de color. Por esta razón, cuando se desea segmentar este color es necesario generar **dos máscaras**, una para cada intervalo de **H**, y posteriormente combinarlas en una sola. El procedimiento general se presenta en la Figura 89.

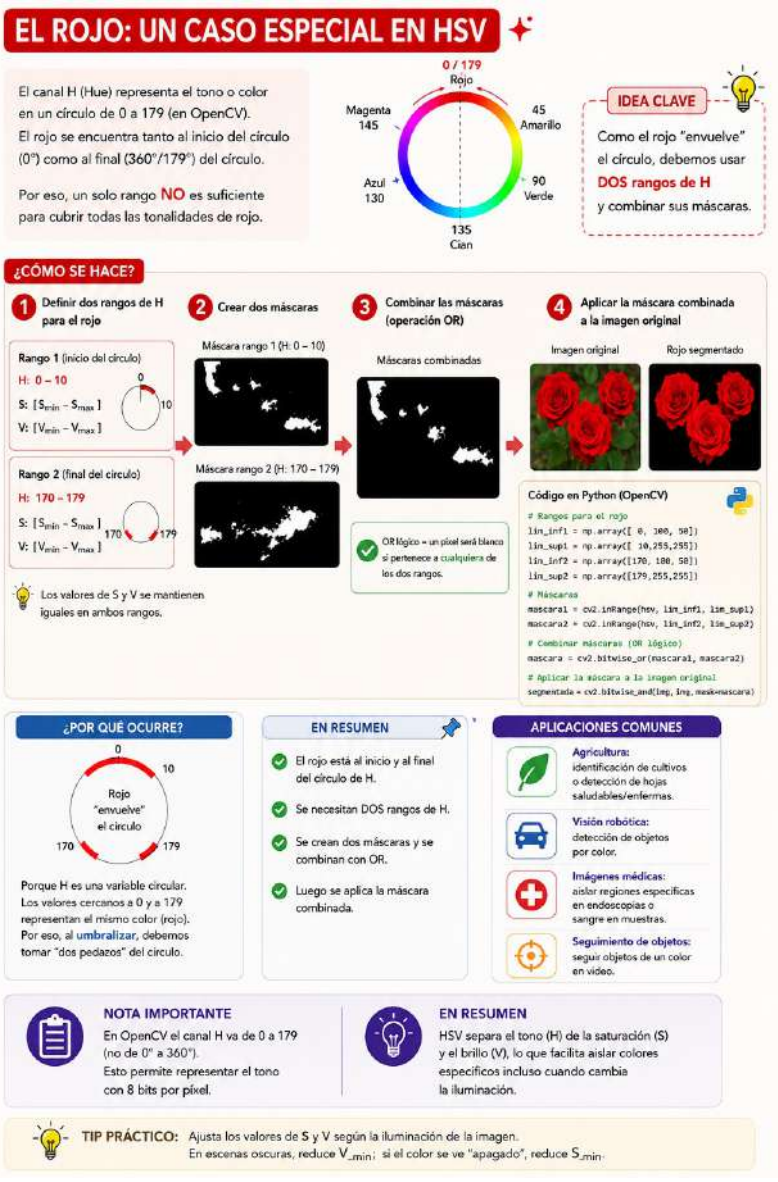


Figura 89. Caso especial de la segmentación del color rojo en HSV.

De esta manera, al fusionar ambas máscaras, es posible segmentar correctamente todas las regiones de la imagen que corresponden al color rojo.

4.5. Ruido en Imágenes y Filtrado

Hasta este momento hemos estudiado diferentes técnicas para mejorar la apariencia de una imagen, detectar sus bordes y segmentar regiones de interés. Sin embargo, en aplicaciones reales las imágenes rara vez son perfectas. Es común que durante el proceso de adquisición, transmisión o almacenamiento aparezcan alteraciones conocidas como **ruido**, las cuales pueden degradar la calidad visual de la imagen y afectar el desempeño de algoritmos como la detección de bordes o la segmentación.

En esta sección estudiaremos los principales tipos de ruido presentes en imágenes digitales y realizaremos una breve introducción a los filtros espaciales utilizados para reducir su efecto. Estos conceptos servirán como base para el caso de estudio que desarrollaremos en el siguiente capítulo.

4.5.1 Tipos de ruido

Seguramente has observado fotografías que presentan una apariencia granulada o pequeños puntos distribuidos sobre la imagen, especialmente cuando fueron tomadas en condiciones de poca iluminación. Estas alteraciones reciben el nombre de **ruido** y corresponden a modificaciones no deseadas de los valores originales de los píxeles. Su origen puede estar asociado al sensor de la cámara, las condiciones de iluminación, el proceso de transmisión o incluso la compresión y el almacenamiento de la imagen.

Desde un punto de vista matemático, una forma sencilla de representar el ruido consiste en asumir que este se suma a la imagen original. A este modelo se le conoce como **modelo aditivo de ruido** y podemos expresarlo mediante la siguiente ecuación:

$$g(x,y) = f(x,y) + n(x,y)$$

donde:

- $f(x,y)$: imagen original. Donde x,y corresponden a fila y columna, respectivamente.
- $n(x,y)$: ruido.
- $g(x,y)$: imagen observada o imagen con ruido.

En otras palabras, cada píxel de la imagen original puede verse alterado por una pequeña perturbación aleatoria, dando lugar a la imagen que finalmente observamos.

Aunque existen diferentes modelos de ruido, en este libro estudiaremos los tres más utilizados en procesamiento digital de imágenes: ruido gaussiano, ruido uniforme y ruido sal y pimienta.



Figura 90. Ruido en imagen: gaussiano, uniforme, sal y pimienta.

Como podemos observar, cada tipo de ruido modifica la imagen de una forma diferente. El **ruido gaussiano** produce una apariencia similar a una imagen envejecida o ligeramente desenfocada, donde los bordes pierden parte de su nitidez (como una fotografía impresa que guardamos durante muchos años). El ruido uniforme también afecta toda la imagen, pero genera una textura más homogénea, como si se hubiera añadido un grano fino distribuido de manera uniforme. Finalmente, el ruido sal y pimienta se caracteriza por la aparición de píxeles blancos y negros aislados (como si la imagen tuviera pecas), los cuales resultan especialmente visibles en regiones de color uniforme, como el cielo.

A simple vista podríamos pensar que todos estos tipos de ruido pueden eliminarse utilizando el mismo filtro. Sin embargo, cada uno presenta características diferentes y, por tanto, requiere una estrategia de filtrado específica. La siguiente sección introduce algunos de los filtros espaciales más utilizados para reducir el efecto del ruido.

4.5.2 Tipos de filtro

Al igual que en la detección de bordes, la mayoría de los filtros espaciales utilizan un kernel o máscara que se desplaza sobre toda la imagen realizando una convolución (estrictamente, una correlación). Sin embargo, el objetivo ahora es completamente diferente. Mientras que en la detección de bordes se buscaba resaltar los cambios bruscos de intensidad, en el filtrado el propósito es reducir el ruido, preservando en la medida de lo posible la información original de la imagen.

Esta diferencia también se refleja en los valores del kernel. Recordemos que los operadores de detección de bordes presentaban cambios de polaridad y una suma total de sus coeficientes igual a cero, ya que su objetivo era responder a las variaciones de intensidad. En cambio, los filtros espaciales tradicionales utilizan coeficientes positivos cuya suma es igual a uno, evitando modificar el nivel promedio de brillo de la imagen mientras realizan el proceso de suavizado.

La principal excepción es el filtro de mediana, el cual no utiliza una suma ponderada de los valores contenidos en el kernel. En su lugar, reemplaza el píxel central por la mediana de los valores presentes en la vecindad, razón por la cual presenta un comportamiento diferente al de los demás filtros espaciales.

En la Figura 91 se resumen las principales diferencias entre los filtros utilizados para la detección de bordes y aquellos empleados para la reducción de ruido, incluyendo como caso especial el filtro de mediana. En los dos primeros casos, el resultado se obtiene mediante una convolución (estrictamente, una correlación) entre la imagen de entrada y una máscara o kernel, el cual se desplaza por toda la imagen, normalmente de izquierda a derecha y de arriba hacia abajo.

Para evitar que la imagen de salida sea más pequeña que la imagen original, es habitual **ampliar sus bordes** antes de realizar el filtrado. Una estrategia muy utilizada consiste en añadir un borde de ceros alrededor de la imagen, aunque también existen otras técnicas de extensión de tamaño de la imagen. De esta manera, al aplicar el kernel se obtiene una imagen de salida con el mismo tamaño que la imagen de entrada.

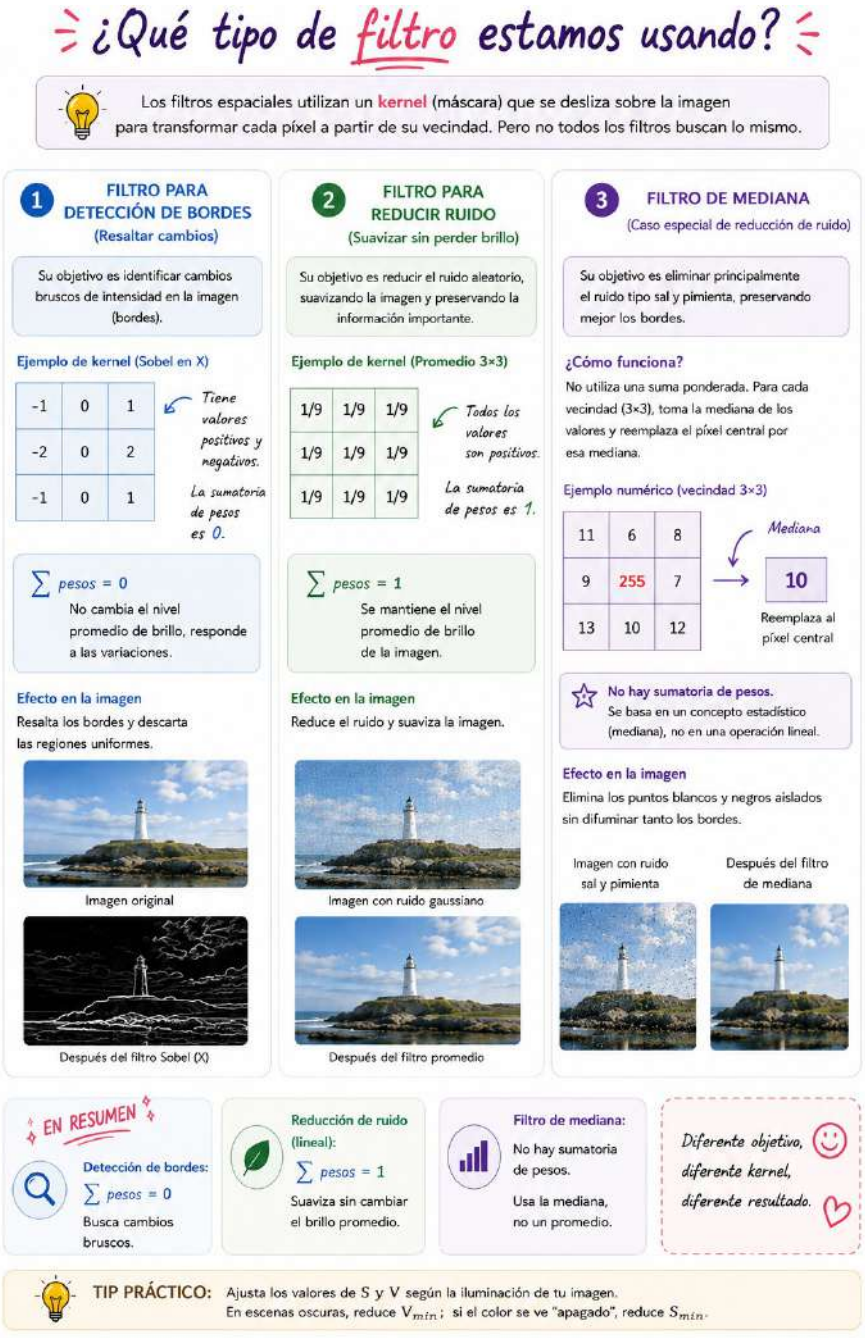


Figura 91. Filtro para detección de bordes vs. filtro para reducción de ruido.

Veamos ahora el proceso de convolución entre el kernel y la imagen.

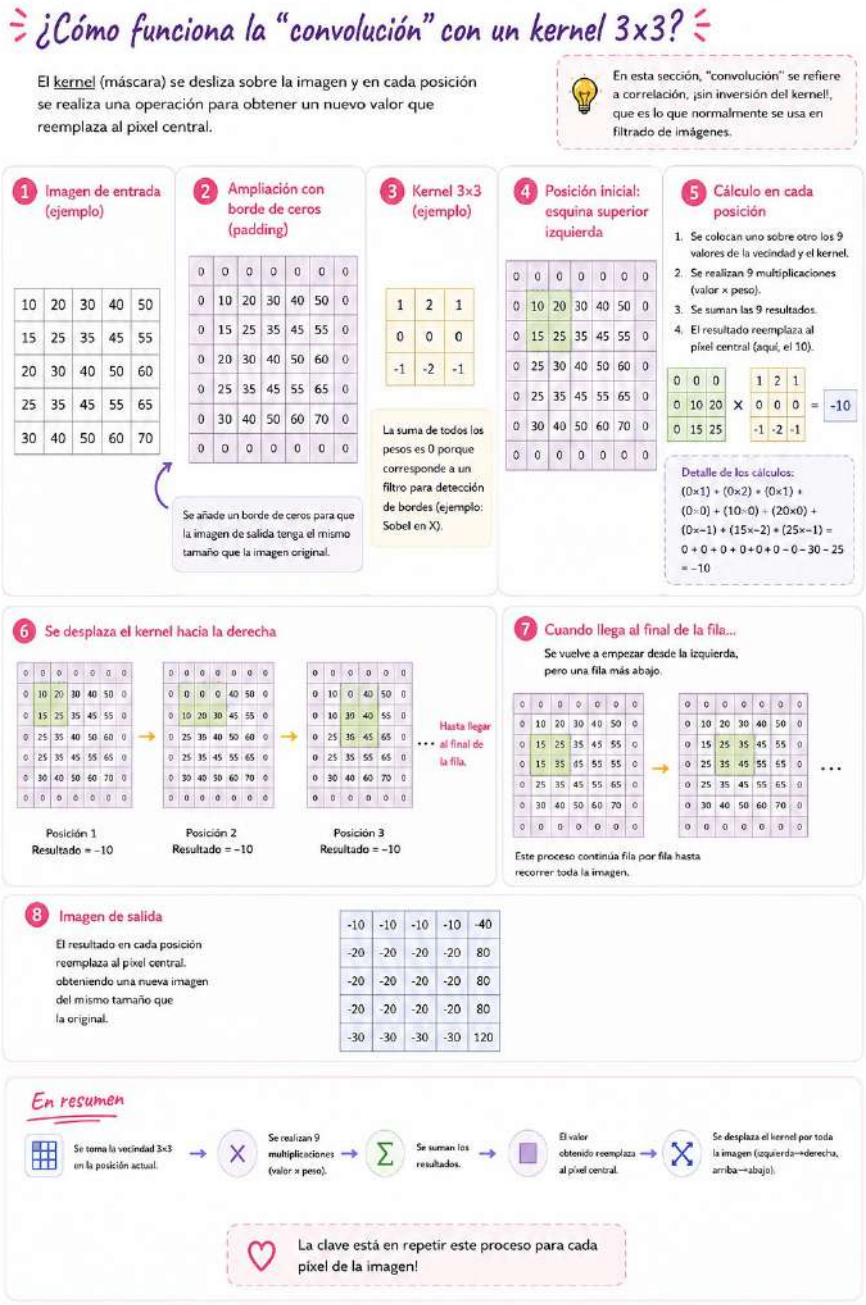


Figura 92. Explicación proceso de convolución.

Y de esta forma obtenemos la imagen de salida, la cual corresponderá a una imagen con bordes resaltados o a una imagen con ruido reducido, dependiendo del tipo de kernel utilizado. Sin embargo, dado que una imagen digital almacena valores enteros de intensidad entre 0 y 255, es posible que durante la convolución algunos resultados queden por fuera de este intervalo.

En estos casos se aplican dos reglas muy sencillas:

- Si el valor calculado para un píxel es menor que 0, se reemplaza por 0.
- Si el valor calculado para un píxel es mayor que 255, se reemplaza por 255.

Este proceso, conocido como saturación o recorte de valores (*clipping*), garantiza que todos los píxeles de la imagen de salida permanezcan dentro del rango permitido (este comportamiento es especialmente común en la detección de bordes, como se observa en la Figura 92).

Por otro lado, en el caso del filtro de mediana ocurre algo diferente. En lugar de utilizar un kernel con coeficientes para realizar multiplicaciones y sumas, se emplea una especie de ventana (o kernel sin pesos) que también se desplaza por toda la imagen. Sin embargo, su propósito no es realizar una convolución, sino encontrar el valor de la mediana de los 9 píxeles contenidos dentro de la ventana de 3×3. Una vez calculada la mediana, este valor reemplaza al píxel central de la vecindad.

Ahora que ya entendimos las diferencias entre los distintos tipos de filtros y el proceso de convolución, estamos listos para conocer los filtros más utilizados en procesamiento digital de imágenes: filtro de promedio, filtro gaussiano, filtro bilateral y filtro de mediana. La explicación de cada uno de ellos, junto con su correspondiente instrucción en Python utilizando OpenCV, se presenta en la Figura 93.

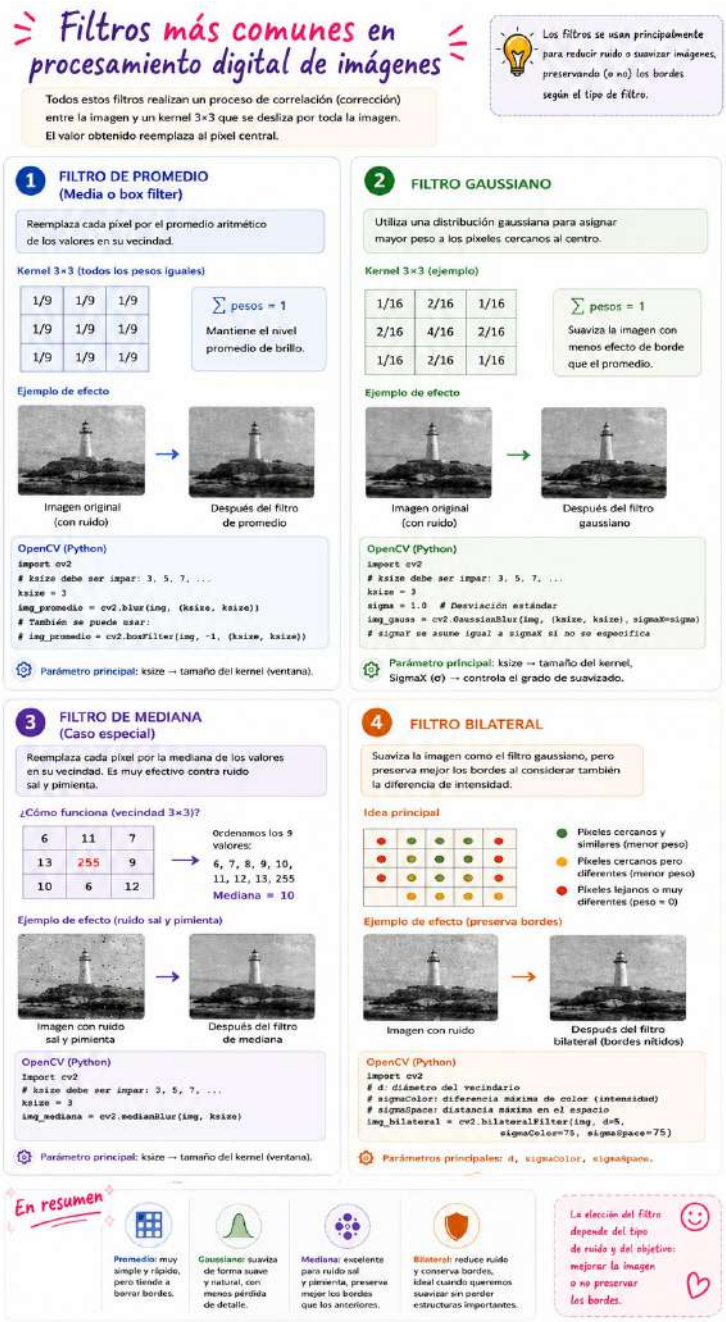


Figura 93. Filtros para reducción de ruido en imágenes: kernel, descripción y código en Python.

Y de esta forma, tenemos que de manera general:

- Si la imagen presenta **ruido sal y pimienta**, el filtro más recomendado es el **filtro de mediana**. Esto se debe a que los valores de sal (255) y pimienta (0) suelen ubicarse en los extremos al ordenar los valores de la vecindad, por lo que normalmente no corresponden a la mediana utilizada para reemplazar el píxel central. En este filtro, el principal parámetro por ajustar es el tamaño del kernel, el cual suele ser de 3 × 3, 5 × 5, 7 × 7, etc. En general, no se recomienda utilizar kernels demasiado grandes, ya que pueden eliminar detalles importantes de la imagen.
- Por otro lado, si la imagen presenta **ruido gaussiano**, el filtro más recomendado es el **filtro bilateral**, ya que considera simultáneamente la proximidad espacial entre los píxeles y la similitud de intensidad (o color). Gracias a ello, es capaz de reducir el ruido preservando mejor los bordes que otros filtros de suavizado.

  En este caso deben ajustarse tres parámetros principales:
  - ❖ d: diámetro de la vecindad utilizada durante el filtrado.
  - ❖ sigmaColor: controla qué tanta diferencia de intensidad o color pueden tener dos píxeles para ser considerados similares.
  - ❖ sigmaSpace: controla la influencia de la distancia espacial entre los píxeles de la vecindad.
- Finalmente, los filtros de promedio y gaussiano también permiten reducir ruido, especialmente cuando las perturbaciones son suaves o aproximadamente uniformes. Sin embargo, ambos tienden a difuminar los bordes de la imagen, por lo que es recomendable utilizar kernels pequeños. Si aun así observamos una pérdida importante de detalle, una mejor alternativa es utilizar el filtro bilateral.

De esta forma, es importante identificar el tipo de ruido presente en la imagen, para así seleccionar el tipo de filtro que mejor reduce el ruido y configurar adecuadamente el o los parámetros del filtro.

4.6. ¿Qué aprendimos en este capítulo?

Como te has podido dar cuenta, el mundo del procesamiento digital de imágenes tiene sus propias particularidades respecto al procesamiento de

señales de audio, razón por la cual necesitábamos una nueva sección para conocer, entender y mejorar este tipo de señales.

Creo que una de las ideas más importantes que aprendimos es que una imagen no solo puede visualizarse, sino también **analizarse, transformarse y utilizarse para extraer información**. A lo largo de este capítulo vimos cómo mejorar su contraste mediante histogramas y ecualización, cómo identificar regiones de interés utilizando detección de bordes y segmentación, y cómo el ruido puede afectar significativamente la calidad de una imagen y, como consecuencia, el desempeño de los algoritmos que trabajan sobre ella.

También comprendimos que no existe un único filtro adecuado para todas las situaciones. La selección del filtro depende del tipo de ruido presente en la imagen y del objetivo que se persiga, ya sea suavizar la imagen, preservar los bordes o eliminar valores atípicos. En ingeniería, como ocurre con frecuencia, **la mejor herramienta siempre dependerá del problema que se desea resolver**.

En el siguiente capítulo dejaremos un poco la teoría para pasar a la práctica. De forma similar a lo que hicimos en la Sección I, abordaremos un caso de estudio que nos permitirá aplicar los conceptos aprendidos a un problema real de segmentación de imágenes, evaluando distintas estrategias para determinar cuál ofrece los mejores resultados.

Al igual que ocurrió con las señales de audio, comprender primero los fundamentos hará que las aplicaciones prácticas del siguiente capítulo tengan mucho más sentido. **Y quizá descubras que una buena solución de ingeniería no comienza cuando ejecutamos un algoritmo, sino mucho antes: cuando entendemos el problema y aprendemos a mirar la información con los ojos adecuados.**

CAPÍTULO V

SEGMENTACIÓN DE IMÁGENES CON TÉCNICAS CLÁSICAS DE PDI

Hasta este momento hemos recorrido los principales fundamentos del procesamiento digital de imágenes. Como habrás notado, este libro no pretende sustituir los textos clásicos de Procesamiento Digital de Imágenes. En ellos encontrarás un tratamiento mucho más profundo de los temas estudiados en el capítulo anterior, así como numerosas técnicas que aquí no hemos abordado, entre ellas las operaciones morfológicas avanzadas, la transformada de Hough, la detección de esquinas, la restauración de imágenes y muchas otras herramientas que forman parte de esta disciplina.

Sin embargo, el propósito de este libro es diferente. Más que presentar un catálogo de algoritmos, busca mostrar cómo un ingeniero construye una solución a partir de un problema real. En la práctica, desarrollar un sistema de visión por computador implica mucho más que aplicar unas cuantas funciones de una librería como OpenCV. Significa comprender el problema, analizar las características de las imágenes disponibles, decidir cómo prepararlas y entender que cada una de estas decisiones influirá en el desempeño de las etapas posteriores.

Con esta idea en mente, durante este capítulo y el siguiente trabajaremos sobre un caso de estudio inspirado en un proyecto de investigación desarrollado hace algunos años, cuyo objetivo era la detección automática de enfermedades en hojas de café. En aquel momento nos enfrentamos a un reto que, a primera vista, parecía sencillo: entrenar un modelo capaz de identificar diferentes

enfermedades a partir de imágenes. Sin embargo, pronto descubrimos que el verdadero desafío comenzaba mucho antes del entrenamiento del modelo.

Las imágenes habían sido capturadas en condiciones reales de cultivo. Por esta razón, presentaban diferencias de iluminación, fondos muy variados, hojas con diferentes tamaños, orientaciones y niveles de afectación, además de ruido e información que no aportaba al problema. Antes de pensar en un modelo de clasificación o en Inteligencia Artificial, fue necesario desarrollar una estrategia que permitiera preparar el conjunto de datos y obtener imágenes más homogéneas y representativas.

Precisamente ese será el objetivo de este capítulo. Aplicaremos los conceptos estudiados hasta el momento para construir un procedimiento de segmentación basado en técnicas clásicas de procesamiento digital de imágenes. A lo largo del capítulo desarrollaremos **tres soluciones**, cada una más elaborada que la anterior, analizaremos sus fortalezas y limitaciones, y evaluaremos cuál de ellas ofrece los mejores resultados para este caso de estudio.

Y, lo más interesante, comprobaremos que incluso una solución que parece funcionar correctamente siempre puede mejorarse cuando se analiza con mayor detalle.

5.1. Conociendo nuestro conjunto de imágenes (RoCole)

Antes de desarrollar cualquier sistema basado en inteligencia artificial, el primer paso consiste en conocer las características del conjunto de datos disponible. Una inspección visual de las imágenes permite identificar diferentes situaciones que pueden afectar el desempeño del modelo y que deberán ser consideradas durante la etapa de preparación del *dataset*.

Para nuestro caso de estudio utilizaremos imágenes provenientes del **dataset RoCoLe (Robusta Coffee Leaf Images Dataset)**, publicado en 2019 por investigadores de la **Universidad de Santiago de Chile**. Este conjunto de datos fue desarrollado para apoyar investigaciones relacionadas con la detección automática de enfermedades en hojas de café, especialmente roya

y otras afecciones foliares, utilizando imágenes capturadas directamente en condiciones reales de cultivo.

Una de las principales fortalezas de este *dataset* es que no fue construido en un entorno controlado de laboratorio, sino en campo, bajo condiciones reales de adquisición. Como consecuencia, las imágenes presentan iluminación natural, diferentes fondos, hojas completas y parciales, distintas orientaciones, variaciones de tamaño y diversos niveles de afectación. Aunque esta variabilidad representa un reto para cualquier sistema de visión por computador, también convierte a RoCoLe en un excelente caso de estudio para integrar los conceptos desarrollados en el **Capítulo 4** de este libro.

La Figura 94 presenta algunos ejemplos de imágenes extraídas del *dataset* RoCoLe. Este conjunto de datos está conformado por **1560 imágenes a color** correspondientes a hojas de café clasificadas en las siguientes categorías: **saludable, ácaro rojo, roya nivel 1, roya nivel 2, roya nivel 3 y roya nivel 4**.

Además, las imágenes presentan una gran variabilidad en sus condiciones de captura, incluyendo **fondos diferentes, iluminación variable, cambios de escala, distintos tamaños, orientaciones diversas y la presencia de ramas, suelo y vegetación**. En consecuencia, el reto no consiste únicamente en reconocer una enfermedad, sino en hacerlo de forma confiable aun cuando las imágenes hayan sido adquiridas bajo condiciones muy diferentes entre sí.

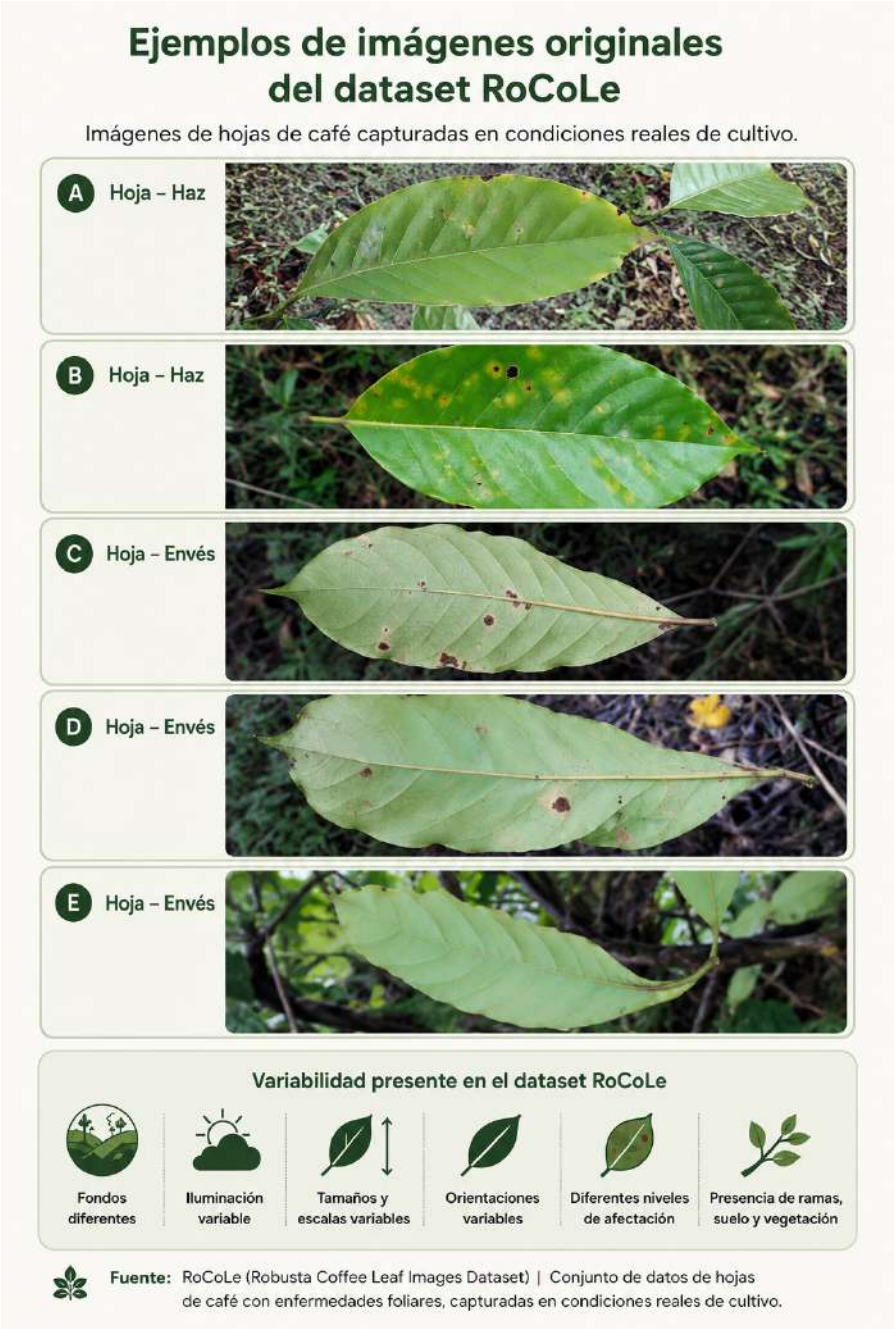


Figura 94. Ejemplo ilustrativo de imágenes pertenecientes al dataset RoCole.

Después de realizar esta primera inspección del conjunto de imágenes, surge una pregunta fundamental:

¿Están realmente listas estas imágenes para ser utilizadas por un modelo de inteligencia artificial?

La respuesta, como ocurre en la mayoría de los proyectos reales de visión por computador, es no del todo. Aunque las imágenes contienen la información requerida para identificar las enfermedades, también presentan múltiples variaciones que pueden dificultar el proceso de aprendizaje del modelo. Antes del entrenamiento es necesario analizar cuidadosamente cuáles de esas variaciones aportan información útil y cuáles, por el contrario, podrían desviar la atención del algoritmo hacia patrones irrelevantes.

Por ejemplo, **imagina que todas las imágenes correspondientes a hojas con roya de nivel 1 hubieran sido capturadas sobre el suelo y con la hoja orientada siempre en la misma dirección**. En ese caso, el modelo podría aprender a asociar el **fondo** o la **orientación de la hoja** con la clase *roya nivel 1*, en lugar de aprender las características propias de la enfermedad. Como consecuencia, cuando el modelo recibiera una nueva imagen con un fondo diferente o con otra orientación, su desempeño podría disminuir considerablemente para la identificación de este nivel de roya.

Por esta razón, la preparación del conjunto de datos tiene como objetivo reducir aquellas variaciones que no aportan información al problema de clasificación, permitiendo que el modelo concentre su atención en las características realmente importantes.

Entonces, nuestra primera tarea consistirá en eliminar automáticamente el fondo de las imágenes. El objetivo es muy sencillo: hacer que el modelo “mire” principalmente la hoja y no el entorno donde fue tomada la fotografía. Para ello reemplazaremos el fondo por un color uniforme (negro o blanco), disminuyendo la influencia de elementos como el suelo, las ramas o la vegetación durante el proceso de aprendizaje.

Pero aparece una pregunta que supongo en este momento te estás haciendo:

¿Cómo le enseñamos a un computador cuáles píxeles pertenecen a la hoja y cuáles pertenecen al fondo?

La buena noticia es que **ya conocemos una posible estrategia**. En el **Capítulo 4** estudiamos la segmentación de imágenes utilizando el modelo **HSV**, donde aprendimos que es posible aislar regiones de una imagen seleccionando rangos específicos de color. En nuestro caso, dado que las hojas presentan predominantemente tonalidades verdes, utilizaremos esta técnica como punto de partida para separar la hoja del fondo.

Como ocurre en la mayoría de los problemas de ingeniería, nuestro objetivo no será obtener una segmentación perfecta desde el primer intento, sino desarrollar un procedimiento que conserve la mayor parte de la hoja y elimine la mayor cantidad posible de información del fondo.

5.2. Reto: Separar automáticamente la hoja principal del fondo

A primera vista, separar automáticamente la hoja del fondo puede parecer una tarea sencilla para una persona, ya que nuestro cerebro identifica de forma casi inmediata cuál es la hoja principal y cuál es el fondo. Sin embargo, para un computador esta información no existe de manera explícita: una imagen no es más que una matriz de píxeles, donde cada píxel almacena un valor de color.

Por esta razón, debemos diseñar un procedimiento que permita diferenciar automáticamente los píxeles pertenecientes a la hoja de aquellos correspondientes al fondo. Una de las estrategias más utilizadas consiste en aprovechar las diferencias de color entre ambos, utilizando para ello el modelo **HSV**, estudiado en el capítulo anterior.

Nuestro objetivo será generar una **máscara binaria**, donde los píxeles pertenecientes a la hoja se representen en blanco y el fondo en negro. Posteriormente, dicha máscara se utilizará para conservar únicamente la región de interés y eliminar el resto de la imagen.

Ejemplo 27. Segmentación de la hoja mediante una máscara binaria en el modelo HSV

La idea consiste en definir un rango de tonalidades verdes y generar una máscara binaria, donde los píxeles que cumplen las condiciones establecidas se representan en blanco y los demás en negro. Posteriormente, aplicaremos esta máscara sobre la imagen original para conservar únicamente las regiones seleccionadas.

Vamos a aplicar esta estrategia sobre cinco imágenes obtenidas del *dataset* RoCoLe, utilizando el siguiente código en Python, muy similar al presentado en el **Ejemplo 25**.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

nombres = ["C10P11E2.jpg", "C10P10H2.jpg", "C10P10H1.jpg",
           "C10P10E2.jpg", "C10P10E1.jpg"] # Imágenes de RoCoLe

lim_inf = np.array([25, 35, 25]) # Límite inferior HSV para tonalidades verdes
lim_sup = np.array([95, 255, 255]) # Límite superior HSV para tonalidades verdes

plt.figure(figsize=(15, 7))

for i, nombre in enumerate(nombres):
    imagen_bgr = cv2.imread(nombre) # Leer imagen
    imagen_rgb = cv2.cvtColor(imagen_bgr, cv2.COLOR_BGR2RGB) # Convertir a RGB
    imagen_hsv = cv2.cvtColor(imagen_bgr, cv2.COLOR_BGR2HSV) # Convertir a HSV

    mascara = cv2.inRange(imagen_hsv, lim_inf, lim_sup) # Crear máscara binaria
    segmentada = cv2.bitwise_and(imagen_rgb, imagen_rgb, mask=mascara) # Aplicar máscara

    plt.subplot(2, 5, i + 1); plt.imshow(imagen_rgb)
    plt.title(f"Original {i + 1}"); plt.axis("off")
    plt.subplot(2, 5, i + 6); plt.imshow(segmentada)
    plt.title(f"Segmentada {i + 1}"); plt.axis("off")
plt.tight_layout(); plt.show()
```

Figura 95. Código en Python del Ejemplo 27. Segmentación de la imagen mediante una máscara de color.

El código comienza definiendo los límites inferior ([25, 35, 25]) y superior ([95, 255, 255]) del modelo HSV para segmentar tonalidades verdes. Recordemos que el canal H (Hue) es el encargado de definir el color, mientras que los canales S y V permiten restringir la saturación y el brillo. Con estos límites, la función cv2.inRange() genera una máscara binaria donde los píxeles dentro del rango toman el valor 255 (blanco) y los demás 0 (negro). Finalmente, esta máscara se aplica sobre la imagen original para conservar únicamente las regiones segmentadas. El resultado se presenta en la Figura 95.

Como era de esperarse, la segmentación basada únicamente en el modelo HSV no logra separar completamente la hoja del fondo. Aunque la hoja principal fue correctamente identificada en todos los casos, también se segmentaron otras regiones de la imagen con tonalidades verdes similares, como hojas y vegetación presentes en el fondo.

Sin embargo, observamos que los mejores resultados se obtuvieron en las imágenes **Original 2, Original 3 y Original 4**. En estos casos, el fondo está constituido principalmente por tierra, hojas secas y vegetación con tonalidades diferentes al verde de la hoja, lo que facilita la segmentación por color. En cambio, en las imágenes **Original 1** y **Original 5** aparecen otras hojas verdes muy próximas a la hoja de interés, las cuales también son seleccionadas por la máscara al compartir un rango de color similar.

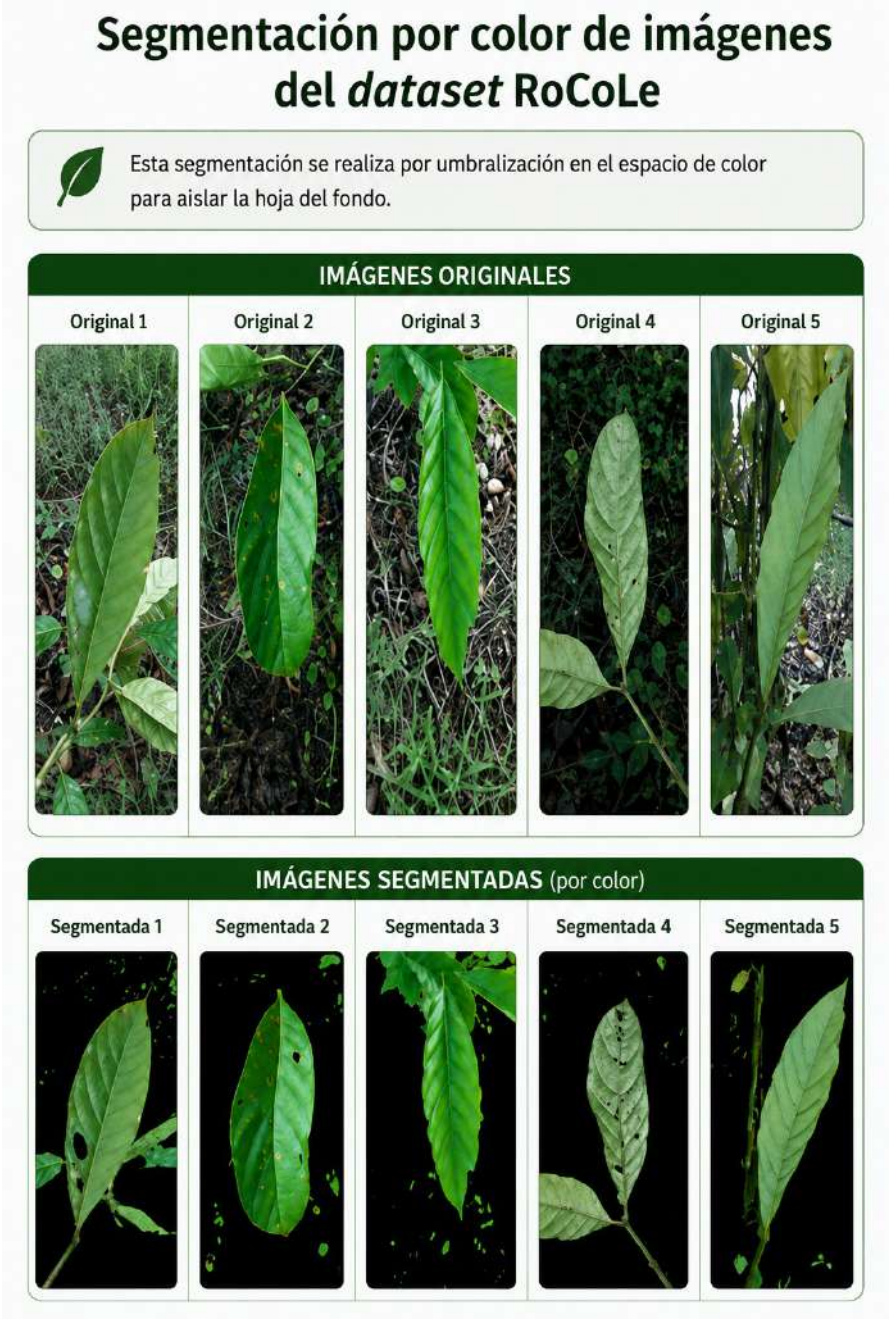


Figura 96. Segmentación por color de imágenes del *dataset* RoCoLe. Primera solución utilizando máscara de color.

En consecuencia, esta primera aproximación constituye un buen punto de partida, pero aún requiere una etapa adicional que permita conservar únicamente la hoja principal.

Entonces surge una nueva pregunta:

¿Cómo podemos indicarle al computador cuál de todas las regiones verdes corresponde realmente a la hoja que queremos analizar?

Ejemplo 28. Segmentación de la hoja con contornos y máscara binaria en el modelo HSV

En el ejemplo anterior comprobamos que la segmentación basada únicamente en el modelo HSV permite identificar correctamente las tonalidades verdes de la imagen. Sin embargo, también observamos que esta estrategia conserva numerosas regiones pertenecientes al fondo, como hojas, ramas y vegetación, ya que todas ellas presentan colores similares a los de la hoja de interés.

Una primera idea para mejorar este resultado consiste en aprovechar una característica adicional de las imágenes del *dataset* RoCoLe. Al observar cuidadosamente las fotografías, notamos que la hoja que deseamos analizar suele corresponder a la región verde de mayor tamaño dentro de la escena. A partir de esta hipótesis, ya no conservaremos todas las regiones segmentadas, sino únicamente aquella cuyo contorno presente el área más grande.

En otras palabras, la segmentación basada en color continuará siendo el primer paso del algoritmo, pero posteriormente incorporaremos un criterio basado en el tamaño de las regiones para eliminar gran parte de la vegetación del fondo y conservar únicamente la hoja principal.

Aplicaremos esta estrategia sobre las mismas cinco imágenes utilizadas en el ejemplo anterior para comparar los resultados obtenidos.

```python
import numpy as np
import matplotlib.pyplot as plt

nombres = ["C10P11E2.jpg", "C10P10H2.jpg", "C10P10H1.jpg",
           "C10P10E2.jpg", "C10P10E1.jpg"] # Imágenes de RoCoLe

lim_inf = np.array([25, 35, 25]) # Límite inferior HSV
lim_sup = np.array([95, 255, 255]) # Límite superior HSV

plt.figure(figsize=(15, 7))

for i, nombre in enumerate(nombres):
    imagen_bgr = cv2.imread(nombre) # Leer imagen
    imagen_rgb = cv2.cvtColor(imagen_bgr, cv2.COLOR_BGR2RGB) # Convertir a RGB
    imagen_hsv = cv2.cvtColor(imagen_bgr, cv2.COLOR_BGR2HSV) # Convertir a HSV

    mascara = cv2.inRange(imagen_hsv, lim_inf, lim_sup) # Máscara inicial
    contornos, _ = cv2.findContours(mascara, cv2.RETR_EXTERNAL,
                                    cv2.CHAIN_APPROX_SIMPLE) # Encontrar regiones
    mascara_roi = np.zeros_like(mascara) # Crear máscara negra

    if contornos:
        mayor = max(contornos, key=cv2.contourArea) # Contorno de mayor área
        cv2.drawContours(mascara_roi, [mayor], -1, 255, -1) # Rellenar región
    segmentada = cv2.bitwise_and(imagen_rgb, imagen_rgb,
                                 mask=mascara_roi) # Conservar hoja principal

    plt.subplot(2, 5, i + 1); plt.imshow(imagen_rgb)
    plt.title(f"Original {i + 1}"); plt.axis("off")
    plt.subplot(2, 5, i + 6); plt.imshow(segmentada)
    plt.title(f"Región de interés {i + 1}"); plt.axis("off")
plt.tight_layout(); plt.show()
```

Figura 97. Código en Python del Ejemplo 28. Segmentación de la imagen mediante una máscara de color y análisis de contornos.

Observa que la mayor parte del código es idéntica al del ejemplo anterior. La única diferencia consiste en que ahora no utilizamos directamente la máscara binaria obtenida mediante HSV. En su lugar, identificamos todos los contornos presentes en la máscara, seleccionamos el de mayor área y construimos una nueva máscara utilizando únicamente esa región. De esta forma, el algoritmo conserva la hoja principal y elimina una parte importante de la vegetación del fondo.

El resultado obtenido se presenta en la Figura 98.

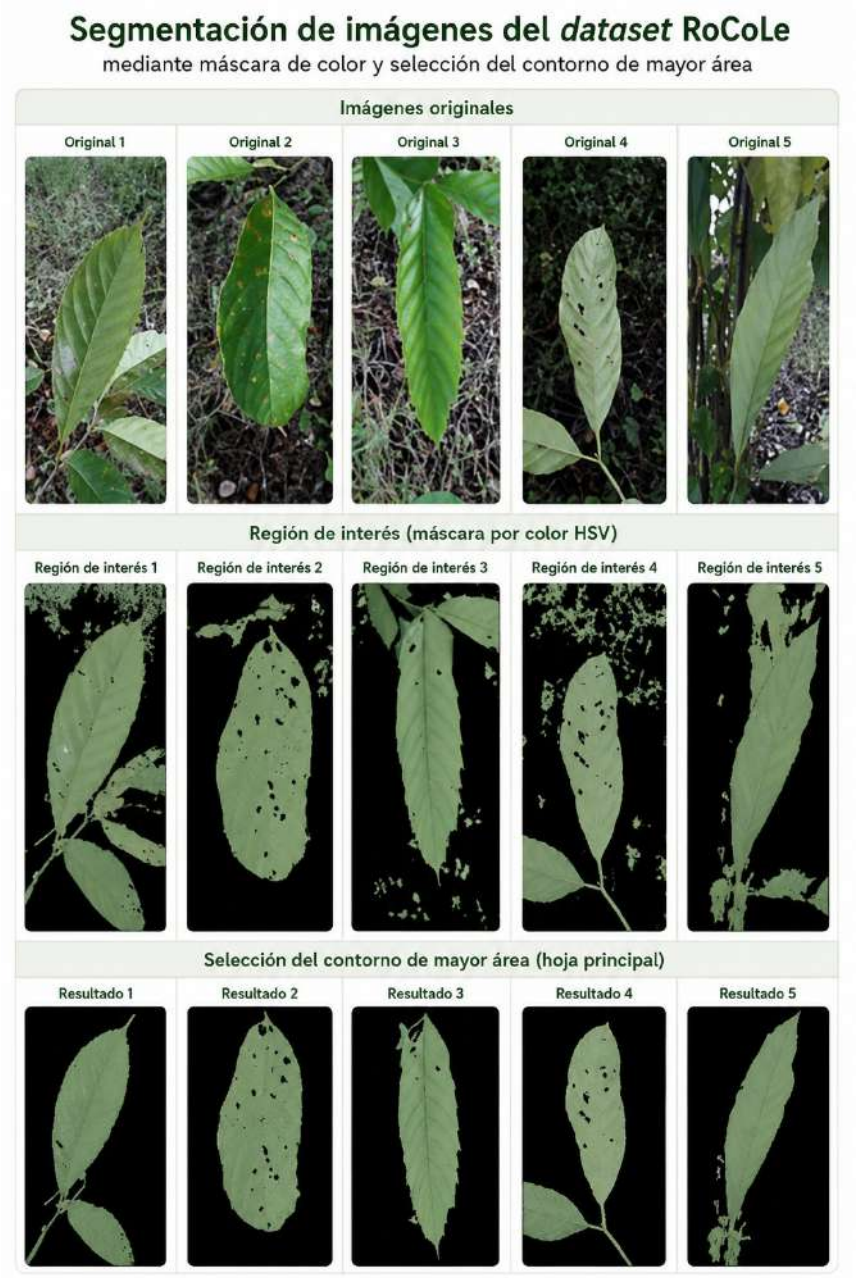


Figura 98. Segmentación de imágenes del *dataset* RoCoLe mediante máscara de color y selección del contorno de mayor área. Segunda solución.

Aunque todavía no obtenemos una segmentación perfecta, sí es posible apreciar una mejora respecto a la primera solución. En términos generales, la incorporación del análisis de contornos permite:

- Reducir la cantidad de vegetación del fondo conservada en la imagen segmentada.
- Disminuir la aparición de falsos negros en el interior de algunas hojas, preservando mejor algunas regiones correspondientes a manchas blancas o amarillas presentes en la hoja original.
- Obtener una hoja principal más continua, especialmente en las imágenes Región de Interés 2, Región de Interés 3 y Región de Interés 4.
- Mantener una mayor parte de la información útil de la hoja sin incrementar significativamente la complejidad del algoritmo.
-

Sin embargo, observamos que todavía permanecen regiones del fondo conectadas a la hoja principal en algunos casos. Esto nos lleva a plantear una nueva pregunta:

¿es posible incorporar información adicional para seleccionar de manera más precisa la región de interés?

Ejemplo 29. Segmentación de la hoja mediante contornos, operaciones morfológicas y análisis de centroides

En el ejemplo anterior utilizamos el área del contorno como criterio para seleccionar la hoja principal. Aunque esta estrategia permitió reducir una parte importante de la vegetación presente en el fondo, todavía se observaron algunos casos donde la región seleccionada incluía información no deseada o presentaba pequeños huecos en el interior de la hoja.

Una forma de refinar esta solución consiste en incorporar información adicional sobre las regiones segmentadas. Para ello, antes de seleccionar la región de interés, aplicaremos dos operaciones morfológicas sobre la máscara binaria: una **apertura**, con el propósito de eliminar pequeñas regiones aisladas, y un **cierre**, para reducir pequeños huecos y conectar regiones próximas pertenecientes a la misma hoja.

Posteriormente, identificaremos todas las regiones conectadas presentes en la máscara y calcularemos, para cada una de ellas, diferentes características, entre ellas su **área**, la **posición de su centroide** y la proporción de píxeles ubicados en la zona central de la imagen. Finalmente, se conservará la región que obtenga la mejor puntuación de acuerdo con estos criterios.

Aplicaremos esta nueva estrategia sobre las mismas cinco imágenes utilizadas en los ejemplos anteriores para comparar los resultados obtenidos.

```
for i, nombre in enumerate(nombres):
    imagen_bgr = cv2.imread(nombre) # Leer imagen
    imagen_rgb = cv2.cvtColor(imagen_bgr, cv2.COLOR_BGR2RGB) # Convertir a RGB
    imagen_hsv = cv2.cvtColor(imagen_bgr, cv2.COLOR_BGR2HSV) # Convertir a HSV

    mascara = cv2.inRange(imagen_hsv, lim_inf, lim_sup) # Máscara inicial
    kernel = np.ones((5, 5), np.uint8)
    mascara = cv2.morphologyEx(mascara, cv2.MORPH_OPEN,
                               kernel, iterations=1) # Eliminar regiones pequeñas
    mascara = cv2.morphologyEx(mascara, cv2.MORPH_CLOSE,
                               kernel, iterations=2) # Cerrar huecos pequeños
    num, etiquetas, stats, centroides = cv2.connectedComponentsWithStats(
        mascara, connectivity=8) # Regiones conectadas

    alto, ancho = mascara.shape
    centro_img = np.array([ancho / 2, alto / 2])

    # Región central de referencia
    x1, x2 = int(0.25 * ancho), int(0.75 * ancho)
    y1, y2 = int(0.15 * alto), int(0.90 * alto)

    mejor_etiqueta = 0; mejor_puntaje = 0
    for j in range(1, num):
        x, y, w, h, area = stats[j]
        centro = centroides[j]

        region = etiquetas == j
        pixeles_centrales = np.sum(region[y1:y2, x1:x2])
        proporcion_central = pixeles_centrales / max(area, 1)

        distancia = np.linalg.norm(centro - centro_img)
        cercania = 1 / (1 + distancia)

        toca_borde = x == 0 or y == 0 or x + w >= ancho or y + h >= alto
        penalizacion = 0.40 if toca_borde else 1.0

        puntaje = area * (1 + 3 * proporcion_central) * cercania * penalizacion

        if puntaje > mejor_puntaje:
            mejor_puntaje = puntaje
            mejor_etiqueta = j
    mascara_roi = np.zeros_like(mascara)
    mascara_roi[etiquetas == mejor_etiqueta] = 255 # Conservar región seleccionada

    # Rellenar huecos internos de la hoja
    contornos, _ = cv2.findContours(mascara_roi, cv2.RETR_EXTERNAL,
                                    cv2.CHAIN_APPROX_SIMPLE)
    mascara_final = np.zeros_like(mascara_roi)

    if contornos:
        mayor = max(contornos, key=cv2.contourArea)
        cv2.drawContours(mascara_final, [mayor], -1, 255, -1)

    segmentada = cv2.bitwise_and(imagen_rgb, imagen_rgb,
                                 mask=mascara_final) # Aplicar máscara

    plt.subplot(2, 5, i + 1); plt.imshow(imagen_rgb)
    plt.title(f"Original {i + 1}"); plt.axis("off")

    plt.subplot(2, 5, i + 6); plt.imshow(segmentada)
    plt.title(f"Región de interés {i + 1}"); plt.axis("off")

plt.tight_layout(); plt.show()
```

Figura 99. Código en Python del Ejemplo 29. Segmentación de la hoja mediante operaciones morfológicas, análisis de contornos y centroides.

Observa que este algoritmo mantiene la estrategia de segmentación basada en HSV y la selección de la región principal mediante contornos. La diferencia respecto al ejemplo anterior consiste en que ahora se incorporan **operaciones morfológicas** para mejorar la calidad de la máscara binaria y un **criterio de selección más robusto**, basado no solo en el área de cada región, sino también en su ubicación dentro de la imagen.

El resultado obtenido se presenta en la Figura 100.

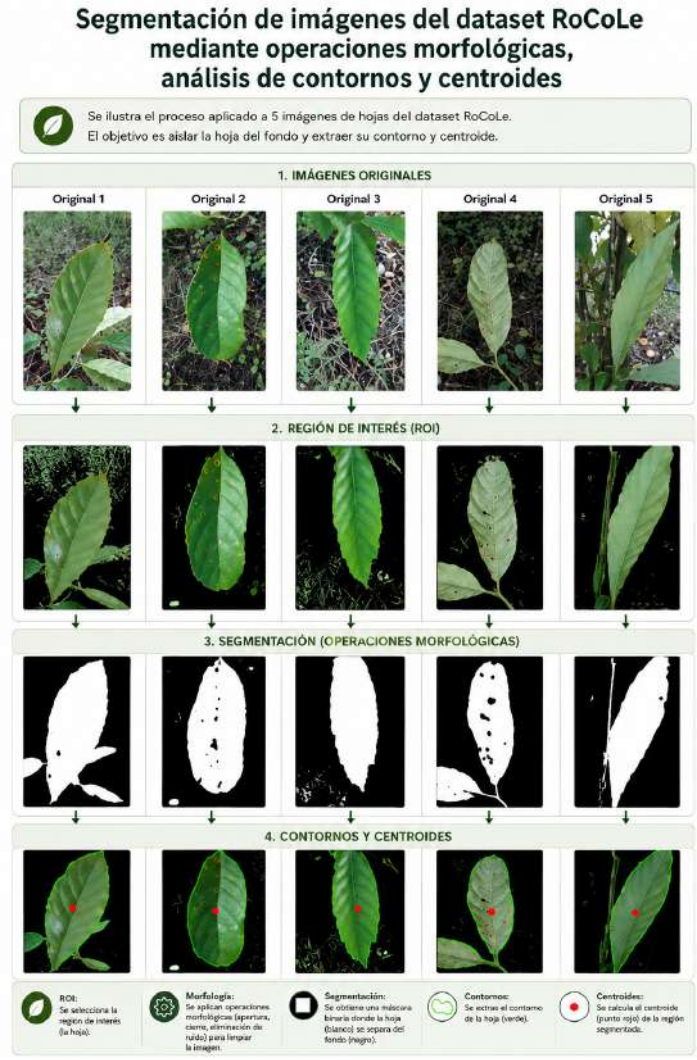


Figura 100. Segmentación de imágenes del dataset RoCoLe mediante operaciones morfológicas, análisis de contornos y centroides. Tercera solución.

Si bien las diferencias respecto a la segunda solución son, en términos generales, sutiles, en la **Región de Interés 4** sí se aprecia una mejora importante. En esta imagen se reducen varios de los falsos negros presentes en el interior de la hoja, conservando mejor las manchas originalmente asociadas con la roya. Además, se cumple parcialmente el objetivo de convertir el entorno en un fondo homogéneo, ya que buena parte de ramas, hojas secas y suelo pasan a representarse en color negro. Aunque la eliminación del fondo aún no es perfecta, la hoja adquiere un mayor protagonismo y queda mejor diferenciada de la escena.

5.3. Análisis comparativo de las soluciones propuestas en el Reto

Después de desarrollar las tres estrategias de segmentación, resulta conveniente comparar sus resultados para identificar las ventajas y limitaciones de cada una de ellas. Recordemos que el objetivo del reto no consistía en obtener una segmentación perfecta, sino en construir, paso a paso, una solución capaz de reducir progresivamente la influencia del fondo y resaltar la hoja de interés.

La Figura 101 resume la evolución de las tres soluciones desarrolladas durante este reto.

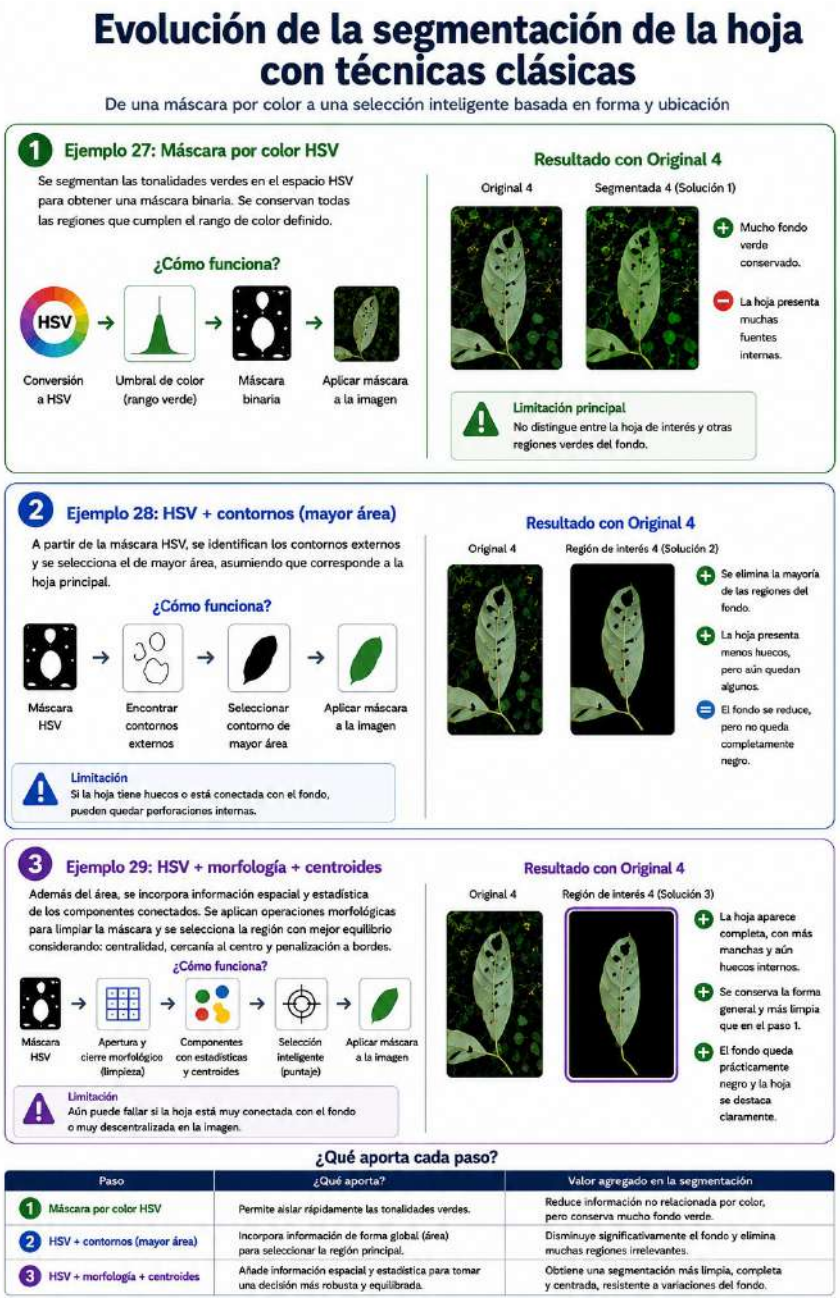


Figura 101. Comparación de las tres estrategias propuestas para la segmentación de la hoja.

A partir de esta comparación podemos observar que cada nueva solución incorpora una etapa adicional de procesamiento y consigue mejorar parcialmente los resultados obtenidos en la etapa anterior. La primera estrategia, basada únicamente en el modelo HSV, permite identificar las tonalidades verdes, pero conserva una gran cantidad de vegetación presente en el fondo y genera falsos negros en el interior de algunas hojas.

La segunda solución incorpora el análisis de contornos para conservar únicamente la región de mayor área, logrando reducir una parte importante del fondo y preservar mejor la estructura de la hoja.

Finalmente, la tercera solución añade operaciones morfológicas y un criterio de selección basado en centroides, refinando la máscara obtenida y mejorando algunos casos particulares, como la **Región de Interés 4**, donde se conservan de mejor manera las manchas asociadas con la roya.

Sin embargo, es importante destacar que **ninguna de las tres soluciones consigue eliminar completamente el fondo en todas las imágenes**. Sin embargo, el objetivo del reto se cumple parcialmente, ya que fue posible transformar gran parte de ramas, hojas secas y suelo en un fondo negro homogéneo, permitiendo que la hoja de interés adquiera un mayor protagonismo dentro de la escena. Más importante aún, este ejercicio demuestra que el desarrollo de una solución de ingeniería rara vez finaliza en el primer intento; por el contrario, es el resultado de un proceso continuo de observación, análisis y refinamiento.

En ingeniería, una solución útil no siempre es una solución perfecta. Con frecuencia, el mayor avance consiste en comprender las limitaciones de una primera aproximación y utilizarlas como punto de partida para construir una mejor.

CAPÍTULO VI

SEGMENTACIÓN AUTOMÁTICA CON INTELIGENCIA ARTIFICIAL

En el capítulo anterior abordamos uno de los problemas más comunes en visión por computador: separar automáticamente un objeto de interés del resto de la escena. Para ello desarrollamos diferentes estrategias basadas en técnicas clásicas de procesamiento digital de imágenes, comenzando con una simple segmentación por color en el modelo HSV y refinando progresivamente los resultados mediante contornos, operaciones morfológicas y análisis de regiones.

Como pudimos comprobar, cada nueva estrategia permitió mejorar parcialmente la calidad de la segmentación. Sin embargo, ninguna de ellas consiguió resolver correctamente todos los casos presentes en el *dataset* RoCoLe. La gran variabilidad en las condiciones de captura, la presencia de hojas superpuestas, ramas, diferentes niveles de iluminación y fondos complejos dificultan la construcción de un conjunto de reglas capaz de funcionar adecuadamente para todas las imágenes.

Entonces surge una pregunta natural:

¿Es posible que un computador aprenda automáticamente a separar la hoja del fondo sin que el ingeniero tenga que diseñar explícitamente todas las reglas de segmentación?

La respuesta es sí, y precisamente ese es uno de los problemas para los cuales el aprendizaje profundo (*Deep Learning*) ha demostrado excelentes resultados. En lugar de programar manualmente un algoritmo de segmentación, construiremos un modelo capaz de aprender esta tarea a partir de ejemplos previamente segmentados por un experto.

En este capítulo utilizaremos una de las arquitecturas más importantes para segmentación semántica de imágenes: **U-Net**. A diferencia de las redes neuronales utilizadas para clasificación, cuyo objetivo consiste en asignar una única etiqueta a toda la imagen, una U-Net realiza una predicción para cada uno de sus píxeles, permitiendo identificar con precisión las regiones que pertenecen al objeto de interés.

Con fines didácticos, trabajaremos inicialmente con un conjunto de entrenamiento muy reducido, compuesto únicamente por cinco imágenes del *dataset* RoCoLe y sus correspondientes máscaras segmentadas manualmente. Aunque este conjunto de datos es insuficiente para desarrollar un sistema robusto, resulta ideal para comprender el flujo completo de entrenamiento de una red de segmentación y analizar cómo el tamaño del conjunto de entrenamiento influye en la capacidad de generalización del modelo.

6.1. ¿Por qué utilizar IA para segmentar imágenes?

La principal dificultad que identificamos en el Capítulo 5 radica en la enorme variabilidad que presentan las imágenes del dataset RoCole. Aunque todas pertenecen a un mismo conjunto de imágenes, existen diferencias en la iluminación, el tamaño de las hojas, su orientación, la presencia de ramas, vegetación, hojas superpuestas y fondos con tonalidades similares al objeto de interés. En estas condiciones, diseñar manualmente un conjunto de reglas capaz de funcionar correctamente para todos los casos puede convertirse en una tarea extremadamente compleja.

Entonces surge una pregunta diferente a las que hemos planteado hasta ahora:

¿Y si, en lugar de programar las reglas de segmentación, permitimos que el computador las aprenda automáticamente a partir de ejemplos?

Esta pregunta representa uno de los cambios más importantes introducidos por la inteligencia artificial moderna. Mientras los métodos clásicos dependen del conocimiento del ingeniero para definir umbrales, filtros o criterios de decisión, los modelos de aprendizaje profundo construyen esas reglas durante el entrenamiento, analizando un conjunto de imágenes para las cuales ya se conoce la respuesta correcta.

En otras palabras, el objetivo deja de ser escribir un algoritmo que separe la hoja del fondo. Ahora el objetivo consiste en enseñar al modelo, mediante ejemplos previamente segmentados, cómo debe realizar esa tarea sobre imágenes que nunca ha visto.

6.2. ¿Qué es U-Net?

En 2015 fue propuesta una arquitectura denominada **U-Net**, desarrollada por Olaf Ronneberger y colaboradores para la segmentación de imágenes biomédicas. Su objetivo inicial consistía en identificar automáticamente regiones de interés, como células, tejidos o lesiones, permitiendo separar con precisión estas estructuras del resto de la imagen. Gracias a su excelente desempeño, esta arquitectura se convirtió rápidamente en un referente para problemas de segmentación y, actualmente, es una de las más utilizadas en aplicaciones de visión por computador, incluyendo agricultura de precisión, imágenes satelitales, inspección industrial, vehículos autónomos y diagnóstico médico.

A diferencia de una red neuronal diseñada para clasificación (por ejemplo, **saludable**, **roya nivel 1**, **roya nivel 2**, etc.), cuya salida corresponde a una única etiqueta para toda la imagen, una **U-Net** genera una imagen de salida con las mismas dimensiones que la imagen de entrada. En otras palabras, tanto la entrada como la salida de la arquitectura corresponden a imágenes. La

diferencia es que, en la imagen de salida, **cada píxel representa la predicción realizada por el modelo**.

En nuestro caso de estudio, cada píxel de la imagen de salida indicará si pertenece a la hoja principal de café o al fondo, permitiendo construir automáticamente la máscara de segmentación.

Pero, te preguntarás ¿Internamente como está conformada una U-Net para poder realizar segmentación de imagen? Los bloques que la conforman y su manera de conexión se presenta en la Figura 101.

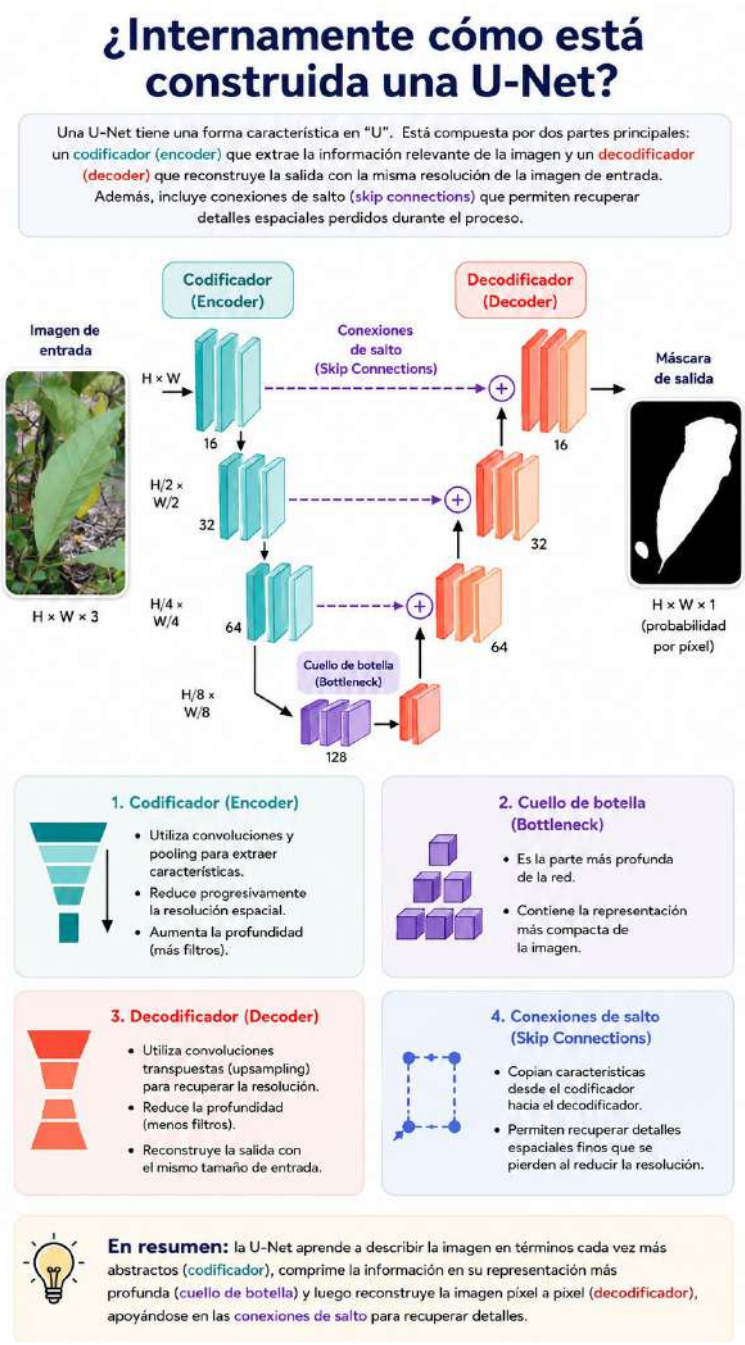


Figura 102. Explicación de la arquitectura U-Net.

Una característica importante de U-Net es que el número de canales cambia continuamente a medida que la información avanza por la red. La imagen de entrada corresponde a una imagen RGB de dimensiones $H \times W \times 3$, donde los tres canales representan las componentes de color **rojo (R)**, **verde (G)** y **azul (B)**.

Sin embargo, después de las primeras convoluciones (como vimos en el Capítulo 4 entre la imagen y el kernel), esos tres canales dejan de representar colores y son reemplazados por un conjunto de **mapas de características** aprendidos automáticamente por la red. Por ejemplo, el primer bloque puede generar **16 mapas de características**, el siguiente **32**, luego **64** y así sucesivamente. Cada uno de estos mapas responde a diferentes patrones presentes en la imagen, como *bordes*, *texturas*, *formas* o combinaciones más complejas aprendidas durante el entrenamiento.

Mientras el codificador reduce progresivamente la resolución espacial de la imagen, el número de mapas de características aumenta, permitiendo representar la información de manera cada vez más rica y abstracta. Posteriormente, durante el decodificador ocurre el proceso contrario: la resolución espacial vuelve a aumentar, mientras que el número de canales disminuye gradualmente hasta obtener una única imagen de salida de dimensiones $H \times W \times 1$.

Ese único canal ya no representa un color, sino una **probabilidad**. En nuestro caso de estudio, cada píxel almacena la probabilidad de pertenecer a la hoja de café. Finalmente, aplicando un umbral (por ejemplo, 0.5), esa imagen probabilística se transforma en la máscara binaria utilizada para segmentar automáticamente la hoja principal del fondo.

6.3. Preparación del conjunto de imágenes

Como ocurre con cualquier modelo supervisado basado en aprendizaje profundo, una U-Net no puede aprender únicamente observando imágenes. Para entrenarla es necesario disponer de ejemplos donde se conozca previamente la respuesta correcta (es decir, parejas de entrada vs. salida). En un problema

de segmentación, esto significa que **cada imagen de entrada debe estar acompañada de una máscara de referencia**, también conocida como **Ground Truth**, que indique cuáles píxeles pertenecen al objeto de interés y cuáles corresponden al fondo.

En nuestro caso de estudio, el objetivo consiste en segmentar automáticamente la hoja principal de café presente en cada fotografía. Por esta razón, para cada imagen original construimos manualmente una máscara de referencia, eliminando el fondo y conservando únicamente la hoja de interés. Estas máscaras representan la respuesta correcta que la U-Net utilizará durante el entrenamiento para aprender la tarea de segmentación.

Nota aclaratoria: en el aprendizaje supervisado, el modelo aprende a partir de ejemplos donde se conoce previamente la respuesta correcta. Durante el entrenamiento, compara continuamente su predicción con dicha respuesta y ajusta sus parámetros para reducir el error. Este mismo principio se utiliza en problemas de clasificación, regresión y segmentación de imágenes.

La Figura 103 presenta las cinco parejas de imágenes utilizadas en este capítulo. En la fila superior se muestran las imágenes originales del *dataset* RoCoLe y, en la fila inferior, las correspondientes **máscaras de referencia (*Ground Truth*)**, obtenidas mediante la herramienta **iloveimg** y posteriormente refinadas mediante edición manual. Durante el entrenamiento, la red recibe la imagen original como entrada y compara continuamente su predicción con la máscara de referencia, ajustando sus parámetros para reducir el error entre ambas.

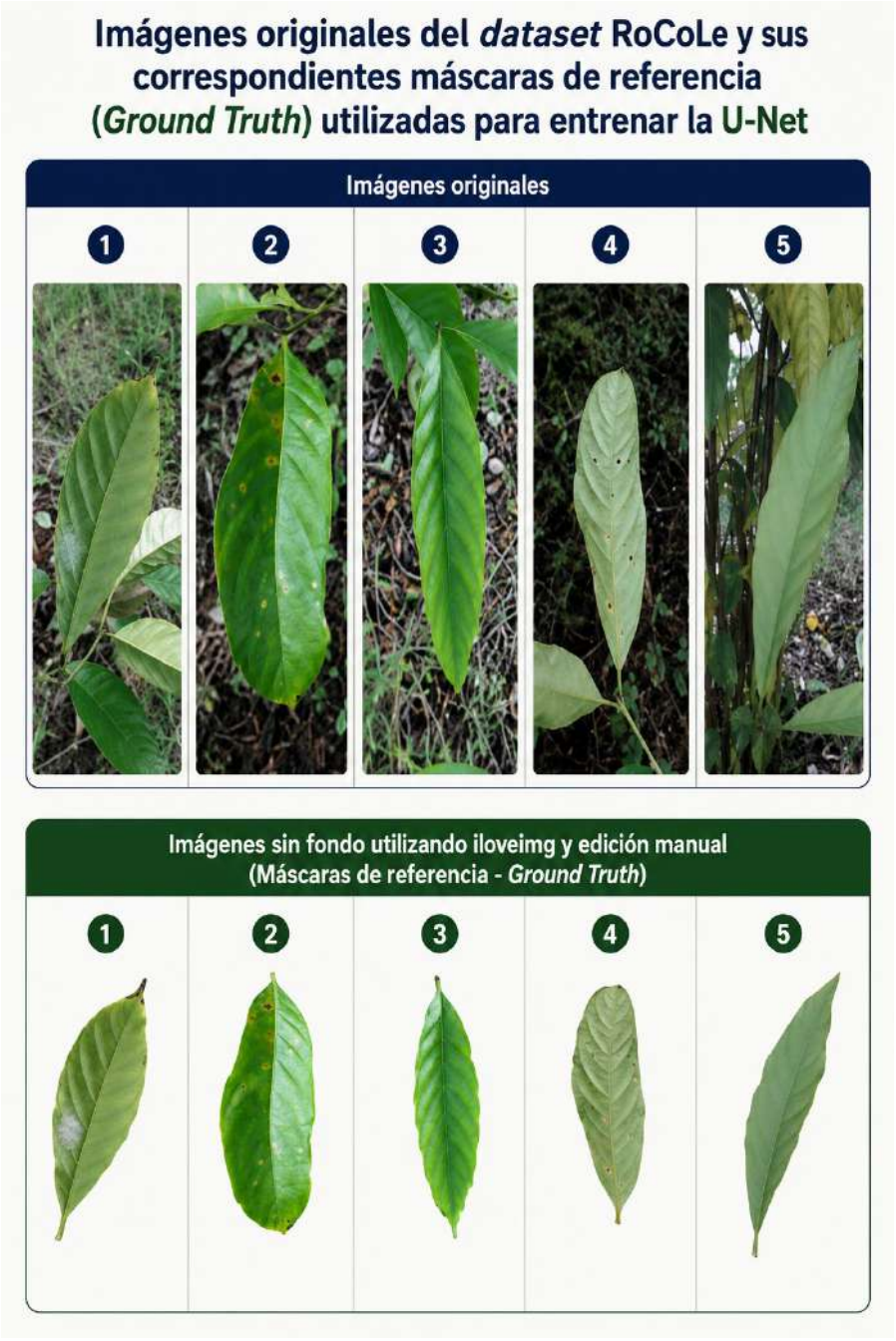


Figura 103. Imágenes originales del dataset RoCoLe y sus correspondientes máscaras de referencia (Ground Truth) utilizadas para entrenar la U-Net.

Aunque en aplicaciones reales suelen emplearse cientos o miles de parejas de entrenamiento, en este capítulo utilizaremos únicamente cinco imágenes con fines didácticos. Este conjunto reducido nos permitirá comprender el flujo completo de entrenamiento de una red de segmentación y, al mismo tiempo, analizar cómo el tamaño del conjunto de entrenamiento influye directamente en la capacidad de generalización del modelo.

6.4. Entrenamiento de la U-Net y Resultados

Una vez preparado el conjunto de entrenamiento, el siguiente paso consiste en entrenar la U-Net para que aprenda a segmentar automáticamente la hoja de café. Durante este proceso, la red analiza repetidamente cada una de las parejas de entrenamiento, formadas por la imagen original y su correspondiente máscara de referencia, ajustando progresivamente sus parámetros internos con el propósito de reducir el error entre la segmentación predicha y la segmentación esperada.

Desde un punto de vista conceptual, el entrenamiento de una U-Net puede resumirse en cuatro etapas fundamentales: lectura y preprocesamiento de las imágenes, construcción de la arquitectura, proceso de entrenamiento, y evaluación del modelo. La Figura 104 resume el flujo general seguido durante el entrenamiento de la U-Net.

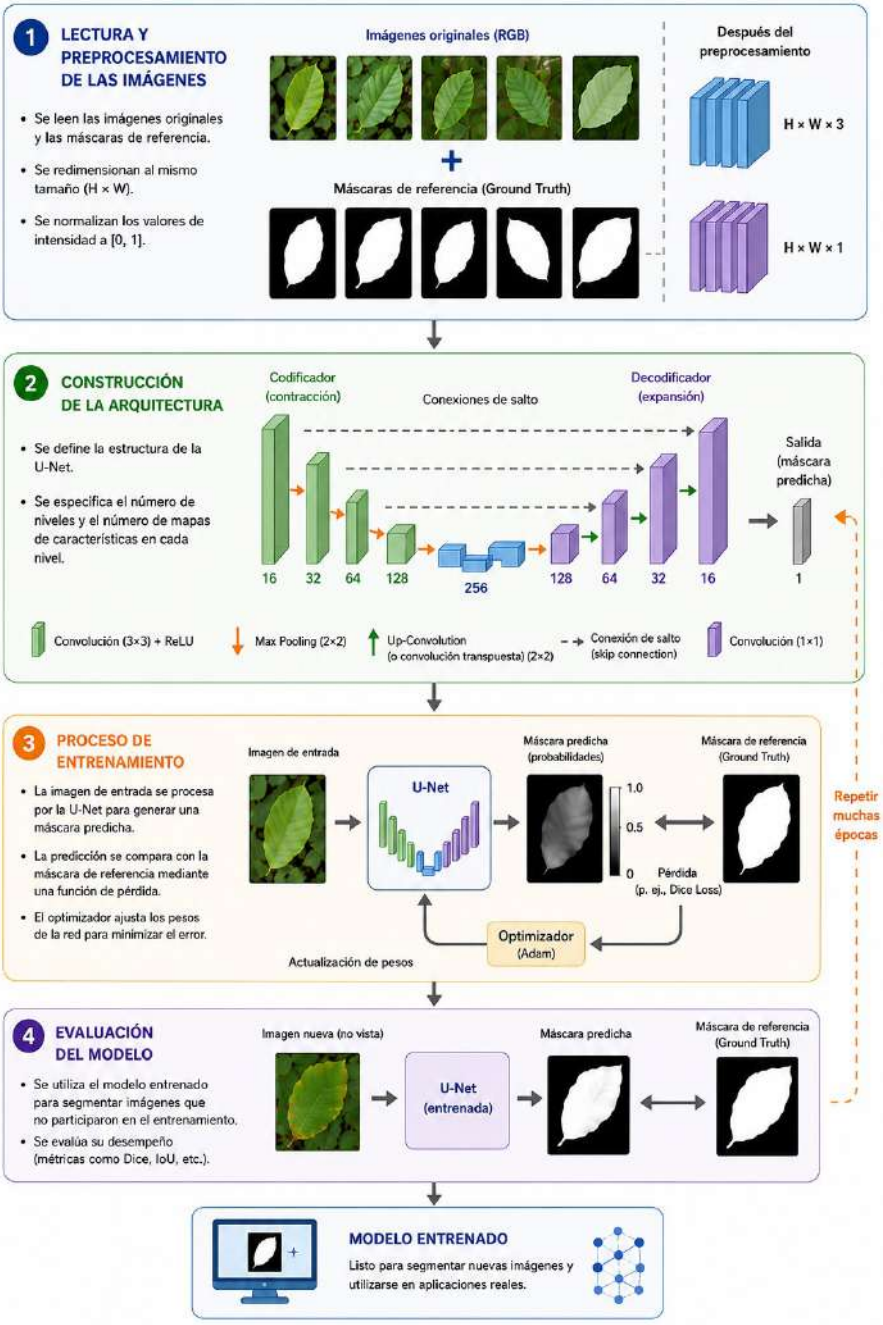


Figura 104. Flujo general de entrenamiento de una U-Net.

A manera de listado, en cada una de las etapas realizaremos:

Lectura y preprocesamiento de las imágenes

- Se verifican las dimensiones del dataset.
- Se calcula el porcentaje de hoja en cada máscara.
- Se visualizan las cinco parejas de entrenamiento.
- El estudiante comprueba que los datos quedaron correctamente preparados.

Construcción de la arquitectura

- Se genera automáticamente el resumen (*summary*) de la U-Net.
- El estudiante verifica que la arquitectura construida corresponde a la explicada en la teoría.

Proceso de entrenamiento

- Se observan las métricas (Dice, IoU y pérdida).
- Se visualizan las curvas de entrenamiento y validación.
- El estudiante comprende cómo evoluciona el aprendizaje de la red.

Evaluación del modelo

- Se obtiene la imagen de probabilidades.
- Se genera la máscara binaria.
- Se produce la imagen segmentada.
- Se compara con el *Ground Truth*.

A continuación, presentaremos un ejemplo de solución utilizando U-Net.

Ejemplo 30. Entrenamiento de una U-Net para la segmentación de la hoja principal del dataset RoCoLe

En este ejemplo entrenaremos una U-Net para segmentar automáticamente la hoja principal presente en las imágenes del *dataset* RoCoLe. Para ello utilizaremos las cinco parejas preparadas previamente, formadas por la imagen original y su correspondiente máscara de referencia. Cuatro parejas se emplearán durante el entrenamiento y la quinta se reservará para evaluar el desempeño del modelo sobre una imagen que no participó en el ajuste de sus parámetros.

Debido a que el conjunto de entrenamiento es extremadamente reducido, el objetivo de este ejemplo no consiste en construir un sistema robusto, sino en comprobar si la arquitectura es capaz de aprender la tarea de segmentación y analizar las limitaciones asociadas a trabajar con pocos datos.

Vamos a dividir el código en varios apartados:

```python
import cv2
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.keras import layers, Model

# ============================================================
# 1. LECTURA Y PREPROCESAMIENTO DE LAS IMÁGENES
# ============================================================
# 1.1. Configuración inicial
np.random.seed(42)
tf.random.set_seed(42)

TAM = 224

originales = ["C10P11E2.jpg", "C10P10H2.jpg", "C10P10H1.jpg",
              "C10P10E2.jpg", "C10P10E1.jpg"]

editadas = ["mascara_1.png", "mascara_2.png", "mascara_3.png",
            "mascara_4.png", "mascara_5.png"]

for nombre in originales + editadas:
    if not os.path.exists(nombre):
        raise FileNotFoundError(f"No se encontró el archivo: {nombre}")
```

```python
# 1.2. Lectura y normalización de las imágenes originales
X = []

for nombre in originales:
    imagen_bgr = cv2.imread(nombre)

    if imagen_bgr is None:
        raise ValueError(f"No fue posible leer la imagen: {nombre}")

    imagen_rgb = cv2.cvtColor(imagen_bgr, cv2.COLOR_BGR2RGB)
    imagen_rgb = cv2.resize(imagen_rgb, (TAM, TAM),
                            interpolation=cv2.INTER_AREA)
    imagen_rgb = imagen_rgb.astype(np.float32) / 255.0
    X.append(imagen_rgb)

X = np.array(X, dtype=np.float32)
```

```python
# 1.3. Construcción de las máscaras de referencia
Y = []
for nombre in editadas:
    imagen = cv2.imread(nombre, cv2.IMREAD_UNCHANGED)
    if imagen is None:
        raise ValueError(f"No fue posible leer la imagen editada: {nombre}")
    # Caso 1: imagen PNG con transparencia
    if imagen.ndim == 3 and imagen.shape[2] == 4:
        canal_alfa = imagen[:, :, 3]
        mascara = (canal_alfa > 10).astype(np.uint8)
    # Caso 2: imagen con fondo blanco
    else:
        if imagen.ndim == 2:
            imagen_rgb = cv2.cvtColor(imagen, cv2.COLOR_GRAY2RGB)
        else:
            imagen_rgb = cv2.cvtColor(imagen, cv2.COLOR_BGR2RGB)

        fondo_blanco = np.all(imagen_rgb >= 245, axis=2)
        mascara = (~fondo_blanco).astype(np.uint8)
    mascara = cv2.resize(mascara, (TAM, TAM),
                         interpolation=cv2.INTER_NEAREST)
    mascara = mascara.astype(np.float32)
    mascara = np.expand_dims(mascara, axis=-1)
    Y.append(mascara)
Y = np.array(Y, dtype=np.float32)
```

```python
# 1.4. Verificación de imágenes y máscaras
print("Forma de X:", X.shape)
print("Forma de Y:", Y.shape)

for i in range(len(Y)):
    valores = np.unique(Y[i])
    porcentaje_hoja = 100 * np.mean(Y[i])
    print(f"Máscara {i + 1}: valores = {valores}, "
          f"porcentaje de hoja = {porcentaje_hoja:.2f}%")

plt.figure(figsize=(15, 6))

for i in range(5):
    plt.subplot(2, 5, i + 1)
    plt.imshow(X[i])
    plt.title(f"Original {i + 1}")
    plt.axis("off")

    plt.subplot(2, 5, i + 6)
    plt.imshow(Y[i, :, :, 0], cmap="gray", vmin=0, vmax=1)
    plt.title(f"Máscara {i + 1}")
    plt.axis("off")

plt.tight_layout()
plt.show()
```

```python
# 1.5. División entre entrenamiento y validación
X_train, Y_train = X[:4], Y[:4]
X_val, Y_val = X[4:], Y[4:]

print("Entrenamiento:", X_train.shape, Y_train.shape)
print("Validación:", X_val.shape, Y_val.shape)
```

Figura 105. Código en Python del Ejemplo 30: Sección Lectura y Preprocesamiento de las Imágenes.

Esta **primera sección**, denominada **Lectura y Preprocesamiento de las Imágenes**, está conformada por las siguientes etapas: configuración inicial, lectura y normalización de las imágenes originales, construcción de las máscaras de referencia, verificación de imágenes y máscaras y división del conjunto de datos en entrenamiento y validación.

Es importante que tanto las imágenes originales como las máscaras de referencia tengan las dimensiones requeridas por la arquitectura U-Net (224 × 224 píxeles) y que sus valores se encuentren normalizados. Estas etapas garantizan que todas las parejas de entrenamiento sean consistentes y puedan ser procesadas adecuadamente por la red neuronal. Finalmente, de las cinco parejas disponibles, cuatro se destinan al entrenamiento del modelo y la restante se reserva para la validación, permitiendo evaluar su capacidad de segmentar una imagen que no participó en el proceso de aprendizaje.

Ahora, continuamos con la segunda sección de código, que se presenta en la Figura 106.

```python
# ==========================================
# 2. CONSTRUCCIÓN DE LA ARQUITECTURA
# ==========================================
# 2.1. Métricas y función de pérdida
def dice_coef(y_true, y_pred, suavizado=1e-6):
    y_true = tf.reshape(tf.cast(y_true, tf.float32), [-1])
    y_pred = tf.reshape(tf.cast(y_pred, tf.float32), [-1])

    interseccion = tf.reduce_sum(y_true * y_pred)
    suma = tf.reduce_sum(y_true) + tf.reduce_sum(y_pred)

    return (2.0 * interseccion + suavizado) / (suma + suavizado)

def iou_coef(y_true, y_pred, suavizado=1e-6):
    y_true = tf.reshape(tf.cast(y_true, tf.float32), [-1])
    y_pred = tf.reshape(tf.cast(y_pred, tf.float32), [-1])

    interseccion = tf.reduce_sum(y_true * y_pred)
    union = tf.reduce_sum(y_true) + tf.reduce_sum(y_pred) - interseccion

    return (interseccion + suavizado) / (union + suavizado)

def perdida_bce_dice(y_true, y_pred):
    bce = tf.keras.losses.binary_crossentropy(y_true, y_pred)
    bce = tf.reduce_mean(bce)
    return bce + (1.0 - dice_coef(y_true, y_pred))


# 2.2. Bloque convolucional
def bloque_conv(x, filtros):
    x = layers.Conv2D(filtros, 3, padding="same", activation="relu",
                      kernel_initializer="he_normal")(x)

    x = layers.Conv2D(filtros, 3, padding="same", activation="relu",
                      kernel_initializer="he_normal")(x)
    return x
```

```python
# 2.3. Construcción de una U-Net pequeña
entrada = layers.Input(shape=(TAM, TAM, 3))

# Codificador
c1 = bloque_conv(entrada, 16)
p1 = layers.MaxPooling2D((2, 2))(c1)
c2 = bloque_conv(p1, 32)
p2 = layers.MaxPooling2D((2, 2))(c2)
c3 = bloque_conv(p2, 64)
p3 = layers.MaxPooling2D((2, 2))(c3)

# Cuello de botella
puente = bloque_conv(p3, 128)

# Decodificador
u3 = layers.Conv2DTranspose(64, 2, strides=2, padding="same")(puente)
u3 = layers.Concatenate()([u3, c3])
c4 = bloque_conv(u3, 64)
u2 = layers.Conv2DTranspose(32, 2, strides=2, padding="same")(c4)
u2 = layers.Concatenate()([u2, c2])
c5 = bloque_conv(u2, 32)
u1 = layers.Conv2DTranspose(16, 2, strides=2, padding="same")(c5)
u1 = layers.Concatenate()([u1, c1])
c6 = bloque_conv(u1, 16)

# Salida: probabilidad de pertenecer a la hoja
salida = layers.Conv2D(1, 1, activation="sigmoid")(c6)
modelo = Model(inputs=entrada, outputs=salida, name="UNet_pequena")
modelo.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-4),
               loss=perdida_bce_dice,
               metrics=[dice_coef, iou_coef])
modelo.summary()
```

Figura 106. Código en Python del Ejemplo 30: Sección Construcción de la Arquitectura.

Esta **segunda sección**, denominada **Construcción de la Arquitectura**, implementa una U-Net de tamaño reducido para la segmentación automática de la hoja principal. En ella se definen los bloques que conforman el codificador, el cuello de botella y el decodificador, así como las conexiones de salto entre ambos. Finalmente, se especifican el optimizador, la función de pérdida y las métricas utilizadas para entrenar y evaluar el modelo.

A continuación, pasamos a la tercera sección.

```
# ============================================================
# 3. PROCESO DE ENTRENAMIENTO
# ============================================================
# 3.1. Configuración de callbacks
callbacks = [
    tf.keras.callbacks.EarlyStopping(
        monitor="val_loss", patience=25, restore_best_weights=True
    ),

    tf.keras.callbacks.ReduceLROnPlateau(
        monitor="val_loss", factor=0.5, patience=10, min_lr=1e-6
    )
]

# 3.2. Entrenamiento de la U-Net
historial = modelo.fit(
    X_train, Y_train,
    validation_data=(X_val, Y_val),
    epochs=150,
    batch_size=1,
    callbacks=callbacks,
    verbose=1
)
```

```
# 3.3. Curvas de entrenamiento
plt.figure(figsize=(8, 4))
plt.plot(historial.history["loss"], label="Pérdida entrenamiento")
plt.plot(historial.history["val_loss"], label="Pérdida validación")

plt.xlabel("Época")
plt.ylabel("Pérdida")
plt.title("Evolución del entrenamiento")
plt.legend()
plt.tight_layout()
plt.show()
```

Figura 107. Código en Python del Ejemplo 30: Sección Proceso de Entrenamiento.

En la **tercera sección**, denominada **Proceso de Entrenamiento**, se configuran los parámetros necesarios para entrenar la U-Net. En primer lugar, se definen los *callbacks*, encargados de detener automáticamente el entrenamiento cuando el modelo deja de mejorar y de ajustar la tasa de aprendizaje si es necesario. Posteriormente, la red procesa de forma iterativa las imágenes de entrenamiento, compara las máscaras predichas con las máscaras de referencia y actualiza sus parámetros internos para reducir el error. Finalmente, se presentan las curvas de entrenamiento y validación, las cuales permiten analizar la evolución del proceso de aprendizaje a lo largo de las diferentes épocas.

Finalmente, tenemos la cuarta sección del código.

```
# ============================================================
# 4. EVALUACIÓN DEL MODELO
# ============================================================

# 4.1. Predicción sobre la imagen de validación
prediccion = modelo.predict(X_val, verbose=0)[0, :, :, 0]

umbral = 0.5
mascara_predicha = (prediccion >= umbral).astype(np.uint8)
segmentada = X_val[0] * mascara_predicha[:, :, np.newaxis]


# 4.3. Comparación con la máscara de referencia
plt.figure(figsize=(12, 4))

plt.subplot(1, 3, 1)
plt.imshow(Y_val[0, :, :, 0], cmap="gray", vmin=0, vmax=1)
plt.title("Máscara de referencia")
plt.axis("off")

plt.subplot(1, 3, 2)
plt.imshow(prediccion, cmap="gray", vmin=0, vmax=1)
plt.title("Probabilidad predicha")
plt.axis("off")

plt.subplot(1, 3, 3)
plt.imshow(mascara_predicha, cmap="gray", vmin=0, vmax=1)
plt.title("Máscara binaria predicha")
plt.axis("off")

plt.tight_layout()
plt.show()
```

```
# 4.2. Visualización del resultado
plt.figure(figsize=(16, 4))

plt.subplot(1, 4, 1)
plt.imshow(X_val[0])
plt.title("Imagen original")
plt.axis("off")

plt.subplot(1, 4, 2)
plt.imshow(prediccion, cmap="gray", vmin=0, vmax=1)
plt.title("Probabilidad predicha")
plt.axis("off")

plt.subplot(1, 4, 3)
plt.imshow(mascara_predicha, cmap="gray", vmin=0, vmax=1)
plt.title("Máscara predicha")
plt.axis("off")

plt.subplot(1, 4, 4)
plt.imshow(segmentada)
plt.title("Imagen segmentada")
plt.axis("off")

plt.tight_layout()
plt.show()
```

Figura 108. Código en Python del Ejemplo 30: Sección Evaluación del Modelo.

En la **cuarta sección**, denominada **Evaluación del Modelo,** se utiliza la U-Net entrenada para segmentar la imagen reservada para validación. Inicialmente, la red genera una imagen de probabilidades, donde cada píxel indica la probabilidad de pertenecer a la hoja principal. Posteriormente, mediante la aplicación de un umbral, esta imagen probabilística se convierte en una máscara binaria de segmentación. Finalmente, la máscara obtenida se aplica sobre la imagen original y se compara con la máscara de referencia (*Ground Truth*), permitiendo evaluar visualmente el desempeño alcanzado por el modelo.

Pasemos ahora a analizar los resultados obtenidos durante la etapa de preprocesamiento del conjunto de entrenamiento. Inicialmente, verificaremos las dimensiones de las imágenes de entrada y de salida, así como el porcentaje de la imagen ocupado por la hoja en cada una de las máscaras de referencia.

```
Forma de X: (5, 224, 224, 3)
Forma de Y: (5, 224, 224, 1)
Máscara 1: valores = [0. 1.], porcentaje de hoja = 31.03%
Máscara 2: valores = [0. 1.], porcentaje de hoja = 32.79%
Máscara 3: valores = [0. 1.], porcentaje de hoja = 25.52%
Máscara 4: valores = [0. 1.], porcentaje de hoja = 19.97%
Máscara 5: valores = [0. 1.], porcentaje de hoja = 25.48%
```

Observamos que el conjunto de datos está conformado por **cinco imágenes de entrada (X)** y sus correspondientes **cinco máscaras de referencia (Y)**. Todas las imágenes fueron redimensionadas a **224 × 224 píxeles**, de acuerdo con los requerimientos de la arquitectura implementada. Sin embargo, las imágenes de entrada poseen **tres canales** (RGB), mientras que las máscaras contienen **un único canal**, donde el valor **1** representa la hoja y el valor **0** corresponde al fondo.

Adicionalmente, el porcentaje de la imagen ocupado por la hoja varía aproximadamente entre el **20 % y el 33 %**, lo que indica que la mayor parte de los píxeles pertenece al fondo. Esta característica es habitual en problemas de segmentación y explica la importancia de utilizar funciones de pérdida, como la combinación de **Binary Cross-Entropy** y **Dice Loss**, que permiten entrenar el

modelo de manera más robusta cuando existe un desbalance entre la región de interés y el fondo.

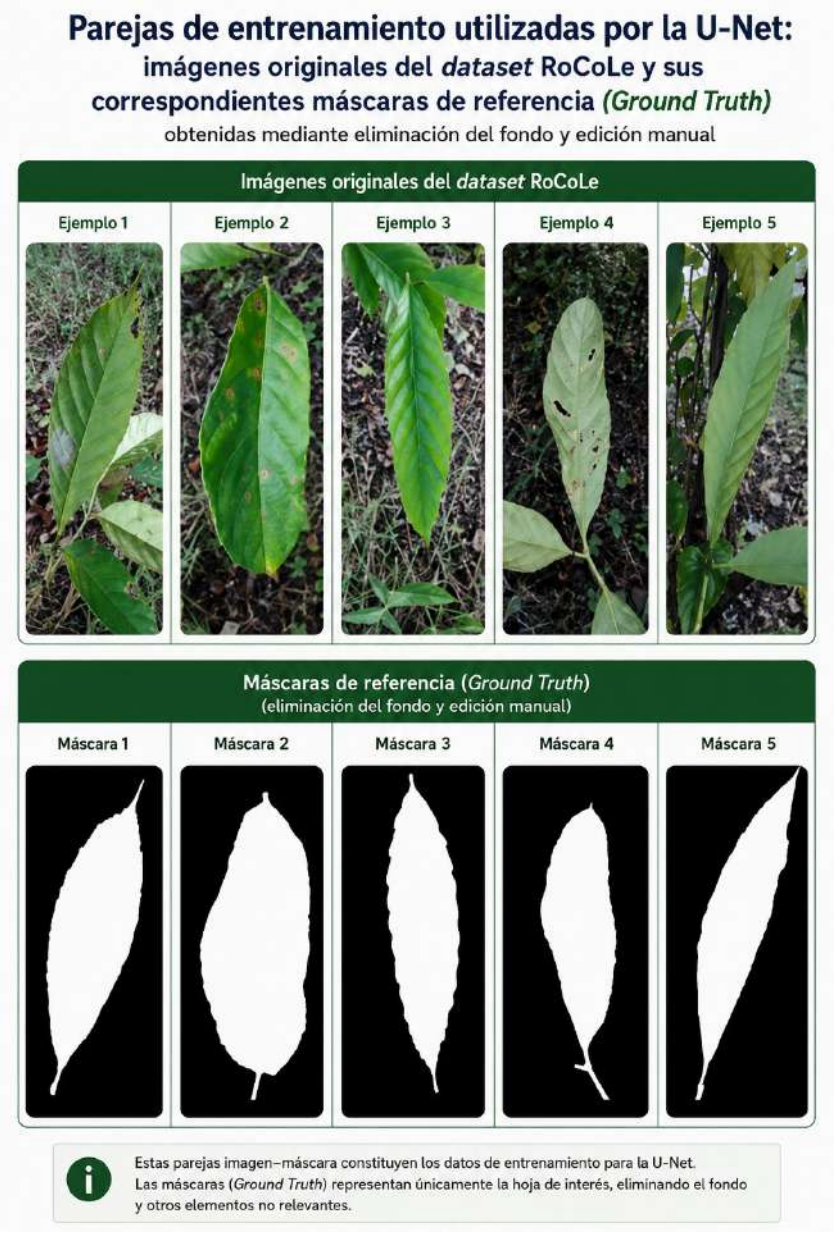


Figura 109. Parejas de entrenamiento utilizadas por la U-Net: imágenes originales del *dataset* RoCoLe y sus correspondientes máscaras de referencia (*Ground Truth*) obtenidas mediante eliminación del fondo y edición manual.

La **Figura 109** presenta las máscaras de referencia (Ground Truth) obtenidas al **finalizar la Sección 1 del código**, correspondiente a la etapa de Lectura y Preprocesamiento de las Imágenes. Es importante destacar que estas no corresponden exactamente a las imágenes editadas manualmente mostradas en la Figura 102. Dichas imágenes constituyen únicamente el punto de partida. Durante el preprocesamiento, el código identifica automáticamente la región de la hoja (a partir del canal alfa o del fondo blanco), genera una máscara binaria, la redimensiona a 224 × 224 píxeles y la convierte en una imagen de un solo canal, donde el valor 1 representa la hoja y el valor 0 el fondo. **Estas son las máscaras que finalmente utiliza la U-Net durante el entrenamiento**.

Posteriormente, obtenemos la información de la arquitectura, así:

Entrenamiento: (4, 224, 224, 3) (4, 224, 224, 1)
Validación: (1, 224, 224, 3) (1, 224, 224, 1)
Model: "UNet_pequeno"

Layer (type)	Output Shape	Param #	Connected to
input_layer (InputLayer)	(None, 224, 224, 3)	0	-
conv2d (Conv2D)	(None, 224, 224, 16)	448	input_layer[0][0]
conv2d_1 (Conv2D)	(None, 224, 224, 16)	2,320	conv2d[0][0]
max_pooling2d (MaxPooling2D)	(None, 112, 112, 16)	0	conv2d_1[0][0]
conv2d_2 (Conv2D)	(None, 112, 112, 32)	4,640	max_pooling2d[0]…
conv2d_3 (Conv2D)	(None, 112, 112, 32)	9,248	conv2d_2[0][0]
max_pooling2d_1 (MaxPooling2D)	(None, 56, 56, 32)	0	conv2d_3[0][0]
conv2d_4 (Conv2D)	(None, 56, 56, 64)	18,496	max_pooling2d_1[…
conv2d_5 (Conv2D)	(None, 56, 56, 64)	36,928	conv2d_4[0][0]
max_pooling2d_2 (MaxPooling2D)	(None, 28, 28, 64)	0	conv2d_5[0][0]
conv2d_6 (Conv2D)	(None, 28, 28, 128)	73,856	max_pooling2d_2[…
conv2d_7 (Conv2D)	(None, 28, 28, 128)	147,584	conv2d_6[0][0]
conv2d_transpose (Conv2DTranspose)	(None, 56, 56, 64)	32,832	conv2d_7[0][0]
concatenate (Concatenate)	(None, 56, 56, 128)	0	conv2d_transpose… conv2d_5[0][0]
conv2d_8 (Conv2D)	(None, 56, 56, 64)	73,792	concatenate[0][0]
conv2d_9 (Conv2D)	(None, 56, 56, 64)	36,928	conv2d_8[0][0]
conv2d_transpose_1 (Conv2DTranspose)	(None, 112, 112, 32)	8,224	conv2d_9[0][0]
concatenate_1 (Concatenate)	(None, 112, 112, 64)	0	conv2d_transpose… conv2d_3[0][0]
conv2d_10 (Conv2D)	(None, 112, 112, 32)	18,464	concatenate_1[0]…
conv2d_11 (Conv2D)	(None, 112, 112, 32)	9,248	conv2d_10[0][0]
conv2d_transpose_2 (Conv2DTranspose)	(None, 224, 224, 16)	2,064	conv2d_11[0][0]
concatenate_2 (Concatenate)	(None, 224, 224, 32)	0	conv2d_transpose… conv2d_1[0][0]
conv2d_12 (Conv2D)	(None, 224, 224, 16)	4,624	concatenate_2[0]…
conv2d_13 (Conv2D)	(None, 224, 224, 16)	2,320	conv2d_12[0][0]
conv2d_14 (Conv2D)	(None, 224, 224, 1)	17	conv2d_13[0][0]

Total params: 482,033 (1.84 MB)
Trainable params: 482,033 (1.84 MB)
Non-trainable params: 0 (0.00 B)

Figura 110. Resumen dela arquitectura U-Net construida por TensorFlow, mostrando las capas, las dimensiones de salida y el número de parámetros entrenables.

La **Figura 110** presenta el resumen de la arquitectura U-Net construida en la **Sección 2 del código**, correspondiente a la etapa de **Construcción de la Arquitectura**. En ella se observan las capas que conforman el codificador, el cuello de botella y el decodificador, así como las dimensiones de salida de cada una y el número de parámetros entrenables. En total, la red está compuesta por **482 mil parámetros** (aprox.), los cuales serán ajustados automáticamente durante el entrenamiento.

A continuación, analizamos los resultados obtenidos durante el entrenamiento de la U-Net, generados en la **tercera sección** del código, correspondiente al ***Proceso de Entrenamiento*.** En cada época, el modelo evalúa su desempeño tanto sobre las imágenes de entrenamiento como sobre la imagen reservada para validación. Para este ejemplo, en la última época ejecutada se obtuvo:

```
Epoch 76/150
4/4 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3s 719ms/step - dice_coef: 0.9557 - iou_coef: 0.9155 - loss: 0.0965 - val_dice_coef: 0.8591 - val_iou_coef: 0.7530 - val_loss: 0.4993 - learning_rate: 2.5000e-05
```

El **coeficiente Dice** mide el grado de coincidencia entre la máscara predicha y la máscara de referencia, tomando valores entre **0 y 1**, donde **1 representa una segmentación perfecta**. En este caso, un valor de **0.9557** indica que la U-Net logró una alta coincidencia sobre las imágenes utilizadas durante el entrenamiento.

Por su parte, el **coeficiente IoU (*Intersection over Union*)** cuantifica la proporción de solapamiento entre la región segmentada por el modelo y la región real. Al igual que el coeficiente Dice, sus valores oscilan entre **0 y 1**, siendo **1 el valor ideal**. El resultado obtenido (**0.9155**) confirma que existe una elevada correspondencia entre ambas regiones.

En cuanto a la validación, los valores de **Dice** e **IoU** son ligeramente inferiores a los obtenidos durante el entrenamiento, lo cual es esperable, ya que el modelo está siendo evaluado sobre una imagen que no participó en el proceso de aprendizaje. No obstante, la diferencia entre ambos conjuntos no es excesiva, lo que sugiere que la U-Net ha logrado una capacidad razonable de generalización, a pesar del reducido tamaño del conjunto de entrenamiento.

En este ejemplo se fijó una semilla aleatoria para favorecer la reproducibilidad del experimento. Por ello, si el código se ejecuta nuevamente bajo las mismas condiciones de software y hardware, es esperable obtener resultados muy similares. Sin embargo, pequeñas diferencias pueden aparecer al utilizar versiones distintas de TensorFlow, diferentes dispositivos de procesamiento (CPU o GPU) u otros entornos de ejecución.

Adicionalmente, obtenemos la gráfica de evolución del entrenamiento:

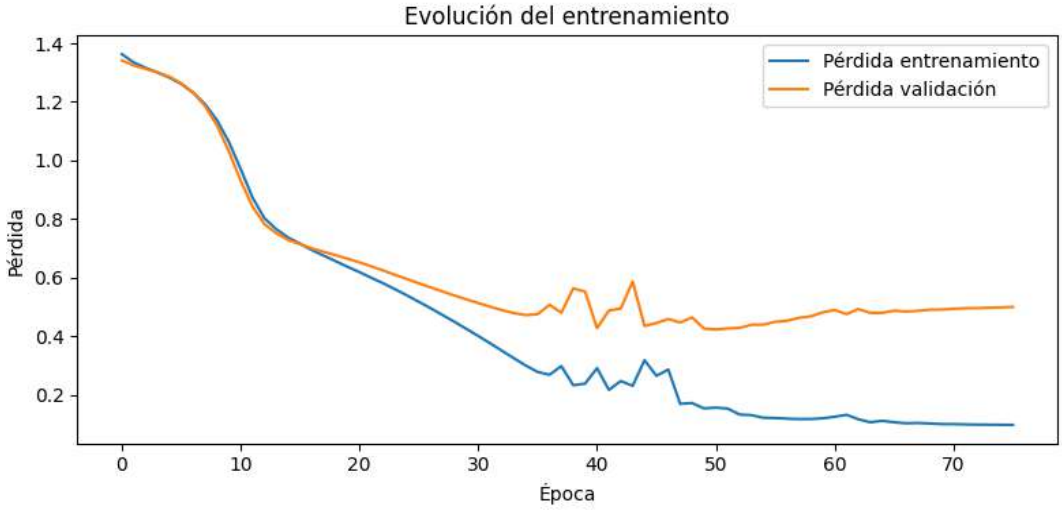


Figura 111. Evolución del entrenamiento de la U-Net.

La **Figura 111** presenta la evolución de la función de pérdida durante el entrenamiento de la U-Net para los conjuntos de entrenamiento y validación. En ambos casos se observa una disminución progresiva del error durante las primeras épocas, indicando que la red aprende gradualmente a segmentar la hoja principal. La pérdida comienza alrededor de **1.35** y disminuye hasta valores cercanos a **0.10** para entrenamiento y **0.43** para validación.

Recordemos que el **valor ideal de la función de pérdida es 0**, ya que indicaría una coincidencia perfecta entre la máscara predicha y la máscara de referencia. En este ejemplo, a partir de aproximadamente la **época 35**, la pérdida de validación deja de disminuir y comienza a estabilizarse, motivo por el cual el

criterio de **EarlyStopping** detiene automáticamente el entrenamiento antes de completar las 150 épocas programadas.

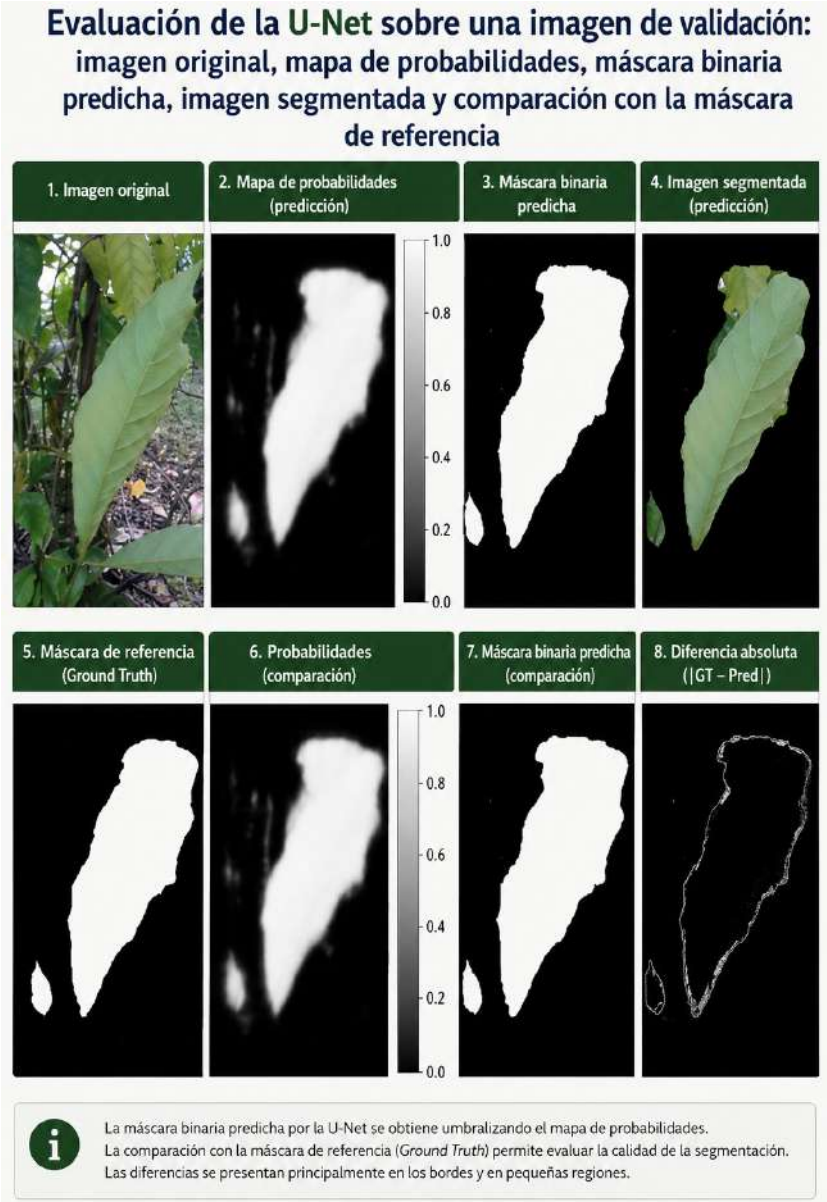


Figura 112. Evaluación de la U-Net sobre una imagen de validación: imagen original, mapa de probabilidades, máscara binaria predicha, imagen segmentada y comparación con la máscara de referencia.

La **Figura 112** presenta el resultado obtenido durante la etapa de evaluación, correspondiente a la **cuarta sección** del código: **Evaluación del Modelo**. Inicialmente, la U-Net recibe como entrada una imagen del *dataset* RoCoLe que no participó en el entrenamiento. Como salida, la red genera una **imagen de probabilidades**, donde los píxeles más claros indican una mayor probabilidad de pertenecer a la hoja principal y los más oscuros corresponden al fondo. Posteriormente, mediante la aplicación de un umbral, esta imagen se convierte en una **máscara binaria**, la cual se utiliza para eliminar el fondo y obtener la imagen segmentada.

En la parte inferior de la figura se compara la **máscara de referencia (*Ground Truth*)** con la máscara predicha por la U-Net. Aunque la segmentación no es perfecta, el modelo logra identificar correctamente la mayor parte de la hoja principal, conservando su forma general y eliminando gran parte del fondo. Considerando que la red fue entrenada únicamente con cuatro imágenes y evaluada sobre una quinta imagen no vista previamente, este resultado demuestra la capacidad de las arquitecturas U-Net para aprender la tarea de segmentación incluso a partir de conjuntos de entrenamiento muy reducidos.

6.5. Comparación entre la segmentación tradicional y la segmentación mediante U-Net

Para finalizar este capítulo, compararemos los resultados obtenidos mediante la mejor estrategia de segmentación tradicional desarrollada en el Capítulo 5 (Solución 3) con los obtenidos utilizando una U-Net entrenada con las mismas imágenes. Aunque ambos enfoques persiguen el mismo objetivo (aislar la hoja principal del fondo), la forma en que lo consiguen es completamente diferente.

La segmentación tradicional se basa en un conjunto de reglas definidas explícitamente por el programador, tales como umbrales de color, operaciones morfológicas, análisis de componentes conectados y criterios basados en el área y la posición de las regiones. En contraste, la U-Net no recibe estas reglas de manera explícita, sino que las aprende automáticamente a partir de ejemplos de entrenamiento (Ver Figura 113).

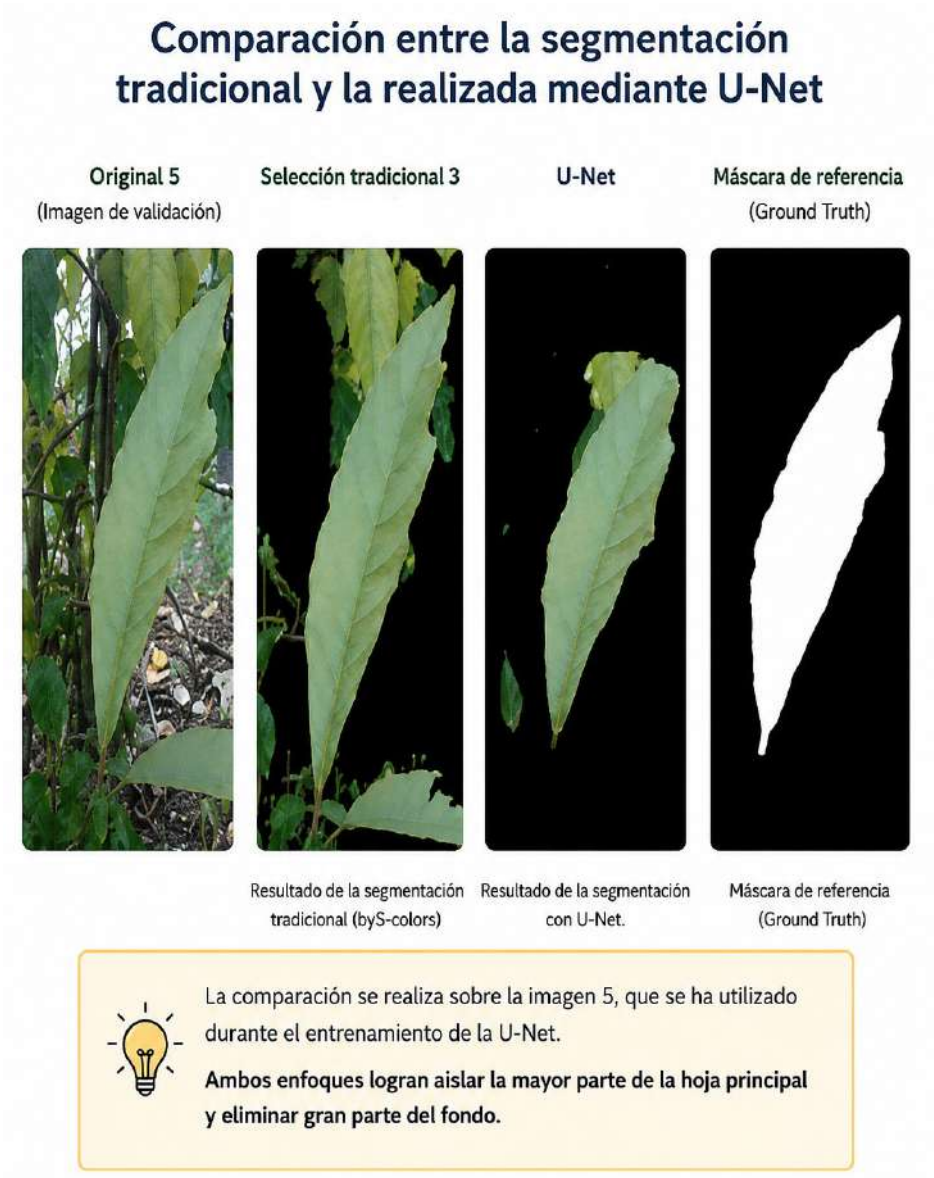


Figura 113. Comparación entre la segmentación tradicional y la realizada mediante U-Net. De izquierda a derecha: imagen original, imagen segmentada con la solución 3 del Capítulo 5, imagen segmentada con U-Net (con 4 imágenes de entrenamiento), y máscara de referencia para U-Net.

Vamos a comparar, en primer lugar, los resultados desde un punto de vista cualitativo. En el caso del enfoque tradicional, la imagen segmentada logró eliminar gran parte del fondo, generando un *background* negro; sin embargo, conservó algunas hojas secundarias cercanas a la hoja principal, especialmente en la parte superior central y en la esquina inferior derecha de la imagen. Por su parte, la U-Net eliminó una mayor proporción del fondo y únicamente conservó pequeños fragmentos de hojas secundarias. No obstante, también perdió una pequeña parte de la hoja principal, correspondiente a la punta ubicada en el extremo superior.

Ninguna de las dos soluciones produjo una segmentación perfecta. Sin embargo, es importante tener en cuenta que la U-Net fue entrenada únicamente con **cuatro parejas de imágenes** (imagen original–máscara de referencia), una cantidad extremadamente reducida para un modelo de aprendizaje profundo. Es de esperarse que, al incrementar el número de ejemplos de entrenamiento, por ejemplo, a **80 imágenes para entrenamiento y 20 para validación**, el modelo logre una mejor capacidad de generalización y, en consecuencia, produzca segmentaciones más precisas.

De esta forma, queda claro que el ejemplo desarrollado en este capítulo tiene un propósito principalmente didáctico. En un proyecto real sería necesario trabajar con un conjunto de entrenamiento mucho más amplio y diverso, que permitiera explotar plenamente el potencial de la U-Net y obtener modelos con una mayor capacidad de generalización. Asimismo, la comparación entre el enfoque tradicional y el basado en aprendizaje profundo no debería limitarse a una evaluación visual, sino complementarse con métricas objetivas, como el **coeficiente Dice** y el **índice IoU**, calculadas sobre un conjunto suficientemente representativo de imágenes de prueba.

6.6. ¿Qué aprendimos en esta sección?

Con este capítulo llegamos al final de nuestro recorrido por el procesamiento digital de imágenes. Durante estas páginas descubrimos que una imagen es mucho más que una fotografía: es una fuente de información que puede analizarse, transformarse y utilizarse para resolver problemas reales de ingeniería. Aprendimos que detrás de una buena solución no existe un único

algoritmo, sino una secuencia de decisiones fundamentadas, donde cada etapa influye en el resultado final.

También comprobamos que una misma situación puede abordarse desde perspectivas completamente diferentes. Primero construimos una solución utilizando técnicas clásicas de procesamiento digital de imágenes, diseñando explícitamente cada regla del algoritmo. Después resolvimos exactamente el mismo problema mediante Inteligencia Artificial, permitiendo que una arquitectura U-Net aprendiera automáticamente a segmentar la imagen a partir de ejemplos. Ninguno de estos enfoques es universalmente mejor que el otro; cada uno aporta fortalezas diferentes y, en muchos casos, la mejor solución surge precisamente de combinarlos.

Espero que, al terminar esta sección, no solo hayas aprendido nuevas técnicas de procesamiento digital de imágenes o Inteligencia Artificial. Espero, sobre todo, que hayas fortalecido una forma de pensar.

Porque las herramientas cambiarán, aparecerán nuevos algoritmos y nuevas arquitecturas, pero la capacidad de observar un problema, comprenderlo, plantear una estrategia, evaluar críticamente los resultados y seguir mejorándolos seguirá siendo una de las cualidades que distinguen a un buen ingeniero.

AL CERRAR ESTE LIBRO

Si llegaste hasta aquí, quiero darte las gracias por haber compartido conmigo este recorrido. Espero que estas páginas te hayan permitido descubrir que el procesamiento digital de señales no es únicamente un conjunto de ecuaciones, algoritmos o líneas de código. Detrás de cada herramienta existe una forma de pensar, una manera de observar un problema, analizar la información disponible y construir soluciones fundamentadas. Esa forma de razonar es, quizá, el aprendizaje más valioso que puede dejar este libro.

Vivimos una época fascinante para la ingeniería. Cada día aparecen nuevas librerías, nuevos modelos de Inteligencia Artificial y nuevas herramientas capaces de resolver problemas que hace apenas unos años parecían imposibles. Sin embargo, estoy convencida de que los fundamentos seguirán siendo nuestro mejor punto de apoyo. Las tecnologías evolucionarán, los lenguajes de programación cambiarán y surgirán arquitecturas más potentes que las que hoy conocemos, pero la capacidad de comprender un problema, formular una estrategia y evaluar críticamente una solución seguirá marcando la diferencia entre utilizar una herramienta y entender realmente cómo resolver un reto de ingeniería.

Si este libro despertó en ti la curiosidad por seguir aprendiendo, cuestionando y experimentando, entonces habrá cumplido su propósito. Ojalá dentro de algunos años no recuerdes cada una de las ecuaciones o cada línea de código que aquí presentamos, pero sí conserves una idea que me ha acompañado durante toda mi vida profesional: **la ingeniería no consiste en memorizar algoritmos, sino en aprender a pensar para crear soluciones que mejoren nuestro entorno.** Si estas páginas lograron transmitirte esa forma de ver nuestra profesión, todo el esfuerzo de escribir este libro habrá valido la pena.