

```
In [1]: import warnings  
import numpy as np  
  
warnings.filterwarnings('ignore')
```

3. Transformaciones de intensidad y filtrado espacial

El término *dominio espacial* se refiere al plano de la imagen en sí, y los métodos de procesamiento de imágenes de esta categoría se basan en la manipulación directa de los píxeles de una imagen.

Esto contrasta con el procesamiento de imágenes en un dominio de transformación que implica primero transformar una imagen en el dominio de transformación, realizar el procesamiento allí y obtener la transformación inversa para devolver los resultados en el dominio espacial.

Dos categorías principales de procesamiento espacial son las transformaciones de intensidad y el filtrado espacial.

- Las transformaciones de intensidad operan en píxeles individuales de una imagen para tareas como manipulación de contraste y umbralización de imágenes.
- El filtrado espacial realiza operaciones en la vecindad de cada píxel de una imagen. Ejemplos de filtrado espacial incluyen el suavizado y la nitidez de imágenes.

3.1. Contexto

- Todas las técnicas de procesamiento de imágenes que veremos en esta unidad se implementan en el dominio espacial, que es el plano que contiene los píxeles de una imagen.
- Las técnicas de dominio espacial operan directamente sobre los píxeles de una imagen
- Como verá, algunas tareas de procesamiento de imágenes son más fáciles o más significativas de implementar en el dominio espacial, mientras que otras son más adecuadas para otros enfoques.

3.1.1. Fundamentos de las transformaciones de intensidad y el filtrado especial

Los procesos de dominio espacial que analizamos se basan en la expresión

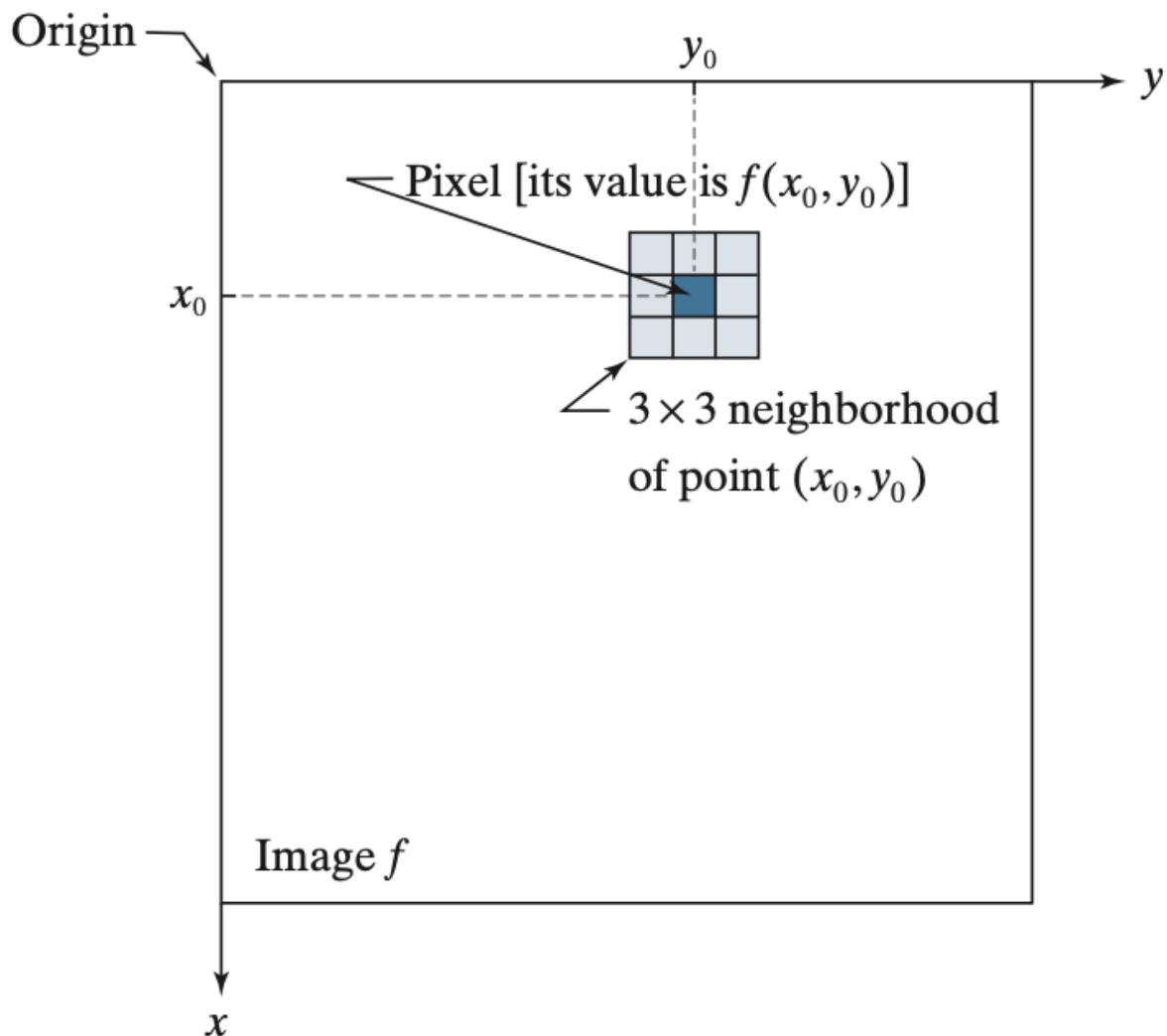
$$g(x, y) = T[f(x, y)]$$

donde $f(x, y)$ es una imagen de entrada, $g(x, y)$ es la imagen de salida y T es un operador en f definido sobre una vecindad del punto (x, y) .

El operador se puede aplicar a los píxeles de una sola imagen o a los píxeles de un conjunto de imágenes, como realizar la suma por elementos de una secuencia de imágenes para reducir el ruido.

La figura muestra la implementación básica de la ecuación.

- El punto (x_0, y_0) que se muestra es una ubicación arbitraria en la imagen y la pequeña región que se muestra es una vecindad de (x_0, y_0) . Normalmente, la vecindad es rectangular, centrada en (x_0, y_0) y de tamaño mucho más pequeño que la imagen.
- El proceso consiste en mover el centro de la vecindad de un píxel a otro y aplicar el operador T a los píxeles de la vecindad para obtener un valor de salida en esa ubicación. Por lo tanto, para cualquier ubicación específica (x_0, y_0) , el valor de la imagen de salida g en esas coordenadas es igual al resultado de aplicar T a la vecindad con origen en (x_0, y_0) en f .



Por ejemplo, supongamos que la vecindad es un cuadrado de tamaño 3×3 y que el operador T se define como "calcular la intensidad promedio de los píxeles en la vecindad".

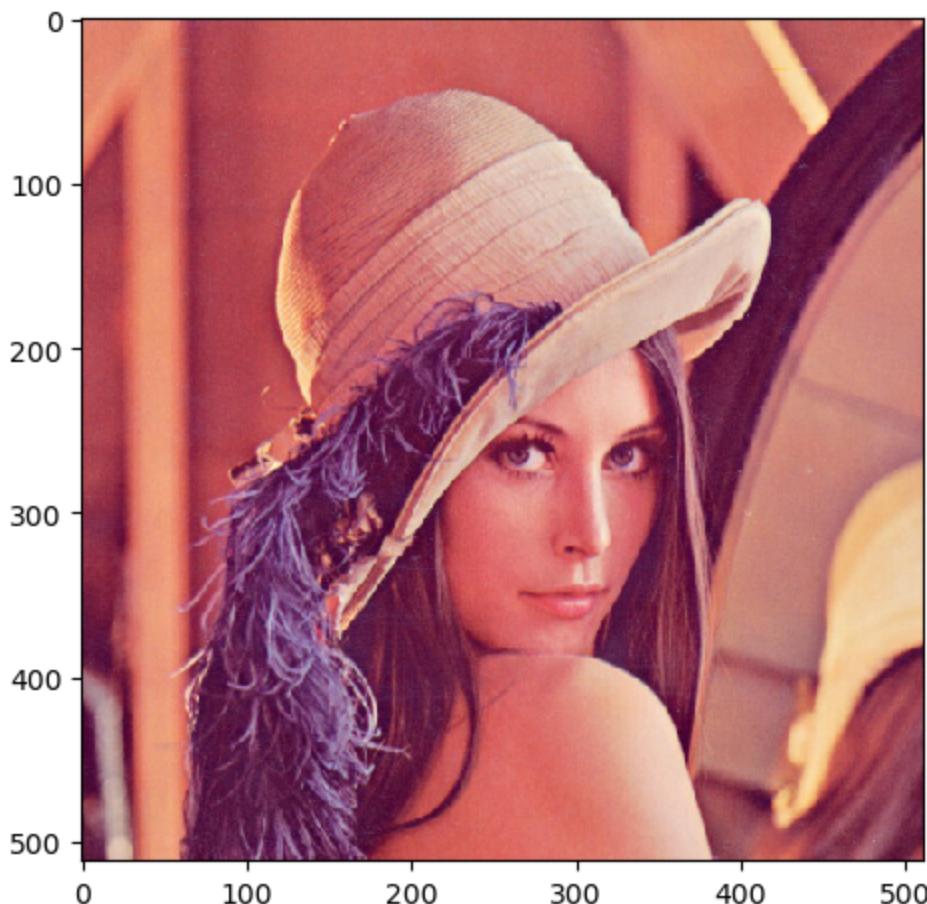
Considere una ubicación arbitraria en una imagen, digamos $(100, 150)$. El resultado en esa ubicación en la imagen de salida, $g(100, 150)$, es la suma de $f(100, 150)$ y sus 8 vecinos, dividida por 9.

Luego, el centro de la vecindad se mueve a la siguiente ubicación adyacente. y el procedimiento se repite para generar el siguiente valor de la imagen de salida g .

Normalmente, el proceso comienza en la parte superior izquierda de la imagen de entrada y avanza píxel a píxel en un escaneo horizontal (vertical), una fila (columna) a la vez.

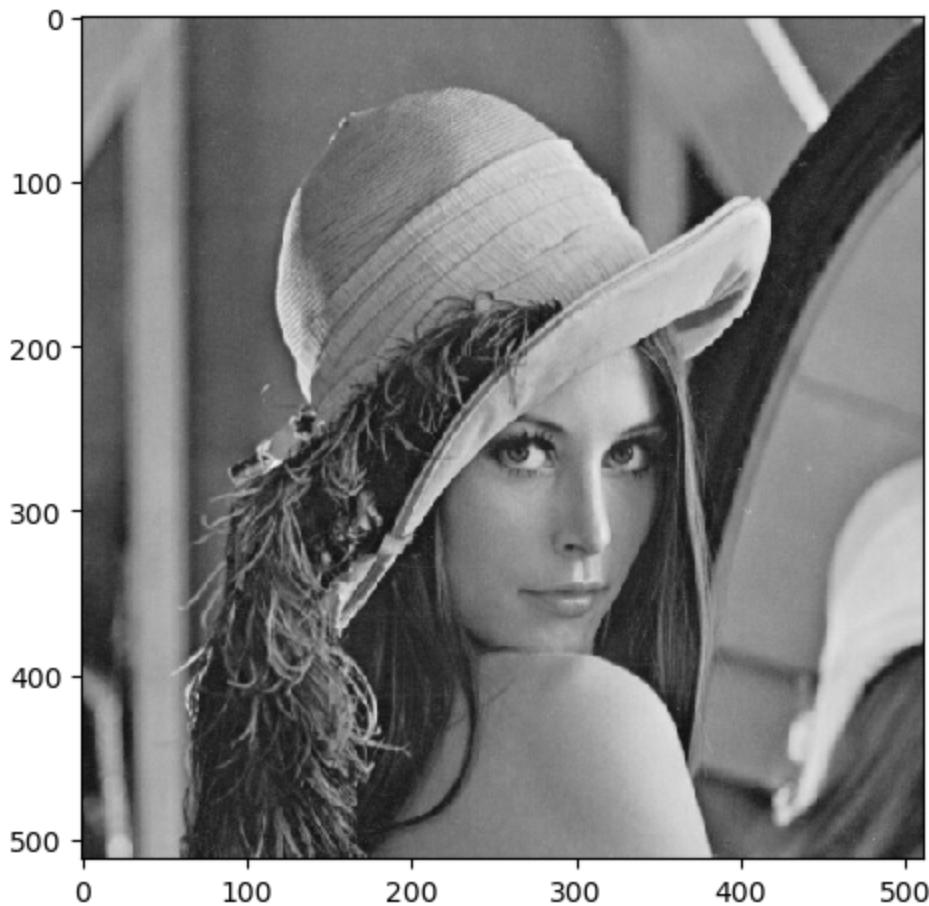
```
In [2]: from skimage.io import imread, imshow  
lena = imread('figs/lena.jpg')  
imshow(lena)
```

```
Out[2]: <matplotlib.image.AxesImage at 0x11abb76d0>
```



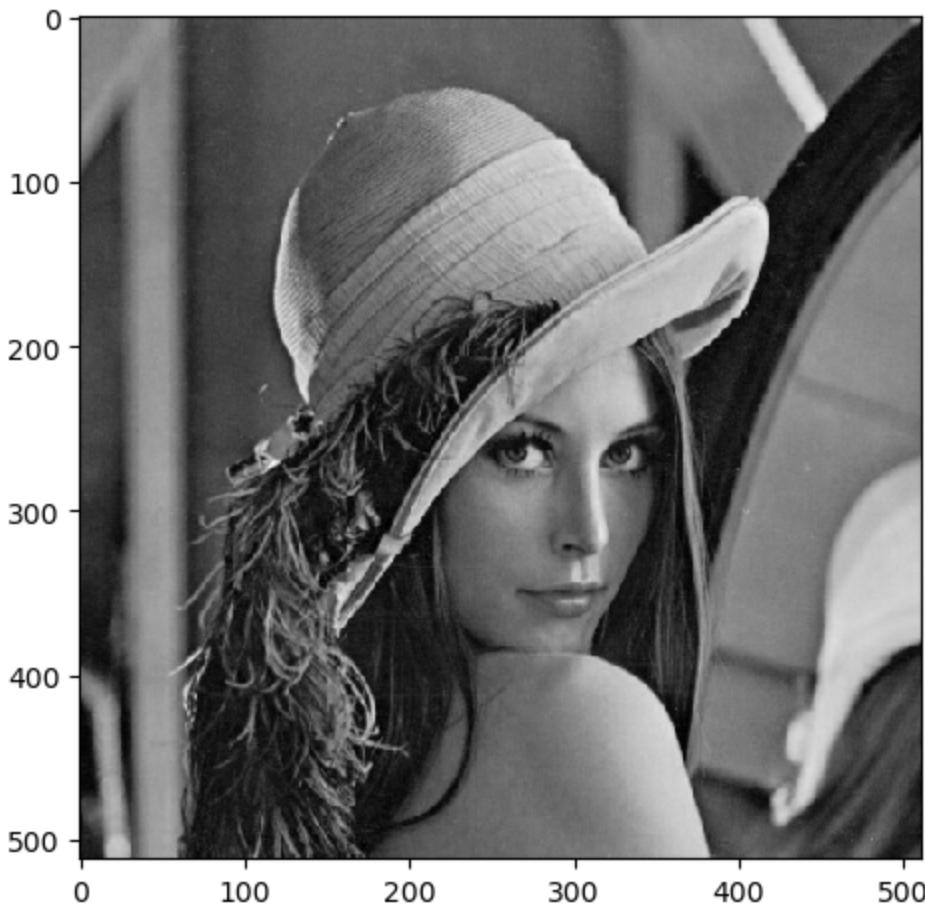
```
In [3]: lena_gris = imread('figs/lena.jpg', as_gray=True)  
imshow(lena_gris)
```

```
Out[3]: <matplotlib.image.AxesImage at 0x11ac53150>
```



```
In [4]: lena_gris = lena[:, :, 1]  
imshow(lena_gris)
```

```
Out[4]: <matplotlib.image.AxesImage at 0x11acc63d0>
```



```
In [5]: import matplotlib.pyplot as plt

def show_images(*images, titles=[], cols=2, cmap='gray', figsize=(10, 10)):
    """
    Display a list of images in a grid.
    Args:
        images (list): A list of images to display.
        titles (list): A list of titles for each image (optional).
        cols (int): The number of columns in the grid (default=2).
        cmap (str): The colormap to use (default='gray').
        figsize (tuple): The size of the figure (default=(10, 10)).
    """
    rows = len(images) // cols + (1 if len(images) % cols else 0)
    fig, axes = plt.subplots(rows, cols, figsize=figsize)
    for i, ax in enumerate(axes.flat):
        if i < len(images):
            ax.imshow(images[i], cmap=cmap)
            if titles is not None:
                ax.set_title(titles[i] if len(titles) > i else 'Imagen {}'.format(i+1))
            ax.axis('off')
    plt.tight_layout()
    plt.show()
```

```
In [71]: def aplicar_convolucion(image, kernel, pad_mode='constant'):
    """
    Aplica una convolución a una imagen usando un kernel.

    Parámetros:
        image (numpy.ndarray): La imagen a convolucionar.
    """
    pass
```

```
kernel (numpy.ndarray): El kernel a usar.
```

Regresa:

numpy.ndarray: La imagen convolucionada.

.....

```
# Obtenemos las dimensiones de la imagen y el kernel
```

```
image_height, image_width = image.shape
```

```
kernel_height, kernel_width = kernel.shape
```

```
# Calculamos el padding
```

```
pad_height = kernel_height // 2
```

```
pad_width = kernel_width // 2
```

```
# Creamos una imagen con padding
```

```
padded_image = np.pad(image, (pad_height, pad_width), pad_mode)
```

```
# Creamos una imagen de salida vacía
```

```
output_image = np.zeros_like(image)
```

```
# Apply the convolution
```

```
for y in range(pad_height, image_height + pad_height):
```

```
    for x in range(pad_width, image_width + pad_width):
```

```
        output_image[y - pad_height, x - pad_width] = np.sum(kernel * padded_image[y:y+kernel_height, x:x+kernel_width])
```

```
return output_image
```

In [72]:

```
tam_filtro = 7
filtro_promediador = np.ones((tam_filtro, tam_filtro)) / tam_filtro**2
lena_promediada = aplicar_convolucion(lena_gris, filtro_promediador)
show_images(lena_gris, lena_promediada, titles=['Original', 'Promediada'])
```

Original



Promediada



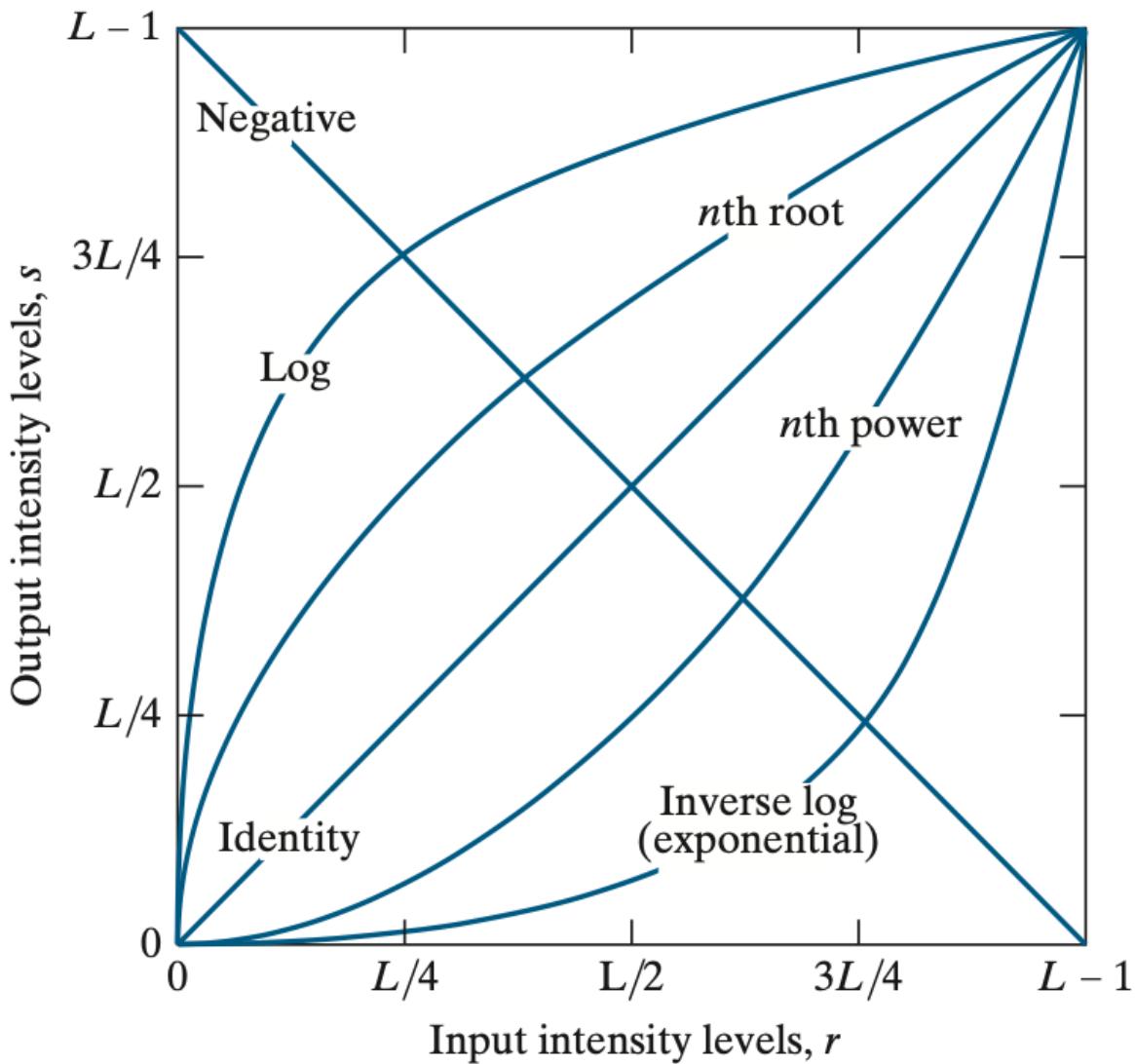
3.2. Algunas funciones básicas de transformación de intensidad

Las transformaciones de intensidad se encuentran entre las técnicas de procesamiento de imágenes más simples.

Denotaremos los valores de los píxeles, antes y después del procesamiento, por r y s , respectivamente. Estos valores están relacionados mediante una transformación T , que asigna un valor de píxel r a un valor de píxel s .

Como introducción a las transformaciones de intensidad, considere la siguiente figura, que muestra tres tipos básicos de funciones utilizadas con frecuencia en el procesamiento de imágenes:

- lineal (transformaciones negativas y de identidad),
- logarítmica (transformaciones logarítmicas y logarítmicas inversas) y
- ley de potencias (transformaciones enésimas y transformaciones de raíz enésima).
- la función de identidad es el caso trivial en el que las intensidades de entrada y salida son idénticas.



3.2.1. Imágenes negativas

El negativo de una imagen con niveles de intensidad en el rango $[0, L - 1]$ se obtiene utilizando la función de transformación negativa que se muestra en la anterior, que tiene la forma:

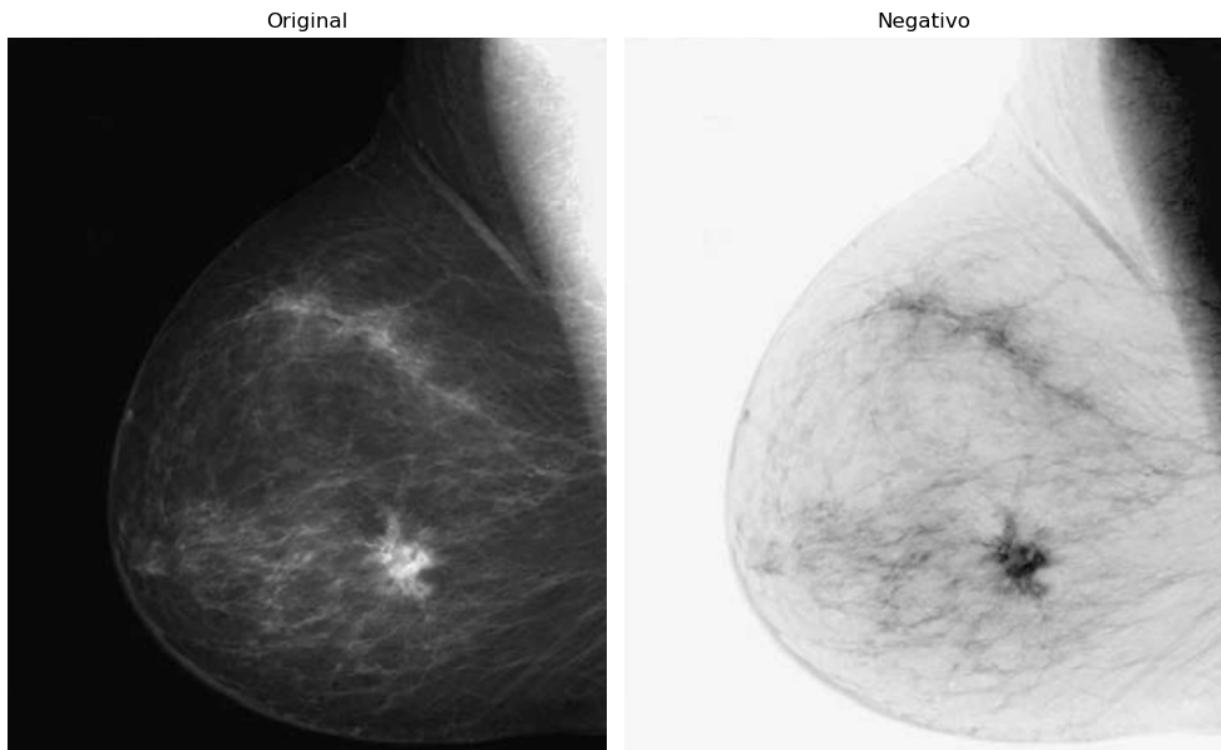
$$s = (L - 1) - r$$

Invertir los niveles de intensidad de una imagen digital de esta manera produce el equivalente a un negativo fotográfico.

Este tipo de procesamiento se utiliza, por ejemplo, para mejorar los detalles blancos o grises incrustados en las regiones oscuras de una imagen, especialmente cuando las áreas negras son dominantes en tamaño.

```
In [8]: mast = imread('figs/mastografia1.png', as_gray=True)
max = np.max(mast)
mast_neg = max - mast

show_images(mast, mast_neg, titles=['Original', 'Negativo'])
```



3.2.2. Transformaciones logarítmicas

La forma general de la transformación logarítmica es:

$$s = c \log(1 + r)$$

donde c es una constante y se supone que $r \geq 0$. La forma de la curva logarítmica muestra que esta transformación transforma un rango estrecho de valores de baja intensidad en la entrada en un rango más amplio de niveles de salida.

Por ejemplo, observe cómo los niveles de entrada en el rango $[0, L/4]$ se asignan a los niveles de salida en el rango $[0, 3L/4]$.

Por el contrario, los valores más altos de los niveles de entrada se asignan a un rango más estrecho en la salida.

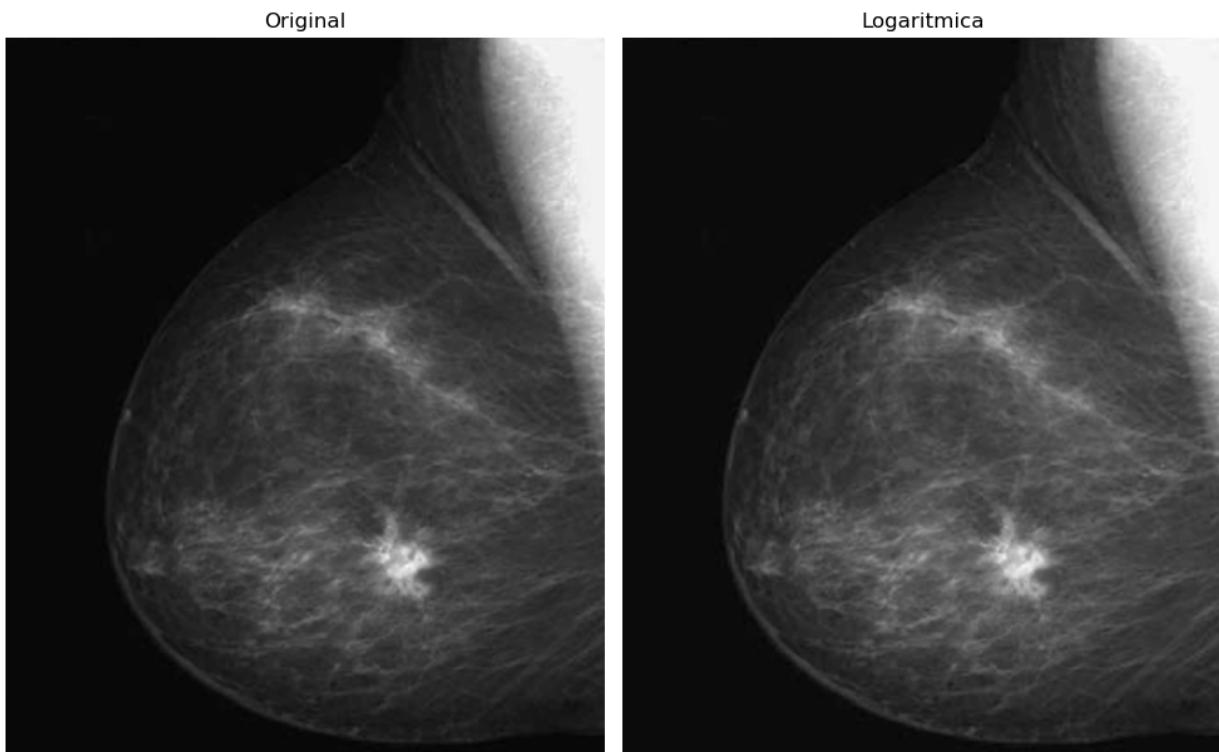
Usamos una transformación de este tipo para expandir los valores de los píxeles oscuros en una imagen, mientras comprimimos los valores de nivel superior.

Lo contrario ocurre con la transformación logarítmica inversa (exponencial).

```
In [9]: lena_log = np.log(1 + lena_gris)  
show_images(lena_gris, lena_log, titles=['Original', 'Logaritmica'])
```



```
In [10]: mast_log = np.log(1 + mast)  
show_images(mast, mast_log, titles=['Original', 'Logaritmica'])
```



3.2.3. Transformaciones de ley de potencia (gamma)

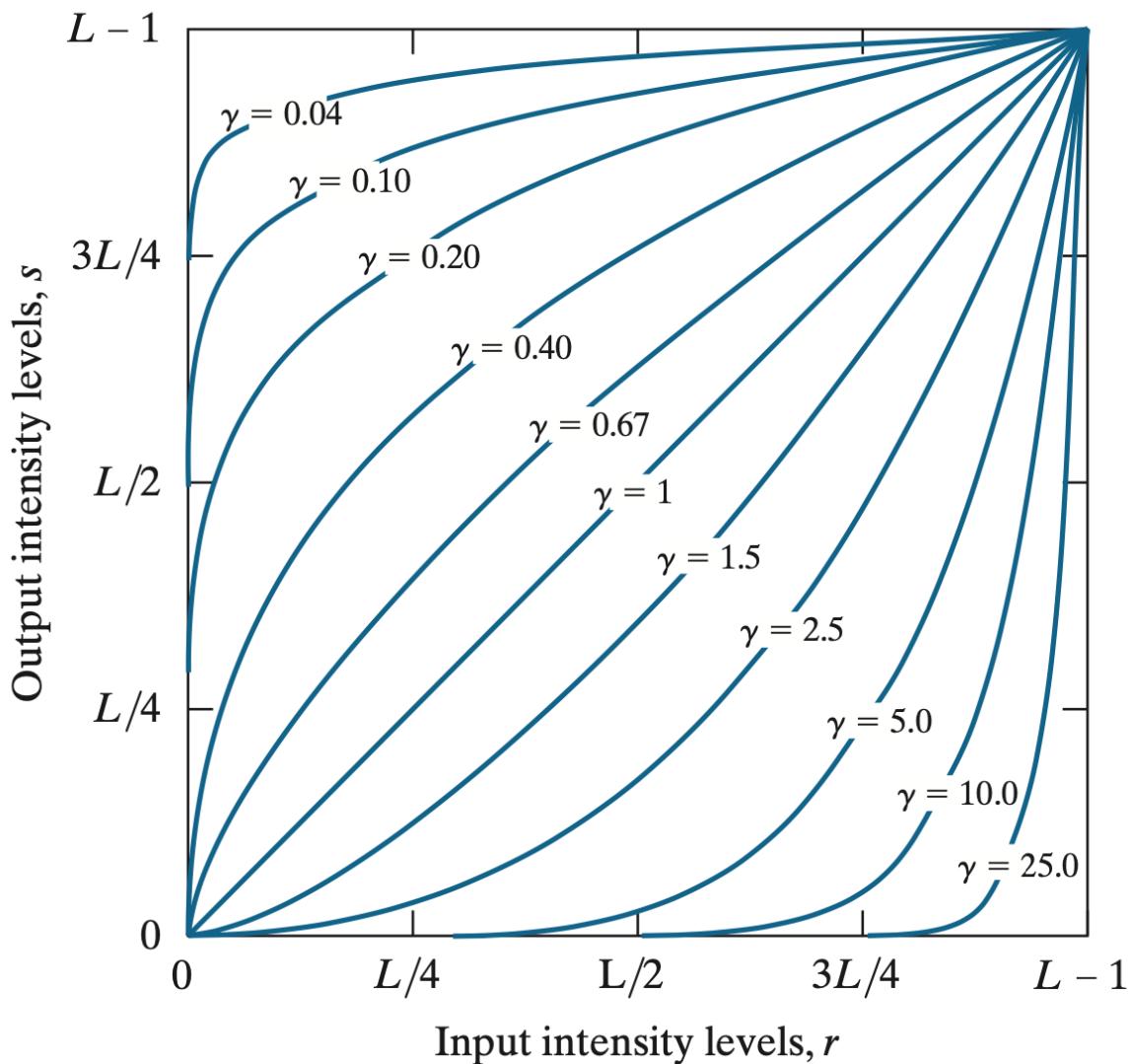
Las transformaciones de ley de potencia tienen la forma:

$$s = r^\gamma$$

donde c y γ son constantes positivas. A veces la ecuación se escribe como $s = c(r + \epsilon)^\gamma$ para tener en cuenta las compensaciones (es decir, una salida medible cuando la entrada es cero). Sin embargo, las compensaciones suelen ser una cuestión de calibración de la pantalla y, como resultado, normalmente se ignoran.

La figura muestra gráficas de s en función de r para varios valores de γ . Al igual que con las transformaciones logarítmicas, las curvas de ley de potencia con valores fraccionarios de γ asignan un rango estrecho de valores de entrada oscuros a un rango más amplio de valores de salida, y ocurre lo contrario para valores más altos de niveles de entrada.

Observe también que se puede obtener una familia de transformaciones simplemente variando γ . Las curvas generadas con valores de $\gamma > 1$ tienen exactamente el efecto opuesto a las generadas con valores de $\gamma < 1$. Cuando $c = \gamma = 1$, la ecuación se reduce a la transformación de identidad.



La respuesta de muchos dispositivos utilizados para la captura, impresión y visualización de imágenes obedece a una ley de potencia.

Por convención, el exponente en una ecuación de ley potencial se denomina gamma. El proceso utilizado para corregir estos fenómenos de respuesta de ley de potencia se llama corrección gamma o codificación gamma.

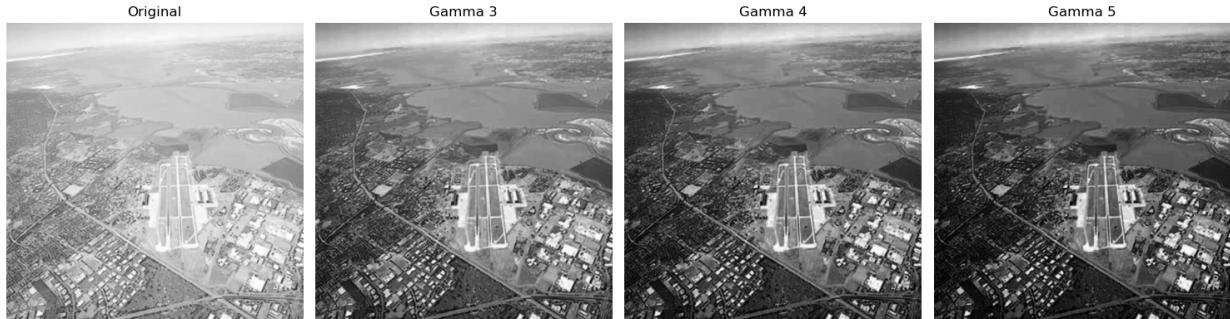
```
In [11]: gamma_ex = imread('figs/gamma2.png', as_gray=True)
gamma_ex_1 = gamma_ex**0.6
gamma_ex_2 = gamma_ex**0.4
gamma_ex_3 = gamma_ex**0.3

show_images(gamma_ex, gamma_ex_1, gamma_ex_2, gamma_ex_3,
            titles=['Original', 'Gamma 0.6', 'Gamma 0.4', 'Gamma 0.3'],
            figsize=(15, 15), cols=4)
```



```
In [12]: airport = imread('figs/gamma3.png', as_gray=True)
airport_gamma_3 = airport**3
airport_gamma_4 = airport**4
airport_gamma_5 = airport**5

show_images(airport, airport_gamma_3, airport_gamma_4, airport_gamma_5,
           titles=['Original', 'Gamma 3', 'Gamma 4', 'Gamma 5'],
           figsize=(15, 15), cols=4)
```



3.2.4. Transformación lineal por partes

Un enfoque complementario a los métodos analizados en las tres secciones anteriores es utilizar funciones lineales por partes.

La ventaja de estas funciones sobre las analizadas hasta ahora es que la forma de funciones por partes puede ser arbitrariamente compleja.

La principal desventaja de estas funciones es que su especificación requiere una considerable intervención del usuario.

Ajuste de contraste

Las imágenes de bajo contraste pueden deberse a una iluminación deficiente, a una falta de rango dinámico en el sensor de imágenes o incluso a una configuración incorrecta de la apertura de la lente durante la adquisición de imágenes.

La ampliación del contraste amplía el rango de niveles de intensidad de una imagen para que abarque el rango de intensidad completo ideal del medio de grabación o dispositivo de visualización.

La figura muestra una transformación típica utilizada para el ajuste del contraste.

Las ubicaciones de los puntos (r_1, s_1) y (r_2, s_2) controlan la forma de la función de transformación.

- Si $r_1 = s_1$ y $r_2 = s_2$ la transformación es una función lineal que no produce cambios en intensidad.
- Si $r_1 = r_2$, $s = 0$ y $s = L - 1$, la transformación se convierte en una función de umbral que crea una imagen binaria.
- Los valores intermedios de (r_1, s_1) y (r_2, s_2) producen varios grados de dispersión en los niveles de intensidad de la imagen de salida, afectando así su contraste. En general, se supone $r \leq r_1 \leq s_1 \leq s_2 \leq s$, de modo que la función tiene un solo valor y es monótonamente creciente. Esto preserva el orden de los niveles de intensidad, evitando así la creación de artefactos de intensidad.

```
In [13]: def ajustar_contraste_por_partes(image, r1, r2, s1, s2):
    """
    Ajusta el contraste de una imagen por partes.

    Parametros:
    image (numpy.ndarray): La imagen a ajustar.
    r1 (float): El punto de inicio de la primera parte.
    r2 (float): El punto final de la primera parte.
    s1 (float): El punto de inicio de la segunda parte.
    s2 (float): El punto final de la segunda parte.

    Regresa:
    numpy.ndarray: La imagen con el contraste ajustado.
    """
    # Obtenemos las dimensiones de la imagen
    image_height, image_width = image.shape

    # Creamos una imagen de salida vacía
    output_image = np.zeros_like(image)

    # Ajustamos el contraste
    for y in range(image_height):
        for x in range(image_width):
            if image[y, x] < r1:
                output_image[y, x] = s1 / r1 * image[y, x]
            elif image[y, x] < r2:
                output_image[y, x] = (s2 - s1) / (r2 - r1) * (image[y, x] - r1) + s1
            else:
                output_image[y, x] = (255 - s2) / (255 - r2) * (image[y, x] - r2) + s2

    return output_image
```

```
In [14]: ajuste_contraste = imread('figs/ajuste_contraste.png')
granos = imread('figs/granos1.png', as_gray=True)
granos = granos * 255 // 1
print(np.min(granos), np.max(granos))

# esto se pone porque matplotlib ajusta el rango de la imagen
granos[0, 0] = 0
granos[0, 1] = 255
```

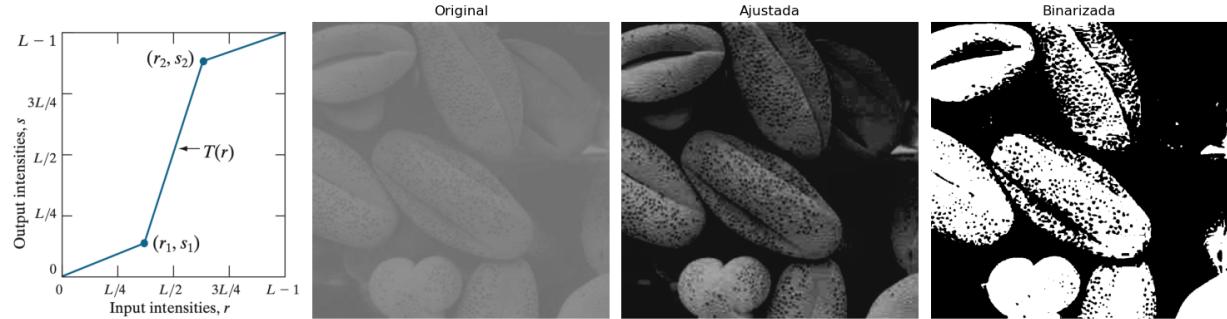
```
r1, s1 = 120, 20
r2, s2 = 150, 200

granos_ajustada = ajustar_contraste_por_partes(granos, r1, r2, s1, s2)
granos_binarizada = ajustar_contraste_por_partes(granos, 127, 127, 0, 255)

106.0 147.0
```

In [15]:

```
show_images(ajuste_contraste, granos, granos_ajustada, granos_binarizada,
            titles=['Original', 'Ajustada', 'Binarizada'],
            figsize=(15, 15), cols=4)
```



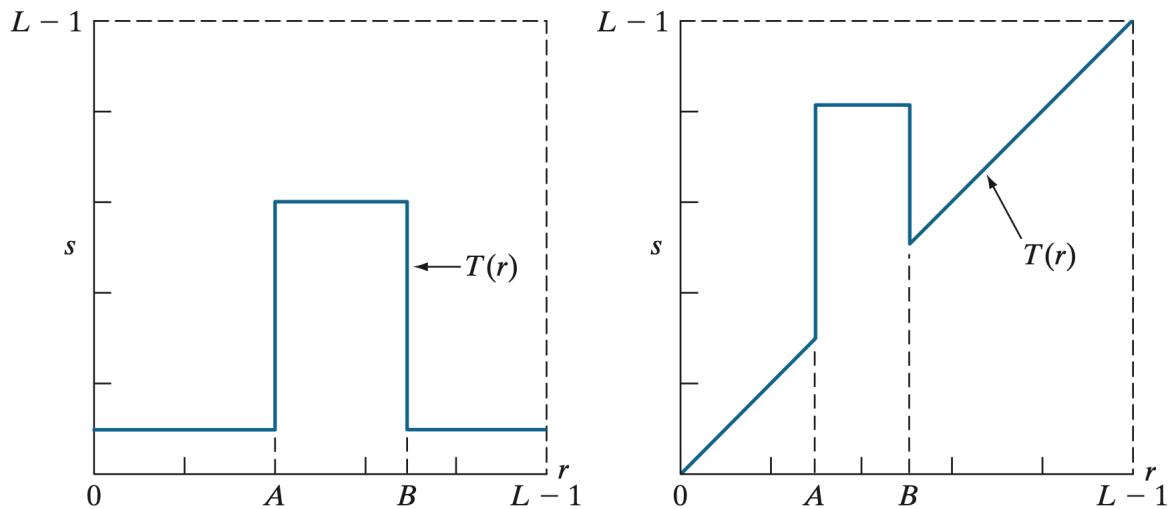
Corte por niveles de intensidad

Hay aplicaciones en las que resulta interesante resaltar un rango concreto de intensidades en una imagen.

Algunas de estas aplicaciones incluyen la mejora de características en imágenes de satélite, como masas de agua, y la mejora de defectos en imágenes de rayos X.

El método, llamado corte por niveles de intensidad, se puede implementar de varias maneras, pero la mayoría son variaciones de dos temas básicos.

- Un enfoque consiste en mostrar en un valor (digamos, blanco) todos los valores en el rango de interés y en otro (digamos, negro) todas las demás intensidades. Esta transformación produce una imagen binaria.
- El segundo enfoque, aclara (u oscurece) el rango de intensidades deseado, pero deja todos los demás niveles de intensidad de la imagen sin cambios.



```
In [16]: def corte_por_niveles(image, r1, r2, k1 = 1, k2 = 0):
    """
    Realiza un corte por niveles de intensidad.

    Parametros:
    image (numpy.ndarray): La imagen a ajustar.
    r1 (float): El punto de inicio del corte.
    r2 (float): El punto final del corte.
    k1 (float): El valor de intensidad para lo que está dentro del rango.
    k2 (float): El valor de intensidad para lo que está fuera del rango.

    Regresa:
    numpy.ndarray: La imagen con el corte aplicado.
    """
    # Obtenemos las dimensiones de la imagen
    image_height, image_width = image.shape

    # Creamos una imagen de salida vacía
    output_image = np.zeros_like(image)

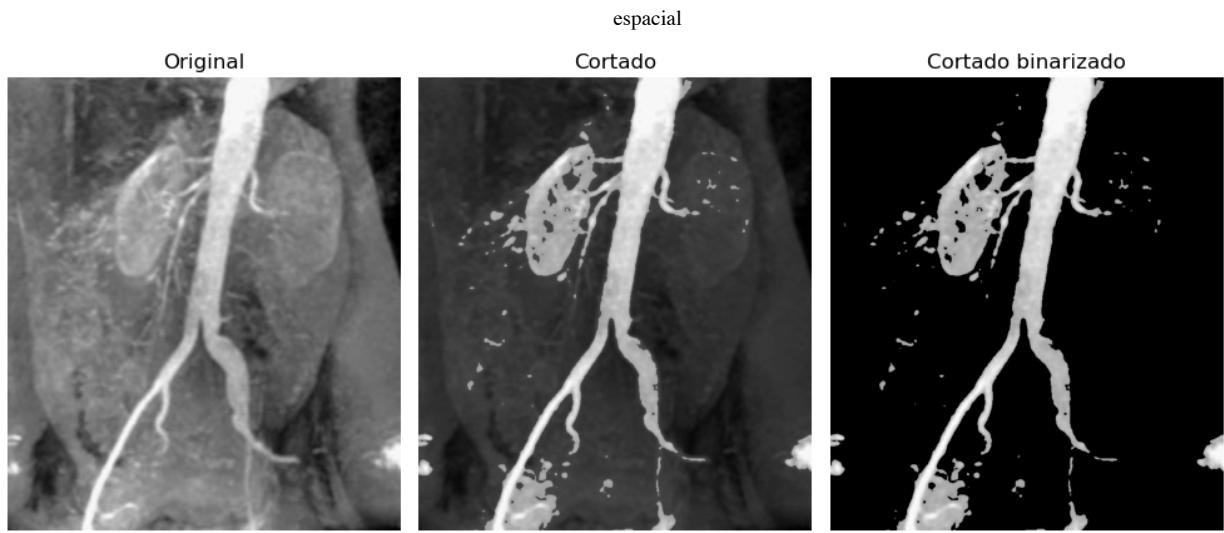
    # Ajustamos el contraste
    for y in range(image_height):
        for x in range(image_width):
            if r1 <= image[y, x] <= r2:
                val = k1 * image[y, x]
                output_image[y, x] = val if val <= 255 else 255
            else:
                output_image[y, x] = k2 * image[y, x]

    return output_image
```

```
In [17]: angiograma = imread('figs/angiograma.png', as_gray=True)
angiograma = angiograma * 255 // 1

angiograma_cortado_bin = corte_por_niveles(angiograma, 150, 255)
angiograma_cortado = corte_por_niveles(angiograma, 150, 255, 1, 0.5)

show_images(angiograma, angiograma_cortado, angiograma_cortado_bin,
            titles=['Original', 'Cortado', 'Cortado binarizado'],
            cols=3)
```

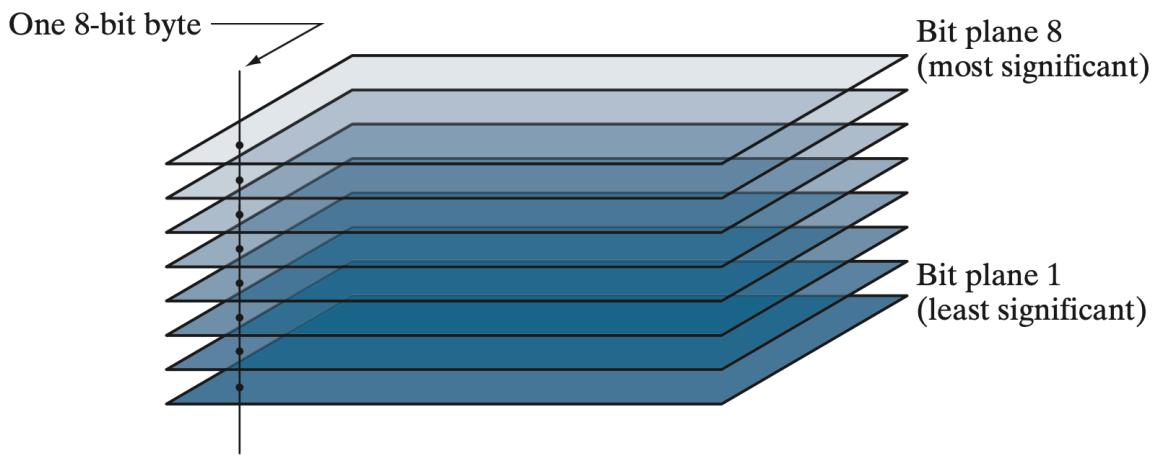


Corte de plano de bits

Los valores de píxeles son números enteros compuestos de bits. Por ejemplo, los valores de una imagen en escala de grises de 256 niveles se componen de 8 bits (un byte).

En lugar de resaltar los rangos de niveles de intensidad, podríamos resaltar la contribución realizada por bits específicos a la apariencia total de la imagen. Como ilustra la figura, se puede considerar que una imagen de 8 bits está compuesta por ocho planos de un bit:

- el plano 1 contiene el bit de orden más bajo de todos los píxeles de la imagen y
- el plano 8 contiene todos los bits de orden más alto.



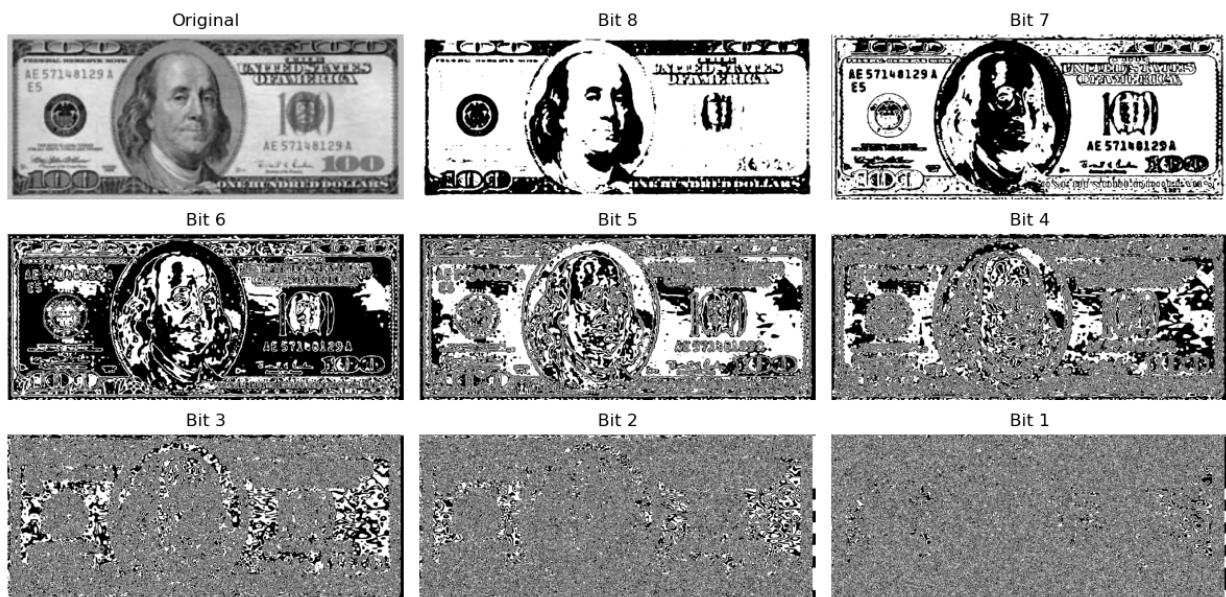
A continuación se muestra una imagen en escala de grises de 8 bits y sus ocho planos de un bit, correspondiendo.

- Observe que los cuatro planos de bits de orden superior, especialmente los dos primeros, contienen una cantidad significativa de datos visualmente significativos.
- Los planos de orden inferior contribuyen a detalles de intensidad más sutiles en la imagen.

```
In [18]: dollar = imread('figs/dollar.png', as_gray=True)
dollar = dollar * 255 // 1
dollar = dollar.astype(np.uint8)

dollar8 = dollar & 0x80
dollar7 = dollar & 0x40
dollar6 = dollar & 0x20
dollar5 = dollar & 0x10
dollar4 = dollar & 0x08
dollar3 = dollar & 0x04
dollar2 = dollar & 0x02
dollar1 = dollar & 0x01
```

```
In [19]: show_images(dollar, dollar8, dollar7, dollar6, dollar5, dollar4, dollar3, dollar2, dollar1,
titles=['Original', 'Bit 8', 'Bit 7', 'Bit 6', 'Bit 5', 'Bit 4', 'Bit 3', 'Bit 2', 'Bit 1'],
cols=3,
figsize=(12, 6))
```



Descomponer una imagen en sus planos de bits es útil para analizar la importancia relativa de cada bit en la imagen, un proceso que ayuda a determinar la idoneidad del número de bits utilizados para cuantificar la imagen.

Además, este tipo de descomposición es útil para la compresión de imágenes, en la que se utilizan menos de todos los planos para reconstruir una imagen.

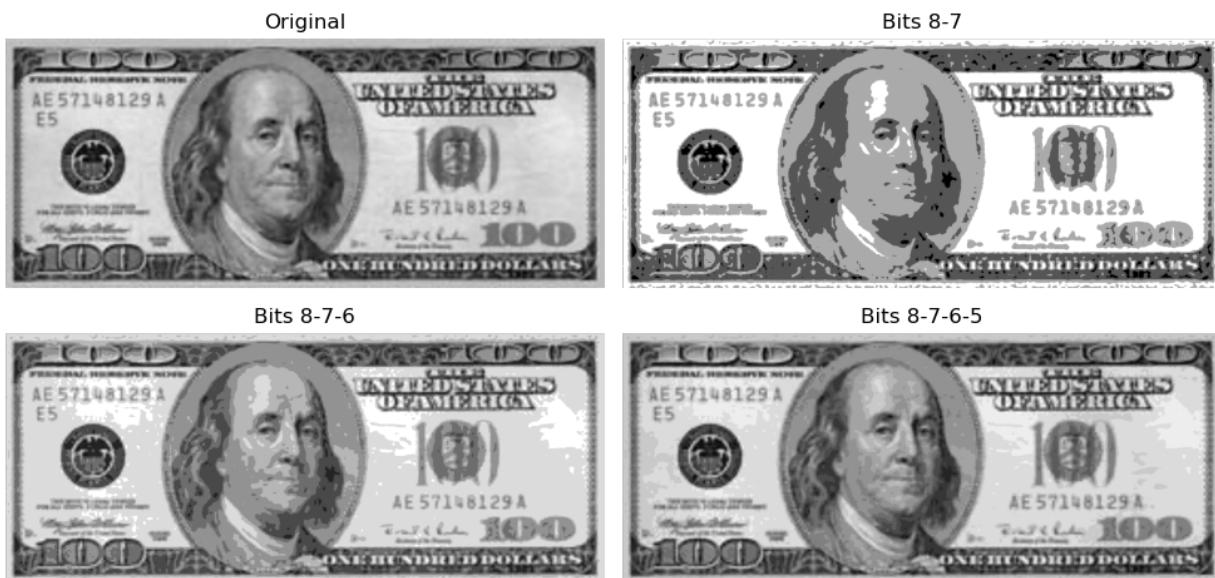
Por ejemplo, a continuación se muestra una imagen reconstruida utilizando los planos de bits 8-7, 8-7-6 y 8-7-6-5 de la descomposición anterior.

- Aunque se restauraron las características principales de la imagen original, la imagen reconstruida parece plana, especialmente en el fondo. Esto no es sorprendente, porque dos planos sólo pueden producir cuatro niveles de intensidad distintos.
- Agregar el plano 6 a la reconstrucción ayudó a mejorar la situación. Tenga en cuenta que el fondo de esta imagen tiene un contorno falso perceptible.

- Este efecto se reduce significativamente al agregar el quinto plano a la reconstrucción. Usar más planos en la reconstrucción no contribuiría significativamente a la apariencia de esta imagen.
- En este ejemplo, almacenar los cuatro planos de bits de orden más alto nos permitiría reconstruir la imagen original con un detalle aceptable. Almacenar estos cuatro planos en lugar de la imagen original requiere un 50% menos de almacenamiento.

```
In [20]: dollar87 = dollar8 | dollar7
dollar876 = dollar87 | dollar6
dollar8765 = dollar876 | dollar5

show_images(dollar, dollar87, dollar876, dollar8765,
            titles=['Original', 'Bits 8-7', 'Bits 8-7-6', 'Bits 8-7-6-5'],
            cols=2,
            figsize=(10, 5))
```



3.3. Procesamiento de histogramas

Sea r_k , para $k = 0, 1, 2, \dots, L - 1$, las intensidades de una imagen digital de L niveles, $f(x, y)$. El histograma no normalizado de f se define como:

$$h(r_k) = n_k, k = 1, 2, \dots, L - 1$$

donde n_k es el número de píxeles en f con intensidad r_k y las subdivisiones de la escala de intensidad se denominan contenedores de histograma.

De manera similar, el histograma normalizado de f se define como:

$$p(r_k) = \frac{h(r_k)}{MN} = \frac{n_k}{MN}$$

donde, como es habitual, M y N son el número de filas y columnas de la imagen, respectivamente.

- Principalmente, trabajamos con histogramas normalizados, a los que nos referimos simplemente como histogramas o histogramas de imágenes.
- La suma de $p(r_k)$ para todos los valores de k es siempre 1.
- Los componentes de $p(r_k)$ son estimaciones de las probabilidades de que se produzcan niveles de intensidad en una imagen.

In [21]:

```
def histograma(image):
    """
    Calcula el histograma de una imagen.

    Parametros:
    image (numpy.ndarray): La imagen a calcular el histograma.

    Regresa:
    numpy.ndarray: El histograma de la imagen.
    """

    # Obtenemos las dimensiones de la imagen
    image_height, image_width = image.shape

    # Creamos un arreglo para el histograma
    hist = np.zeros(256)

    # Calculamos el histograma
    for y in range(image_height):
        for x in range(image_width):
            hist[int(image[y, x])] += 1

    for i in range(256):
        hist[i] /= image_height * image_width

    return hist
```

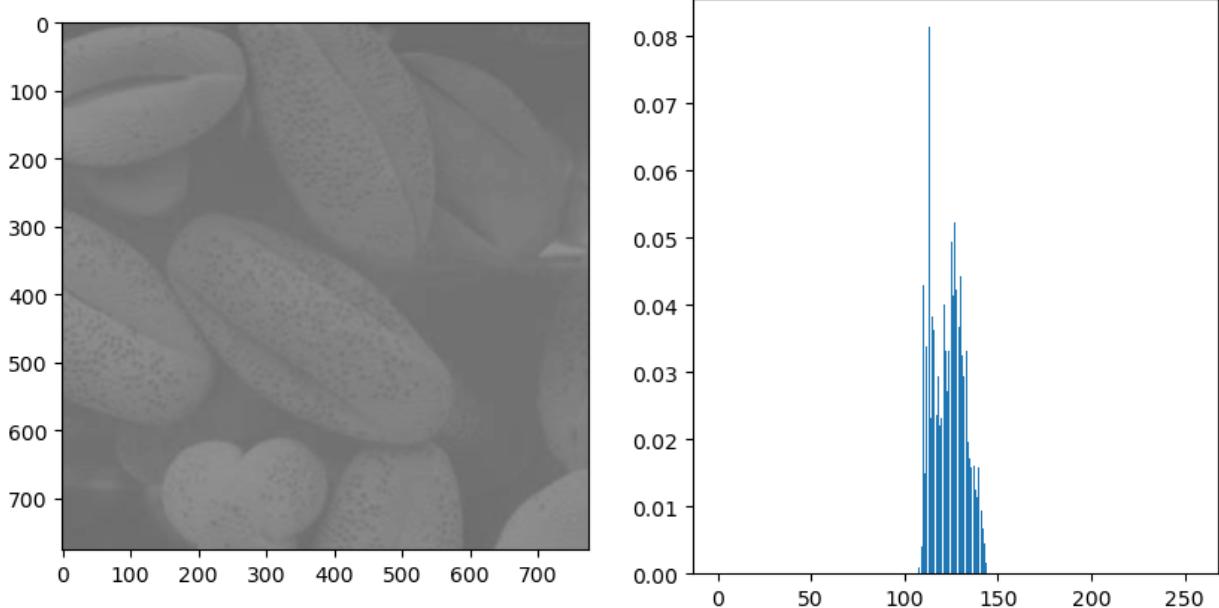
In [22]:

```
granos_hist = histograma(granos)

plt.subplots(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(granos, cmap='gray')
plt.subplot(1, 2, 2)
plt.bar(np.arange(256), granos_hist)
```

Out[22]:

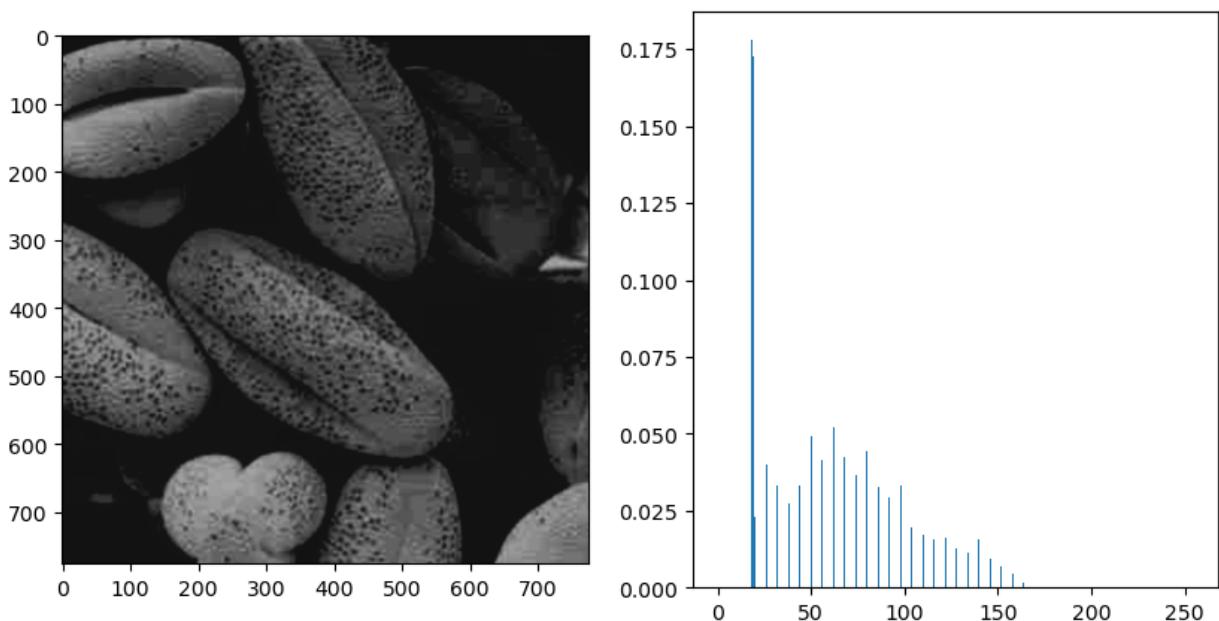
<BarContainer object of 256 artists>



```
In [23]: granos_ajustada_hist = histograma(granos_ajustada)

plt.subplots(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(granos_ajustada, cmap='gray')
plt.subplot(1, 2, 2)
plt.bar(np.arange(256), granos_ajustada_hist)
```

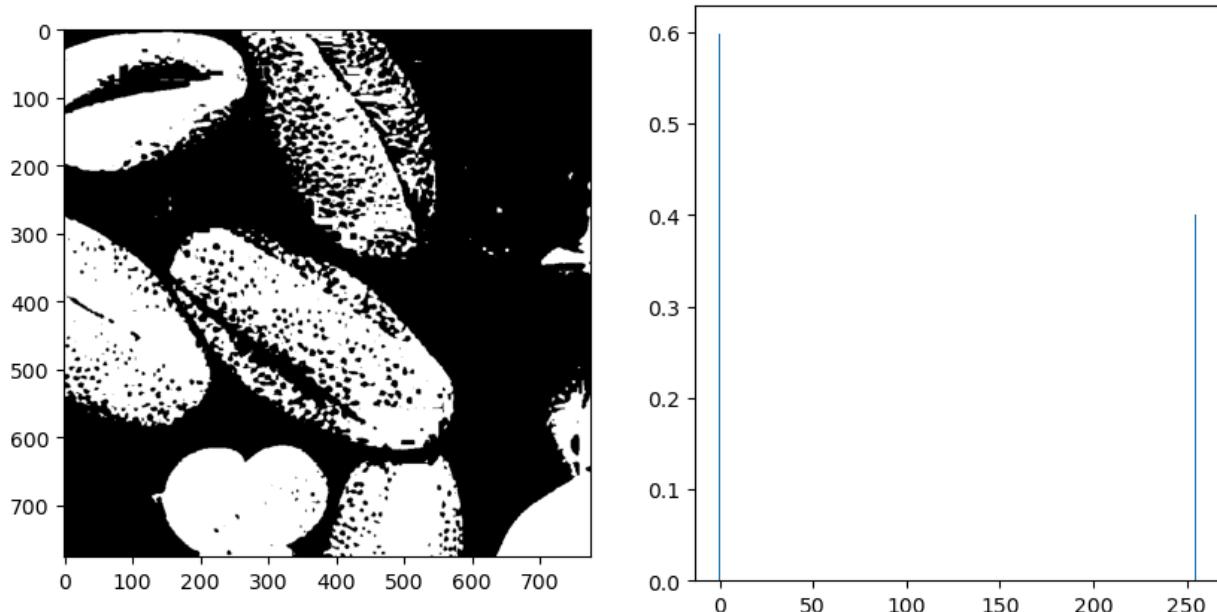
Out[23]: <BarContainer object of 256 artists>



```
In [24]: granos_binarizada_hist = histograma(granos_binarizada)

plt.subplots(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(granos_binarizada, cmap='gray')
plt.subplot(1, 2, 2)
plt.bar(np.arange(256), granos_binarizada_hist)
```

Out[24]: <BarContainer object of 256 artists>



3.3.1. Ecualización del histograma

Suponiendo valores de intensidad inicialmente continuos, sea la variable r la que denota las intensidades de una imagen a procesar.

Como es habitual, suponemos que r está en el rango $[0, L - 1]$, donde $r = 0$ representa el negro y $r = L - 1$ representa el blanco.

Para que r satisfaga estas condiciones, centramos la atención en transformaciones (mapeos de intensidad) de la forma:

$$s = T(r), 0 \leq r \leq N - 1$$

que producen un valor de intensidad de salida s , para un valor de intensidad dado r en la imagen de entrada. Asumimos que

1. $T(r)$ es una función monótona creciente en el intervalo $0 \leq r \leq L - 1$; y
2. $0 \leq T(r) \leq L - 1$ para $0 \leq r \leq L - 1$.

NOTA: Una función $T(r)$ es una función monótona creciente si $T(r_1) \geq T(r_2)$ para $r_1 > r_2$. $T(r)$ es una función creciente estrictamente monótona si $T(r_1) > T(r_2)$ para $r_1 > r_2$. Se aplican definiciones similares a una función decreciente monótona.

En algunas formulaciones que se analizarán en breve, utilizamos la transformación inversa:

$$r = T^{-1}(s), 0 \leq s \leq L - 1$$

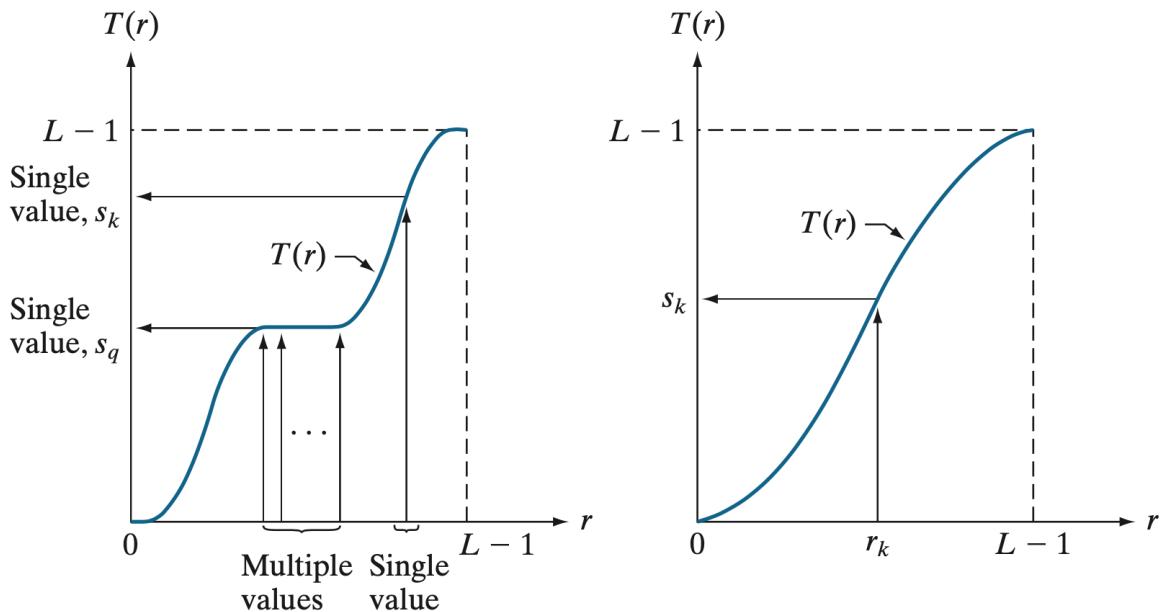
en cuyo caso cambiamos la condición (a) a:

1. $T(r)$ es una función creciente estrictamente monótona en el intervalo $0 \leq r \leq L - 1$.

La condición en (1) de que $T(r)$ aumente monótonamente garantiza que los valores de intensidad de salida nunca serán menores que los valores de entrada correspondientes, evitando así artefactos creados por inversiones de intensidad.

La condición (2) garantiza que el rango de intensidades de salida es el mismo que el de entrada.

Finalmente, la condición (a) garantiza que las asignaciones desde s hasta r serán uno a uno, evitando así ambigüedades.



La intensidad de una imagen puede verse como una variable aleatoria en el intervalo $[0, L - 1]$. Sean $p_r(r)$ y $p_s(s)$ las funciones de distribución de probabilidad (PDF - *probability distribution function*) de los valores de intensidad r y s en dos imágenes diferentes. Los subíndices de p indican que p_r y p_s son funciones diferentes.

Un resultado fundamental de la teoría de la probabilidad es que si $p_r(r)$ y $T(r)$ son conocidos, y $T(r)$ es continua y diferenciable en el rango de valores de interés, entonces la distribución de probabilidad de la variable transformada (mapeada) s puede ser obtenido como:

$$p_s(s) = p_r(r) \left| \frac{ds}{dr} \right|$$

Así, vemos que la PDF de la variable de intensidad de salida, s , está determinada por la PDF de las intensidades de entrada y la función de transformación utilizada [recordemos que r y s están relacionados por $T(r)$]. Una función de transformación de particular importancia en el procesamiento de imágenes es

$$s = T(r) = (L - 1) \int_0^r p_r(w) dw$$

donde w es una variable ficticia de integración. La integral del lado derecho es la función de distribución acumulativa (CDF - *cumulative distribution function*) de la variable aleatoria r .

Debido a que las PDF siempre son positivas y la integral de una función es el área bajo la función, el área bajo la función no puede disminuir a medida que r aumenta.

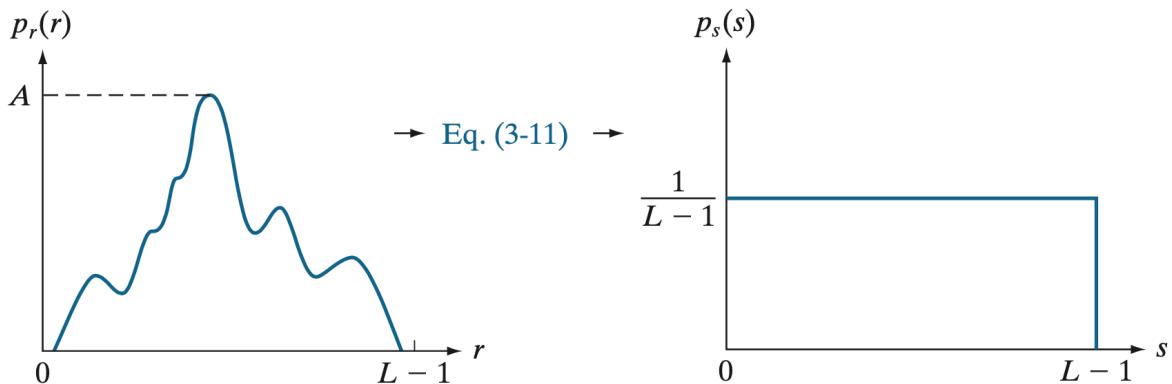
Cuando el límite superior de esta ecuación es $r = (L - 1)$, la integral se evalúa como 1, como debe ser para una PDF.

Sabemos por la regla de Leibniz en cálculo que la derivada de una integral definida con respecto a su límite superior es el integrando evaluado en el límite. Eso es:

$$\frac{ds}{dr} = \frac{dT(r)}{dr} = (L - 1) \frac{d}{dr} \left[\int_0^r p_r(w) dw \right] = (L - 1)p_r(r)$$

Sustituyendo este resultado por dr/ds , y observando que todos los valores de probabilidad son positivos, se obtiene

$$p_s(s) = p_r(r) \left| \frac{dr}{ds} \right| = p_r(r) \left| \frac{1}{(L - 1)p_r(r)} \right| = \frac{1}{L - 1}$$



Para valores discretos, trabajamos con probabilidades y sumatorias en lugar de funciones de densidad de probabilidad e integrales.

Recuerde que la probabilidad de ocurrencia del nivel de intensidad r_k en una imagen digital se aproxima por:

$$p_r(r_k) = \frac{n_k}{MN}$$

donde M y N son el número total de píxeles en la imagen, y n_k denota el número de píxeles que tienen intensidad r_k .

La transformación en su forma discreta será:

$$s_k = T(r_k) = (L - 1) \sum_{j=0}^k p_r(r_j), k = 1, 2, \dots, L - 1$$

donde, L es el número de niveles de intensidad posibles en la imagen (por ejemplo, 256 para una imagen de 8 bits).

Se obtiene una imagen procesada (de salida) utilizando la ecuación anterior para asignar cada píxel de la imagen de entrada con intensidad r_k a un píxel correspondiente con nivel s_k en la imagen de salida. Esto se denomina *ecualización de histograma*.

```
In [25]: def ecualizar_histograma(image):
    """
    Ecualiza el histograma de una imagen.

    Parámetros:
    image (numpy.ndarray): La imagen a ecualizar.

    Regresa:
    numpy.ndarray: La imagen con el histograma ecualizado.
    """
    # Obtenemos las dimensiones de la imagen
    image_height, image_width = image.shape

    # Calculamos el histograma
    hist = histograma(image)

    # Calculamos la función de distribución acumulada
    cdf = np.zeros(256)
    cdf[0] = hist[0]
    for i in range(1, 256):
        cdf[i] = cdf[i - 1] + hist[i]

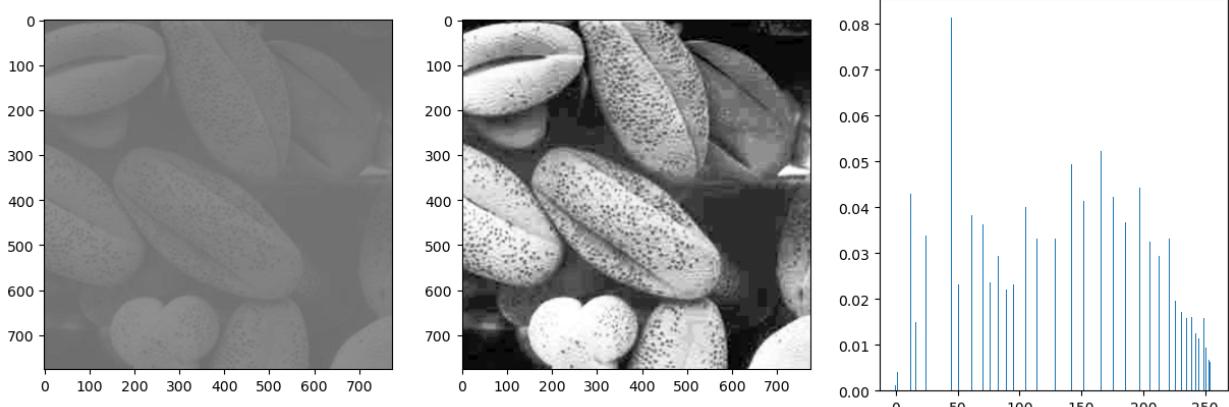
    # Calculamos la ecualización
    ecualizada = np.zeros_like(image)
    for y in range(image_height):
        for x in range(image_width):
            ecualizada[y, x] = cdf[image[y, x]] * 255

    return ecualizada
```

```
In [26]: def plot_images_hist(imagen, ecualizada, histograma):
    plt.subplots(figsize=(15, 5))
    plt.subplot(1, 3, 1)
    plt.imshow(imagen, cmap='gray')
    plt.subplot(1, 3, 2)
    plt.imshow(ecualizada, cmap='gray')
    plt.subplot(1, 3, 3)
    plt.bar(np.arange(256), histograma)
```

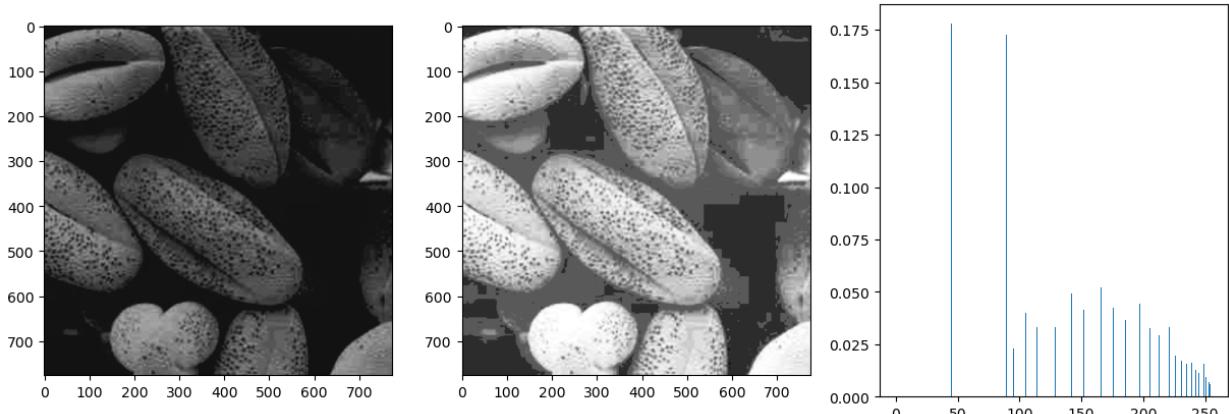
```
In [27]: # granos = granos * 255 // 1
granos = granos.astype(np.uint8)
granos_ecualizada = ecualizar_histograma(granos)
granos_ecualizada_hist = histograma(granos_ecualizada)
```

```
plot_images_hist(granos, granos_ecualizada, granos_ecualizada_hist)
```



```
In [28]: granos_ajustada = ajustar_contraste_por_partes(granos, r1, r2, s1, s2)
granos_ajustada = granos_ajustada.astype(np.uint8)
granos_ajustada_ecualizada = ecualizar_histograma(granos_ajustada)
granos_ajustada_ecualizada_hist = histograma(granos_ajustada_ecualizada)
```

```
plot_images_hist(granos_ajustada, granos_ajustada_ecualizada, granos_ajustada_ecualizada_hist)
```



3.4. Fundamentos del filtrado espacial

El filtrado espacial se utiliza en un amplio espectro de aplicaciones de procesamiento de imágenes, por lo que es importante tener una comprensión sólida de los principios de filtrado.

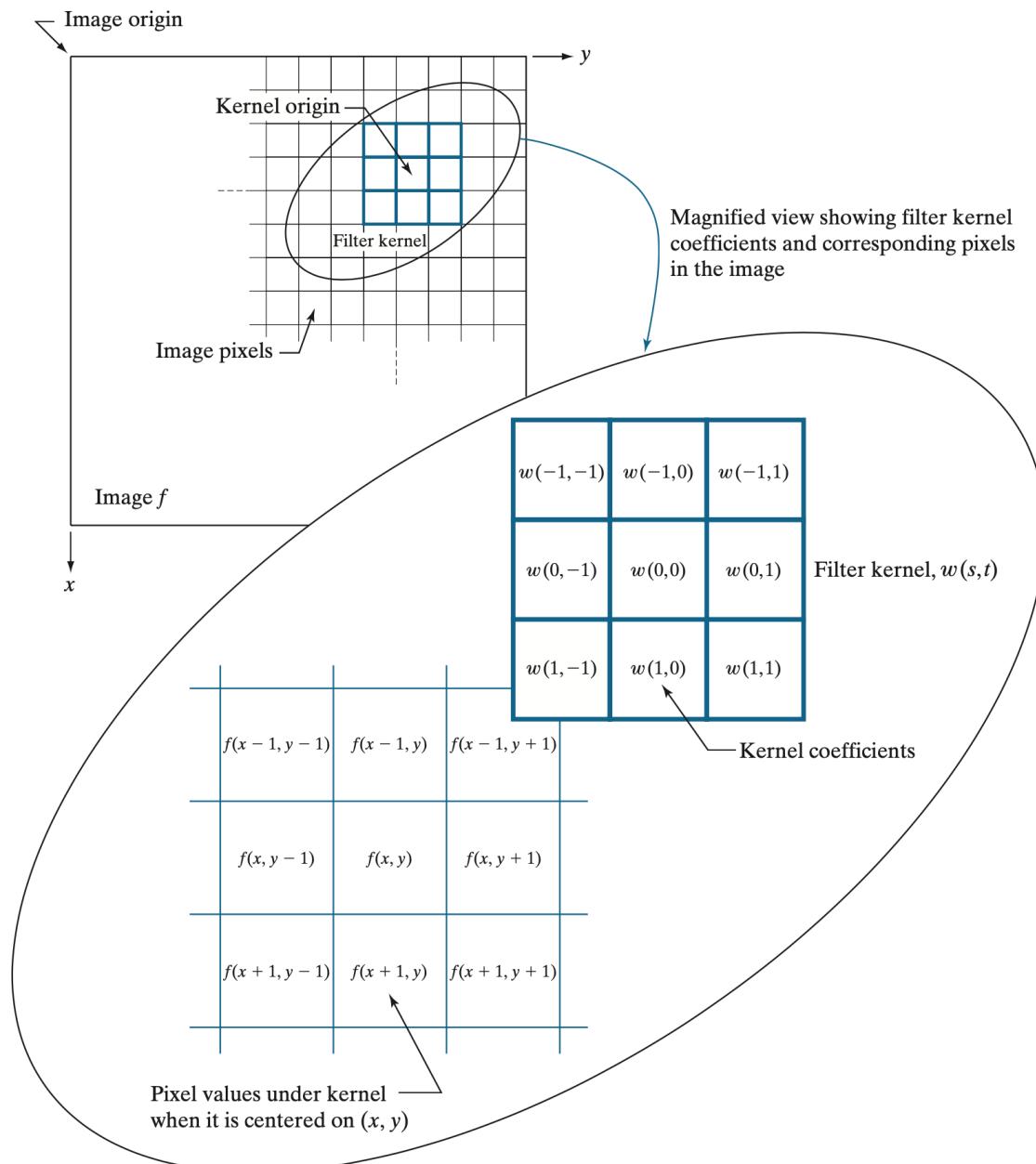
El nombre filtro se toma prestado del procesamiento en el dominio de la frecuencia, donde "filtrado" se refiere a pasar, modificar o rechazar componentes de frecuencia específicos de una imagen.

- Por ejemplo, un filtro que pasa bajas frecuencias se llama filtro de paso-bajo.
- El efecto neto producido por un filtro de paso bajo es suavizar una imagen difuminándola.
- Podemos lograr un suavizado similar directamente en la imagen mediante el uso de filtros espaciales.

3.4.1. La mecánica del filtrado espacial

Un filtro espacial lineal realiza una operación de suma de productos entre una imagen f y un núcleo de filtro w .

- El núcleo es una matriz cuyo tamaño define la vecindad de operación y cuyos coeficientes determinan la naturaleza del filtro.
- Otros términos utilizados para referirse a un núcleo de filtro espacial son máscara, plantilla y ventana.
- Usamos el término kernel de filtro o simplemente kernel.



En cualquier punto (x, y) de la imagen, la respuesta, $g(x, y)$, del filtro es la suma de los productos de los coeficientes del núcleo y los píxeles de la imagen abarcados por el núcleo:

$$g(x, y) = w(-1, -1)f(x - 1, y - 1) + w(-1, 0)f(x - 1, y) + \cdots + w(0, 0)f(x, y) + \cdots +$$

Observe que el coeficiente central del núcleo, $w(0, 0)$, se alinea con el píxel en la ubicación (x, y) .

Para un núcleo de tamaño $m \times n$, asumimos que $m = 2a + 1$ y $n = 2b + 1$, donde a y b son números enteros no negativos.

- Esto significa que nos centramos en núcleos de tamaño impar en ambas direcciones de coordenadas.

En general, el filtrado espacial lineal de una imagen de tamaño $M \times N$ con un núcleo de tamaño $m \times n$ viene dado por la expresión:

$$g(x, y) = \sum_{s=-a}^a \sum_{t=-b}^b w(s, t)f(x + s, y + t)$$

donde x e y varían de modo que el centro (origen) del núcleo visita cada píxel en f una vez.

Para un valor fijo de (x, y) , se define implementa la suma de productos de la forma que se muestra en la ecuación, pero para un núcleo de tamaño impar arbitrario.

3.4.2. Correlación y convolución espacial

La correlación espacial se ilustró gráficamente en la figura previa y se describe matemáticamente mediante la ecuación anterior. La correlación consiste en mover el centro de un núcleo sobre una imagen y calcular la suma de productos en cada ubicación.

La mecánica de la convolución espacial es la misma, excepto que el núcleo de correlación se gira 180°.

Así, cuando los valores de un núcleo son simétricos respecto de su centro, la correlación y la convolución producen el mismo resultado.

La correlación de un núcleo w de tamaño $m \times n$ con una imagen $f(x, y)$, denotada como $(w \star f)(x, y)$, viene dada por la ecuación:

$$(w \star f)(x, y) = \sum_{s=-a}^a \sum_{t=-b}^b w(s, t)f(x + s, y + t)$$

De manera similar, la convolución de un núcleo w de tamaño $m \times n$ con una imagen $f(x, y)$, denotada por $(w \star\!\! \star f)(x, y)$, se define como:

$$(w \star\!\! \star f)(x, y) = \sum_{s=-a}^a \sum_{t=-b}^b w(s, t)f(x - s, y - t)$$

donde los signos menos alinean las coordenadas de f y w cuando una de las funciones se gira 180°.

Debido a que la convolución es conmutativa, es irrelevante si se rota w o f , pero la rotación del núcleo se usa por convención.

Propiedad	Convolución	Correlación
Comutativa	$f \star g = g \star f$	-
Asociativa	$f \star (g \star h) = (f \star g) \star h$	-
Distributiva	$f \star (g + h) = (f \star g) + (f \star h)$	$f \star (g + h) = (f \star g) + (f \star h)$

Nuestros núcleos no dependen de (x, y) , un hecho que a veces hacemos explícito escribiendo el lado izquierdo de la ecuación como $w \star f(x, y)$.

A veces, una imagen se filtra (es decir, se convoluciona) secuencialmente, en etapas, utilizando un núcleo diferente en cada etapa.

- Por ejemplo, supongamos que una imagen f se filtra con un kernel w_1 , el resultado se filtra con un kernel w_2 , ese resultado se filtra con un tercer kernel, y así sucesivamente, para Q etapas.
- Debido a la propiedad comutativa de la convolución, este filtrado de varias etapas se puede realizar en una única operación de filtrado, $w \star f$, donde:

$$w = w_1 \star w_2 \star w_3 \star \dots \star w_Q$$

- El tamaño de w se obtiene a partir de los tamaños de los núcleos individuales mediante aplicaciones sucesivas de estos.
- Si todos los núcleos individuales son de tamaño $m \times n$, de estas ecuaciones se deduce que w será de tamaño $W_v \times W_h$, donde:

$$W_v = Q(m - 1) + m$$

$$W_h = Q(n - 1) + n$$

3.4.3. Núcleos separables

Se considera que una función 2-D $G(x, y)$ es separable si se puede escribir como el producto de dos funciones 1-D, $G_1(x)$ y $G_2(y)$; es decir, $G(x, y) = G_1(x)G_2(y)$.

Un núcleo de filtro espacial es una matriz, y un núcleo separable es una matriz que puede ser expresada como el producto exterior de dos vectores.

Por ejemplo, el núcleo 2x3:

$$w = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

es separable ya que puede ser expresada como el producto externo de los vectores:

$$\mathbf{c} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \text{ y } \mathbf{r} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

Esto es:

$$\mathbf{c}\mathbf{r}^T = \begin{bmatrix} 1 \\ 1 \end{bmatrix} [1 \ 1 \ 1] = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} = w$$

Un núcleo separable de tamaño $m \times n$ se puede expresar como el producto exterior de dos vectores, \mathbf{v} y \mathbf{w} :

$$w = \mathbf{v}\mathbf{w}^T$$

donde \mathbf{v} y \mathbf{w} son vectores de tamaño $m \times 1$ y $n \times 1$, respectivamente. Para un grano cuadrado de tamaño $m \times m$, escribimos:

$$w = \mathbf{v}\mathbf{v}^T$$

La importancia de los núcleos separables radica en las ventajas computacionales que resultan de la propiedad asociativa de la convolución.

Si tenemos un núcleo w que se puede descomponer en dos núcleos más simples, tales que $w = w_1 \star w_2$, entonces de las propiedades commutativas y asociativas se deduce que:

$$w \star f = (w_1 \star w_2) \star f = (w_2 \star w_1) \star f = w_2 \star (w_1 \star f) = (w_1 \star f) \star w_2$$

3.4.4. Sobre cómo se construyen los núcleos

Consideramos tres enfoques básicos para construir filtros espaciales.

- Un enfoque se basa en formular filtros basados en propiedades matemáticas.
 - Por ejemplo, un filtro que calcula el promedio de píxeles en un vecindario desenfoca una imagen. Calcular un promedio es análogo a la integración.
- Un segundo enfoque se basa en muestrear una función espacial bidimensional cuya forma tiene una propiedad deseada.

- Por ejemplo, se pueden usar muestras de una función gaussiana para construir un filtro de promedio ponderado (paso bajo).
- Un tercer enfoque consiste en diseñar un filtro espacial con una respuesta de frecuencia específica. Ilustraremos estas técnicas en la Sección 3.7.

3.5. Filtros espaciales de suavizado (pasa bajo)

Los filtros espaciales de suavizado (también llamados promedio) se utilizan para reducir las transiciones bruscas de intensidad.

- Debido a que el ruido aleatorio normalmente consiste en transiciones bruscas de intensidad, una aplicación obvia del suavizado es la reducción de ruido.
- El suavizado se utiliza para reducir los detalles irrelevantes en una imagen, donde "irrelevante" se refiere a regiones de píxeles que son pequeñas con respecto al tamaño del núcleo del filtro.
- Otra aplicación es suavizar los contornos falsos que resultan del uso de un número insuficiente de niveles de intensidad en una imagen.

Los filtros de suavizado se utilizan en combinación con otras técnicas para mejorar la imagen, como las técnicas de procesamiento de histogramas analizadas previamente, y el enmascaramiento de enfoque.

Como comentamos, el filtrado espacial lineal consiste en convolucionar una imagen con un núcleo de filtro.

- La convolución de un núcleo de suavizado con una imagen desenfoca la imagen, y el grado de desenfoque está determinado por el tamaño del núcleo y los valores de sus coeficientes.
- Además de ser útiles en innumerables aplicaciones de procesamiento de imágenes, los filtros de paso bajo son fundamentales, en el sentido de que otros filtros importantes, incluidos los filtros de nitidez (paso alto), paso de banda y rechazo de banda, pueden derivarse de filtros de paso bajo.

3.5.1. Filtros de caja

El núcleo de filtro de paso bajo separable más simple es el núcleo de caja, cuyos coeficientes tienen el mismo valor (normalmente 1).

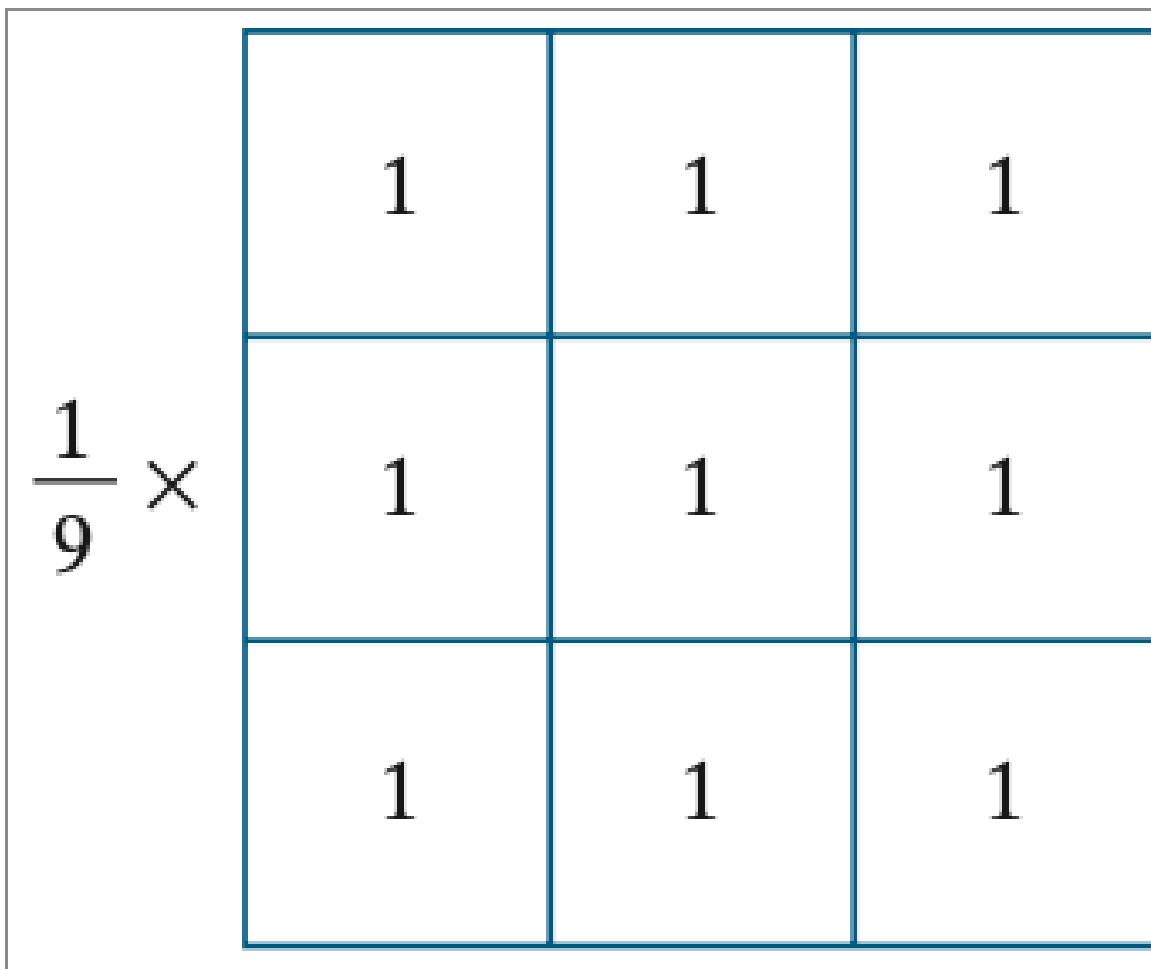
El nombre "filtro de caja" proviene de un núcleo constante que se asemeja a una caja cuando se ve en 3-D.

- Un filtro de caja $m \times n$ es una matriz $m \times n$ de unos, con una constante de normalización delante, cuyo valor es 1 dividido por la suma de los valores de los

coeficientes (es decir, $\frac{1}{mn}$ cuando todos los coeficientes son unos).

- Esta normalización tiene dos propósitos:

- En primer lugar, el valor medio de un área de intensidad constante sería igual a esa intensidad en la imagen filtrada, como debería ser.
- En segundo lugar, normalizar el núcleo de esta manera evita introducir un sesgo durante el filtrado; es decir, la suma de los píxeles de las imágenes original y filtrada será la misma.



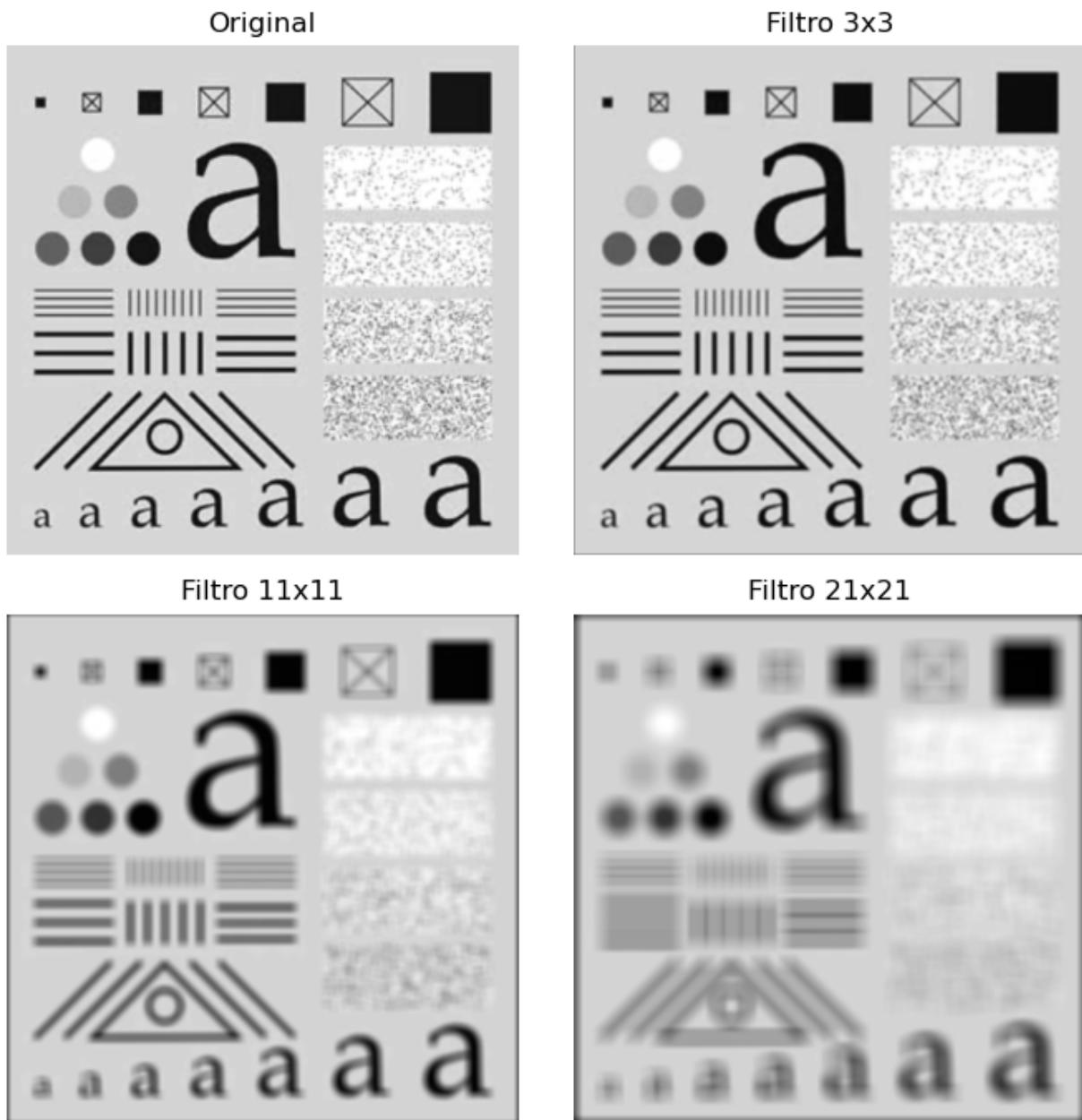
```
In [31]: test_pattern = imread('figs/test_pattern.png', as_gray=True)
test_pattern = test_pattern * 255 // 1
test_pattern = test_pattern.astype(np.uint8)
```

```
In [45]: box_kernel_lowpass_3 = np.ones((3, 3)) / 9
box_kernel_lowpass_11 = np.ones((11, 11)) / 121
box_kernel_lowpass_21 = np.ones((21, 21)) / 441

test_pattern_lowpass_3 = aplicar_convolucion(test_pattern, box_kernel_lowpass_3)
test_pattern_lowpass_11 = aplicar_convolucion(test_pattern, box_kernel_lowpass_11)
test_pattern_lowpass_21 = aplicar_convolucion(test_pattern, box_kernel_lowpass_21)
```

In [46]:

```
show_images(test_pattern, test_pattern_lowpass_3, test_pattern_lowpass_11, tes-
titles=['Original', 'Filtro 3x3', 'Filtro 11x11', 'Filtro 21x21'],
cols=2,
figsize=(7, 7))
```



3.5.2. Filtro basa-bajo Gaussiano

Debido a su simplicidad, los filtros de caja son adecuados para una experimentación rápida y, a menudo, producen resultados de suavizado que son visualmente aceptables.

Sin embargo, los filtros de caja tienen limitaciones que los convierten en malas opciones en muchas aplicaciones.

- Por ejemplo, una lente desenfocada a menudo se modela como un filtro de paso bajo, pero los filtros de caja son malas aproximaciones a las características de desenfoque de las lentes.

- Otra limitación es el hecho de que los filtros de caja favorecen el desenfoque en direcciones perpendiculares. En aplicaciones que implican imágenes con un alto nivel de detalle o con fuertes componentes geométricos, la direccionalidad de los filtros de caja a menudo produce resultados indeseables.

Los núcleos elegidos en aplicaciones como las que acabamos de mencionar son circularmente simétricos (también llamados isotrópicos, lo que significa que su respuesta es independiente de la orientación).

Resulta que los núcleos gaussianos de la forma:

$$w(s, t) = G(s, t) = K e^{-\frac{s^2+t^2}{2\sigma^2}}$$

son los únicos núcleos circularmente simétricos que también son separables. Por lo que disfrutan de las mismas ventajas computacionales que los filtros de caja.

Sea $r = \sqrt{s^2 + t^2}$, entonces podemos reescribir la ecuación anterior como:

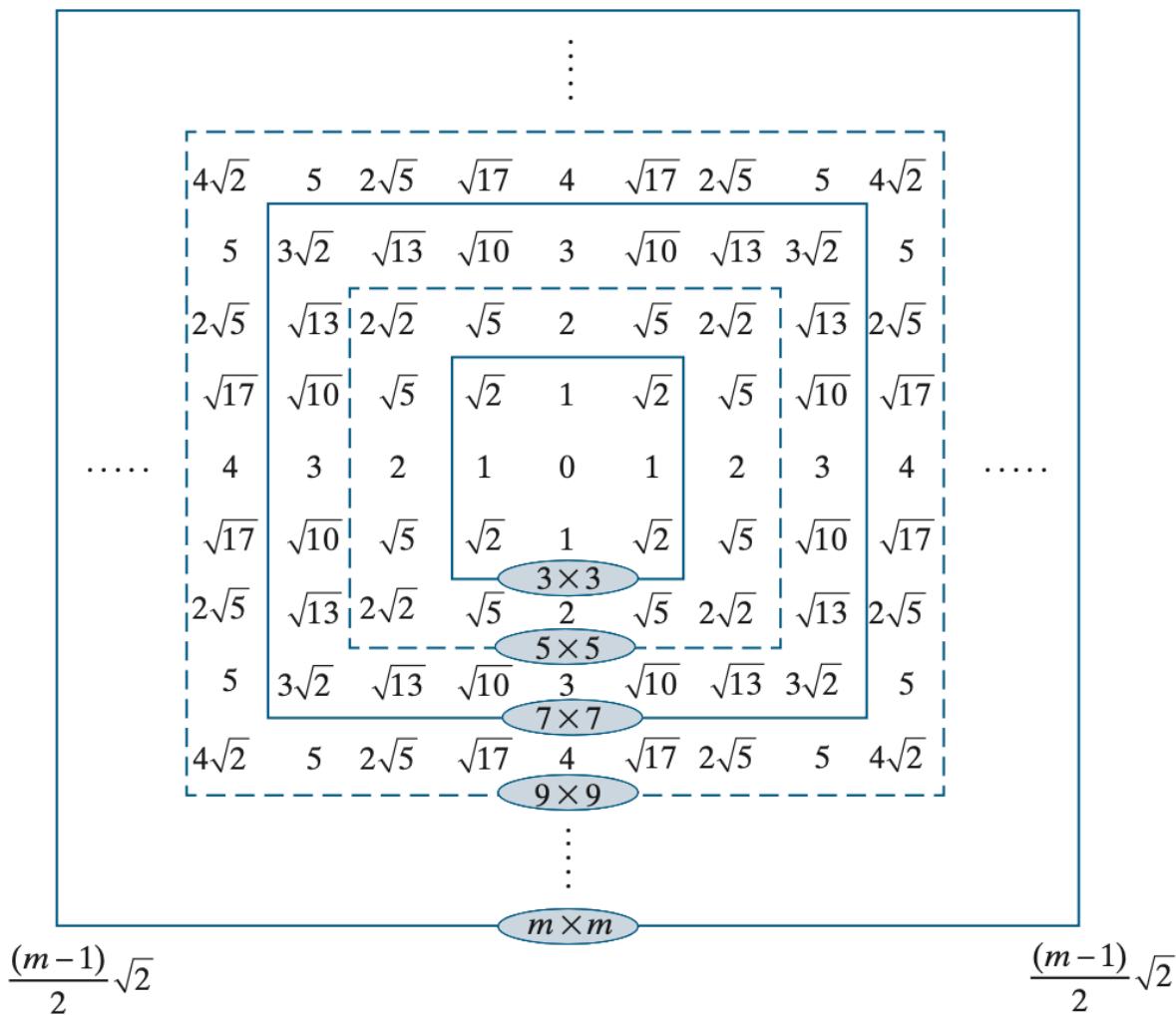
$$G(r) = K e^{-\frac{r^2}{2\sigma^2}}$$

La variable r es la distancia desde el centro hasta cualquier punto de la función G .

La figura muestra valores de r para varios tamaños de núcleo utilizando valores enteros para s y t .

$$\frac{(m-1)}{2} \sqrt{2}$$

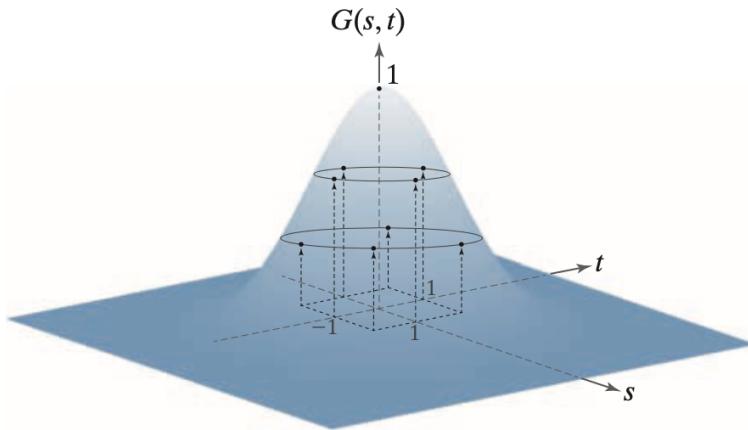
$$\frac{(m-1)}{2} \sqrt{2}$$



El núcleo de la figura se obtiene muestreando la ecuación anterior (con $K = 1$ y $s = 1$). Se muestra un gráfico en perspectiva de una función gaussiana e ilustra que las muestras utilizadas para generar ese núcleo se obtuvieron especificando valores de s y t , y luego "leyendo" los valores de la función en esas coordenadas.

Estos valores son los coeficientes del kernel. Normalizar el kernel dividiendo sus coeficientes por la suma de los coeficientes completa la especificación del kernel.

Debido a que los núcleos gaussianos son separables, simplemente podríamos tomar muestras a lo largo de una sección transversal que pasa por el centro y usar las muestras para formar el vector v



$$\frac{1}{4.8976} \times$$

0.3679	0.6065	0.3679
0.6065	1.0000	0.6065
0.3679	0.6065	0.3679

```
In [38]: # funcion para crear un nucleo de filtro pasabajos gaussiano
def crear_kernel_gaussiano(tam, sigma):
    """
    Crea un kernel gaussiano.

    Parametros:
    tam (int): El tamaño del kernel.
    sigma (float): El valor de sigma.

    Regresa:
    numpy.ndarray: El kernel gaussiano.
    """
    # Obtenemos el centro del kernel
    center = tam // 2

    # Creamos el kernel
    kernel = np.zeros((tam, tam))

    # Calculamos el kernel
    for y in range(tam):
        for x in range(tam):
            kernel[y, x] = np.exp(-((x - center)**2 + (y - center)**2) / (2 * sigma**2))

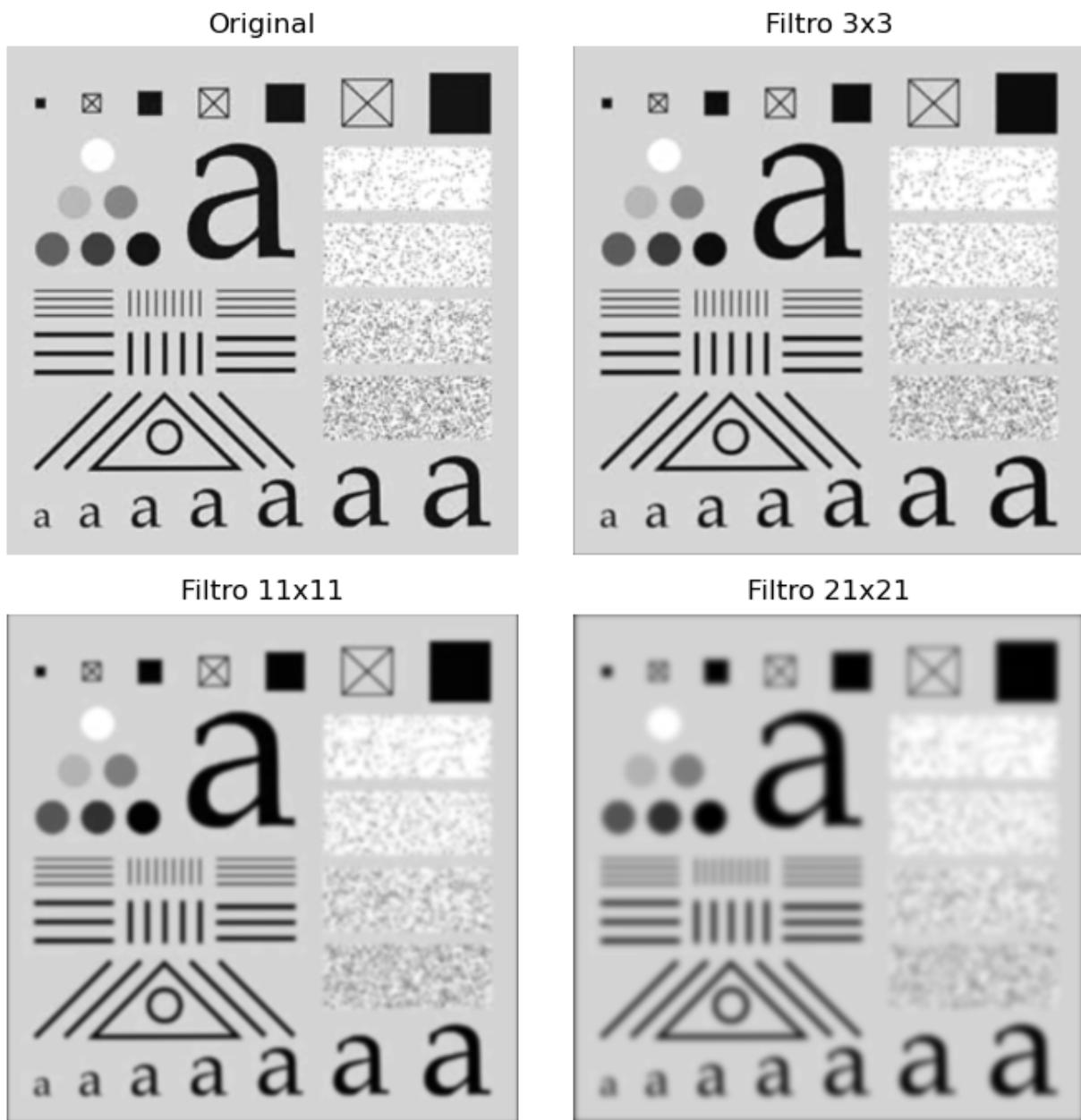
    # Normalizamos el kernel
    kernel = kernel / np.sum(kernel)

    return kernel
```

```
In [47]: gaussian_kernel_lowpass_3 = crear_kernel_gaussiano(3, 1)
gaussian_kernel_lowpass_11 = crear_kernel_gaussiano(11, 2)
gaussian_kernel_lowpass_21 = crear_kernel_gaussiano(21, 3.5)

test_pattern_lowpass_gauss3 = aplicar_convolucion(test_pattern, gaussian_kernel_lowpass_3)
test_pattern_lowpass_gauss11 = aplicar_convolucion(test_pattern, gaussian_kernel_lowpass_11)
test_pattern_lowpass_gauss21 = aplicar_convolucion(test_pattern, gaussian_kernel_lowpass_21)
```

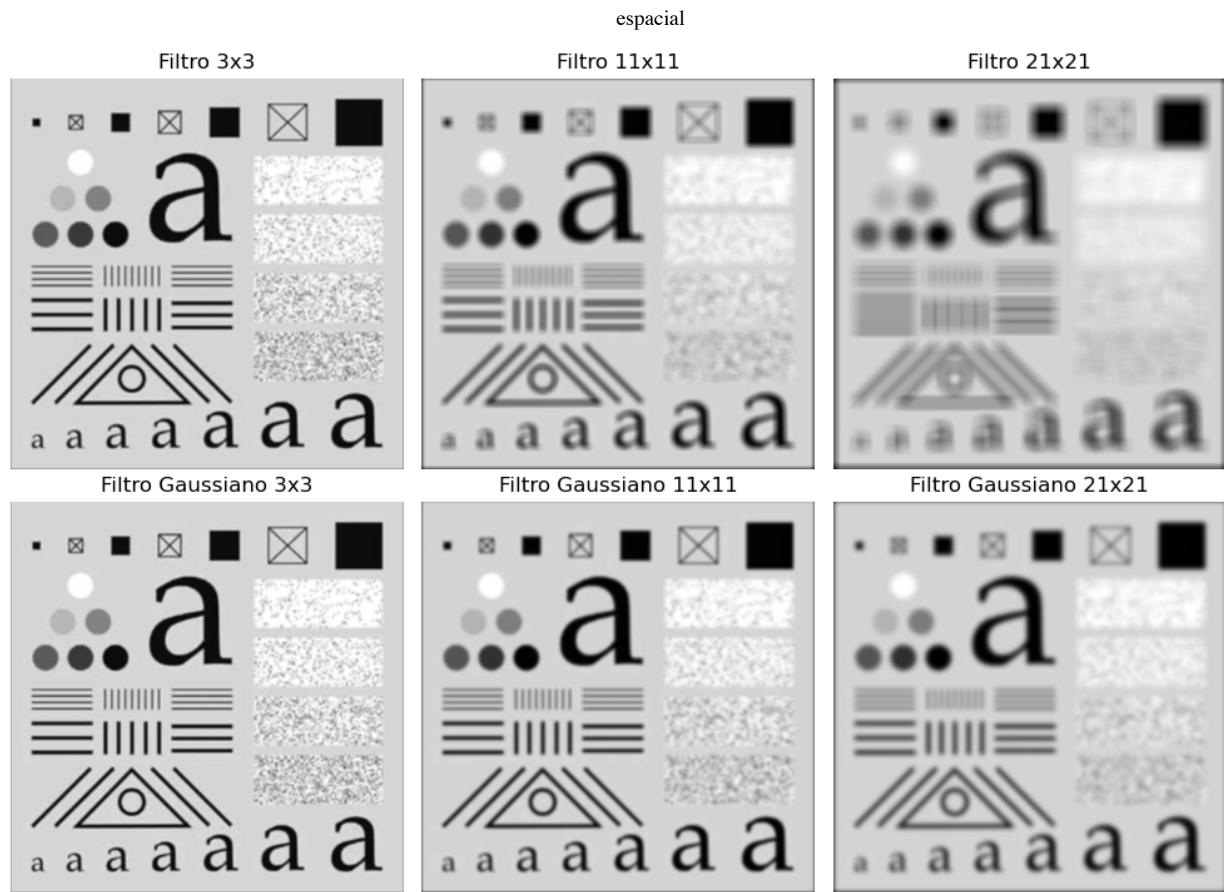
```
In [44]: show_images(test_pattern, test_pattern_lowpass_gauss3, test_pattern_lowpass_gauss11,
                  titles=['Original', 'Filtro 3x3', 'Filtro 11x11', 'Filtro 21x21'],
                  cols=2,
                  figsize=(7, 7))
```



Los resultados muestran poca diferencia visual en el desenfoque. A pesar de esto, existen algunas diferencias sutiles que no son evidentes a primera vista.

- Por ejemplo, compare la letra grande "a" en las siguientes figuras, la última es mucho más suave en los bordes.

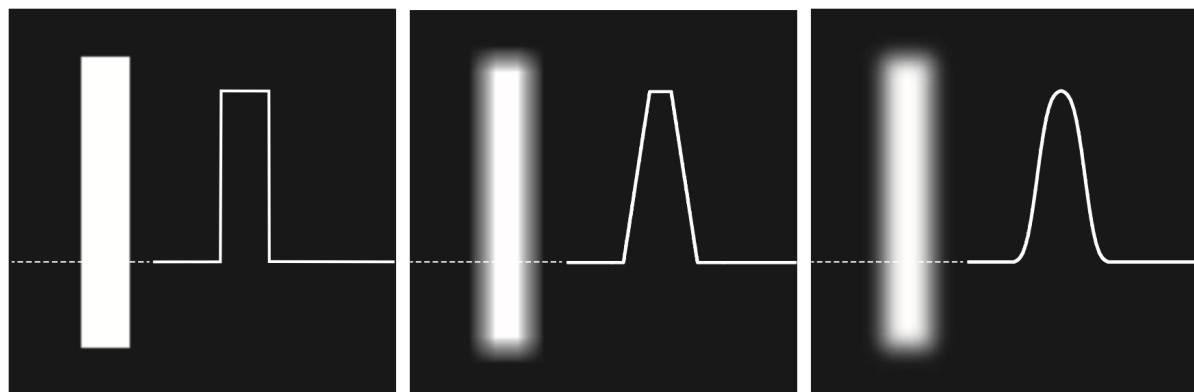
```
In [50]: show_images(test_pattern_lowpass_3, test_pattern_lowpass_11, test_pattern_lowpass_gauss3, test_pattern_lowpass_gauss11, test_pattern_lowpass_gauss21, titles=['Filtro 3x3', 'Filtro 11x11', 'Filtro 21x21', 'Filtro Gauss 3x3', 'Filtro Gauss 11x11', 'Filtro Gauss 21x21'], cols=3, figsize=(10, 7))
```



La figura siguiente muestra más claramente este tipo de comportamiento diferente entre los núcleos de caja y gaussiano.

La imagen del rectángulo se suavizó utilizando un nucleos de caja y gaussiano.

- Como muestran los perfiles de intensidad, el filtro de caja produjo un suavizado lineal, con la transición del negro al blanco (es decir, en un borde) teniendo la forma de una rampa.
 - Utilizaríamos este tipo de filtro cuando se deseé un menor suavizado de bordes.
- Por el contrario, el filtro gaussiano produjo resultados significativamente más suaves alrededor de las transiciones de los bordes.
 - Utilizaríamos este tipo de filtro cuando se deseá un suavizado generalmente uniforme.



Los valores de una función gaussiana a una distancia mayor que 3σ de la media son lo suficientemente pequeños como para ignorarlos.

- Esto significa que si seleccionamos el tamaño de un núcleo gaussiano para que sea $[6\sigma] \times [6\sigma]$, tenemos la seguridad de obtener esencialmente el mismo resultado que si hubiésemos utilizado un núcleo gaussiano arbitrariamente grande.
- Visto de otra manera, esta propiedad nos dice que no se gana nada utilizando un núcleo gaussiano mayor que $[6\sigma] \times [6\sigma]$ para el procesamiento de imágenes.
- Debido a que normalmente trabajamos con núcleos de dimensiones impares, usaríamos el entero impar más pequeño que satisfaga esta condición (por ejemplo, un núcleo de 43×43 si $\sigma = 7$).

```
In [67]: gaussian_kernel_lowpass_43 = crear_kernel_gaussiano(43, 5)
gaussian_kernel_lowpass_85 = crear_kernel_gaussiano(85, 5)

tp_lowpass_gauss43 = aplicar_convolucion_v2(test_pattern, gaussian_kernel_lowpass_43)
tp_lowpass_gauss85 = aplicar_convolucion_v2(test_pattern, gaussian_kernel_lowpass_85)
diferencia = tp_lowpass_gauss43 - tp_lowpass_gauss85
```

```
In [68]: show_images(tp_lowpass_gauss43, tp_lowpass_gauss85, diferencia,
                 titles=['Filtro Gaussiano 43x43', 'Filtro Gaussiano 85x85', 'Diferencia'],
                 cols=3,
                 figsize=(10, 7))
```



Como muestran los resultados de los ejemplos anteriores, el relleno con ceros en una imagen introduce bordes oscuros en el resultado filtrado, y el grosor de los bordes depende del tamaño y tipo de núcleo de filtro utilizado.

Existen otros dos métodos de relleno de imágenes:

- relleno de espejo (también llamado simétrico), en el que los valores fuera de los límites de la imagen se obtienen reflejando la imagen a través de su borde; y
- relleno de replica, en el que los valores fuera del límite se establecen iguales al valor del borde de la imagen más cercano.

Este último relleno es útil cuando las áreas cercanas al borde de la imagen son constantes.

Por el contrario, el relleno de espejo es más aplicable cuando las áreas cercanas al borde contienen detalles de la imagen.

In [152]:

```
gaussian_kernel_lowpass_91 = crear_kernel_gaussiano(91, 15)

tp_lowpass_gauss_zero_padding = aplicar_convolucion(test_pattern, gaussian_kernel_lowpass_91, zero_padding)
tp_lowpass_gauss_mirror_padding = aplicar_convolucion(test_pattern, gaussian_kernel_lowpass_91, mirror_padding)
tp_lowpass_gauss_replicate_padding = aplicar_convolucion(test_pattern, gaussian_kernel_lowpass_91, replicate_padding)
```

In [77]:

```
show_images(tp_lowpass_gauss_zero_padding, tp_lowpass_gauss_mirror_padding, tp_lowpass_gauss_replicate_padding,
           titles=['Zero padding', 'Mirror padding', 'Replicate padding'],
           cols=3,
           figsize=(10, 7))
```



Considere la imagen de tablero de ajedrez de 1024×1024 (figura izquierda), cuyos cuadrados internos tienen un tamaño de 64×64 píxeles.

- La figura de en medio es el resultado del filtrado de paso bajo de la imagen con un núcleo gaussiano de 257×257 (cuatro veces el tamaño de los cuadrados), $K = 1$ y $\sigma = 64$ (igual al tamaño de los cuadrados).
- Este núcleo es lo suficientemente grande como para difuminar los cuadrados.
- Finalmente, la figura de la derecha es el resultado de dividir la figura izquierda por la figura de en medio. Aunque el resultado no es perfectamente plano, definitivamente es una mejora con respecto a la imagen sombreada.

In [82]:

```
checkers = imread('figs/checkers_shade.png', as_gray=True)
checkers = checkers * 255 // 1
checkers = checkers.astype(np.uint8)
```

In [88]:

```
gklp_256 = crear_kernel_gaussiano(257, 64)

checkers_lowpass_256 = aplicar_convolucion(checkers, gklp_256, 'edge')
checkers_result = checkers / checkers_lowpass_256
```

In [89]:

```
show_images(checkers, checkers_lowpass_256, checkers_result,
           titles=['Original', 'Filtro Gaussiano', 'Resultado'],
```

```
cols=3,
figsize=(10, 7))
```



3.5.3. Filtros estadísticos de orden

Los filtros estadísticos de orden son filtros espaciales no lineales cuya respuesta se basa en ordenar (clasificar) los píxeles contenidos en la región abarcada por el filtro.

El suavizado se logra reemplazando el valor del píxel central con el valor determinado por el resultado de la clasificación.

- El filtro más conocido en esta categoría es el filtro de mediana, que, como su nombre lo indica, reemplaza el valor del píxel central por la mediana de los valores de intensidad en la vecindad de ese píxel.
 - proporcionan excelentes capacidades de reducción de ruido para ciertos tipos de ruido aleatorio, con una borrosidad considerablemente menor que los filtros de suavizado lineal de tamaño similar.
 - son particularmente efectivos en presencia de ruido impulsivo (a veces llamado ruido de sal y pimienta, cuando se manifiesta como puntos blancos y negros superpuestos en una imagen).

La mediana, ξ , de un conjunto de valores es tal que la mitad de los valores del conjunto son menores o iguales que ξ y la otra mitad son mayores o iguales que ξ .

Para realizar el filtrado de mediana en un punto de una imagen, primero ordenamos los valores de los píxeles del vecindario, determinamos su mediana y asignamos ese valor al píxel de la imagen filtrada correspondiente al centro del vecindario.

- Por ejemplo, en un vecindario de 3×3 la mediana es el quinto valor más grande, en un vecindario de 5×5 es el decimotercer valor más grande, y así sucesivamente.

Cuando varios valores en una vecindad son iguales, todos los valores iguales se agrupan.

- Por ejemplo, supongamos que una vecindad de 3×3 tiene valores $(10, 20, 20, 20, 15, 20, 20, 25, 100)$. Estos valores se ordenan como

- (10, 15, 20, 20, 20, 20, 20, 25, 100), lo que da como resultado una mediana de 20.
- Por lo tanto, la función principal de los filtros de mediana es forzar que los puntos se parezcan más a sus vecinos.

```
In [110...]: sp = imread('figs/salt-n-pepper.png', as_gray=True)
sp = sp * 255 // 1
sp = sp.astype(np.uint8)
```

```
In [111...]: # funcion para aplicar filtro mediana
def aplicar_filtro_mediana(image, tam):
    """
    Aplica un filtro de mediana a una imagen.

    Parametros:
    image (numpy.ndarray): La imagen a aplicar el filtro.
    tam (int): El tamaño del filtro.

    Regresa:
    numpy.ndarray: La imagen con el filtro aplicado.
    """
    # Obtenemos las dimensiones de la imagen
    image_height, image_width = image.shape

    # Calculamos el padding
    pad = tam // 2

    # Creamos una imagen con padding
    padded_image = np.pad(image, (pad, pad), 'edge')

    # Creamos una imagen de salida vacía
    output_image = np.zeros_like(image)

    # Apply the convolution
    for y in range(pad, image_height + pad):
        for x in range(pad, image_width + pad):
            output_image[y - pad, x - pad] = np.median(padded_image[y - pad:y + pad, x - pad:x + pad])

    return output_image
```

```
In [112...]: sp_gaussian = aplicar_convolucion(sp, crear_kernel_gaussiano(19, 3), 'edge')
```

```
In [113...]: sp_median_3 = aplicar_filtro_mediana(sp, 19)
```

```
In [114...]: show_images(sp, sp_gaussian, sp_median_3,
                  titles=['Original', 'Filtro Gaussiano 19x19', 'Filtro Mediana 19x19'],
                  cols=3,
                  figsize=(10, 7))
```



Los grupos aislados de píxeles que son claros u oscuros con respecto a sus vecinos, y cuya área es menor que $\frac{m^2}{2}$ (la mitad del área del filtro), son obligados por un filtro mediano de $m \times m$ a tener el valor de la intensidad mediana de los píxeles de la vecindad.

El filtro de mediana es, con diferencia, el filtro de estadística de orden más útil en el procesamiento de imágenes, pero no es el único.

- La mediana representa el percentil 50 de un conjunto de números clasificados, pero la clasificación se presta a muchas otras posibilidades.
- El uso del percentil 100 da como resultado el llamado filtro máximo, que es útil para encontrar los puntos más brillantes de una imagen o para erosionar áreas oscuras adyacentes a regiones claras.
- El filtro del percentil 0 es el filtro mínimo y se utiliza para el propósito opuesto.

In [115...]

```
# funcion para aplicar filtro percentile
def aplicar_filtro_percentile(image, tam, p):
    """
    Aplica un filtro de percentile a una imagen.

    Parametros:
    image (numpy.ndarray): La imagen a aplicar el filtro.
    tam (int): El tamaño del filtro.
    p (float): El valor de p.

    Regresa:
    numpy.ndarray: La imagen con el filtro aplicado.
    """

    # Obtenemos las dimensiones de la imagen
    image_height, image_width = image.shape

    # Calculamos el padding
    pad = tam // 2

    # Creamos una imagen con padding
    padded_image = np.pad(image, (pad, pad), 'edge')

    # Creamos una imagen de salida vacía
    output_image = np.zeros_like(image)
```

```
# Apply the convolution
for y in range(pad, image_height + pad):
    for x in range(pad, image_width + pad):
        output_image[y - pad, x - pad] = np.percentile(padded_image[y - pad, x - pad], 19)

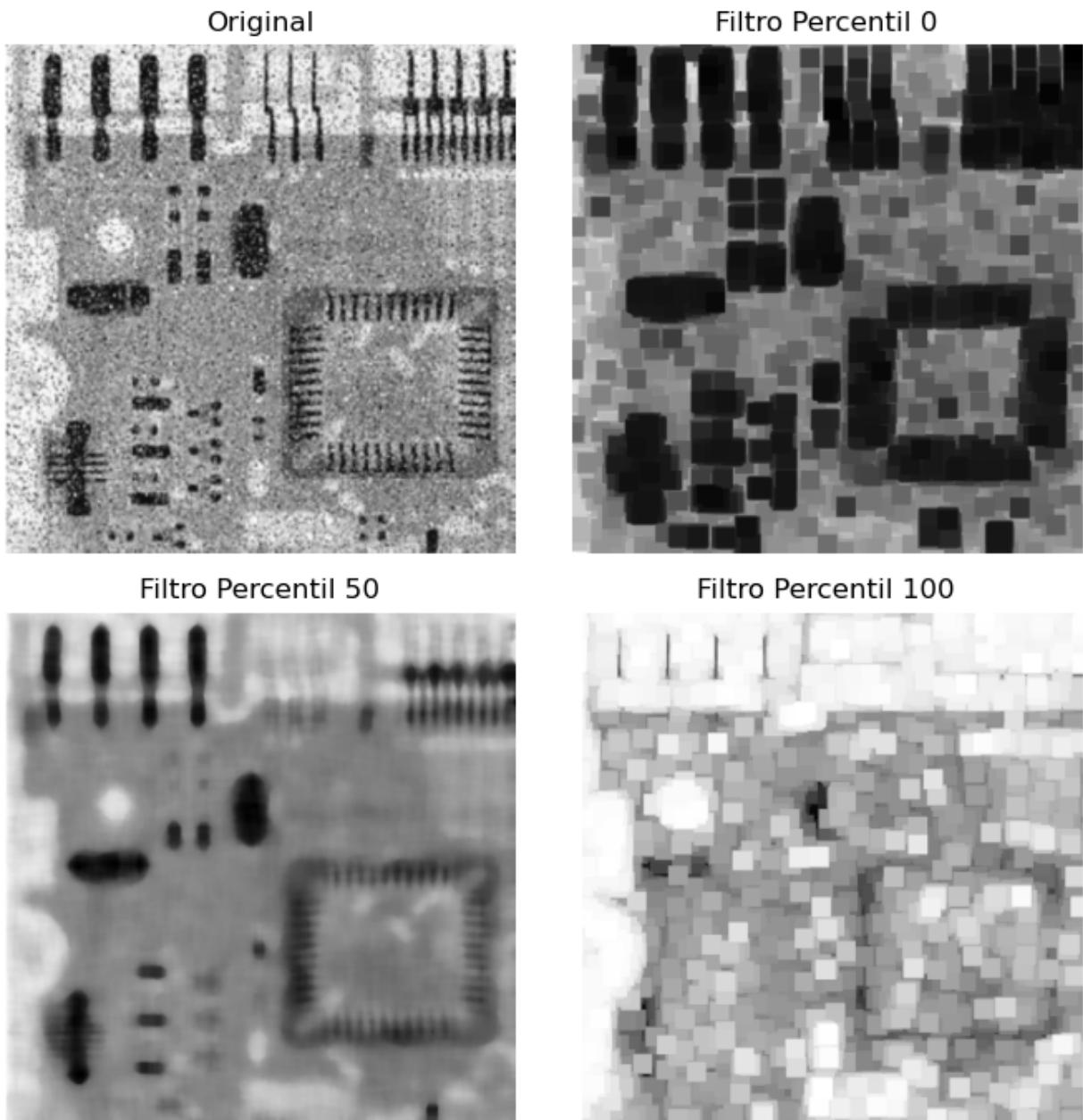
return output_image
```

In [116...]

```
sp_percentile_0 = aplicar_filtro_percentil(sp, 19, 0)
sp_percentile_50 = aplicar_filtro_percentil(sp, 19, 50)
sp_percentile_100 = aplicar_filtro_percentil(sp, 19, 100)
```

In [150...]

```
show_images(sp, sp_percentile_0, sp_percentile_50, sp_percentile_100,
            titles=['Original', 'Filtro Percentil 0', 'Filtro Percentil 50', 'Filtro Percentil 100'],
            cols=2,
            figsize=(7, 7))
```



3.6. Filtros espaciales de nitidez (pasa alto)

El enfoque, o filtro de nitidez, resalta las transiciones de intensidad. Los usos de la nitidez de imágenes van desde la impresión electrónica y la obtención de imágenes médicas hasta la inspección industrial y la guía autónoma en sistemas militares.

Anteriormente, vimos que el desenfoque de la imagen se podía lograr en el dominio espacial promediando (suavizando) los píxeles en un vecindario. Dado que el promedio es análogo a la integración, es lógico concluir que la nitidez se puede lograr mediante la diferenciación espacial.

La fuerza de la respuesta de un operador derivativo es proporcional a la magnitud de la discontinuidad de intensidad en el punto en el que se aplica el operador.

- Por tanto, la diferenciación de imágenes mejora los bordes y otras discontinuidades (como el ruido) y resta importancia a las áreas con intensidades que varían lentamente.

El suavizado a menudo se denomina filtrado de paso bajo, un término tomado del procesamiento en el dominio de la frecuencia.

De manera similar, la nitidez a menudo se denomina filtrado de paso alto. En este caso, se dejan pasar las frecuencias altas (que son responsables de los detalles finos), mientras que las frecuencias bajas se atenúan o rechazan.

3.6.1. Bases

Algunas de las propiedades fundamentales de estos derivados en un contexto digital.

Para simplificar la explicación, inicialmente centramos la atención en las derivadas unidimensionales.

- En particular, estamos interesados en el comportamiento de estas derivadas en
 - áreas de intensidad constante,
 - al inicio y al final de las discontinuidades y
 - a lo largo de rampas de intensidad.

Las derivadas de una función digital se definen en términos de diferencias. Hay varias formas de definir estas diferencias. Sin embargo, requerimos que cualquier definición que usemos para una *primera derivada*:

1. Debe ser cero en zonas de intensidad constante.
2. Debe ser distinto de cero al inicio de un paso o rampa de intensidad.
3. Debe ser distinto de cero a lo largo de las rampas de intensidad.

De manera similar, cualquier definición de *segunda derivada*:

1. Debe ser cero en zonas de intensidad constante.
2. Debe ser distinto de cero al inicio y al final de un paso o rampa de intensidad.

3. Debe ser cero en las rampas de intensidad.

Se trata de cantidades digitales cuyos valores son finitos. Por lo tanto, el cambio de intensidad máximo posible también es finito y la distancia más corta sobre la cual puede ocurrir ese cambio es entre píxeles adyacentes.

Una definición básica de la derivada de primer orden de una función unidimensional $f(x)$ es la diferencia:

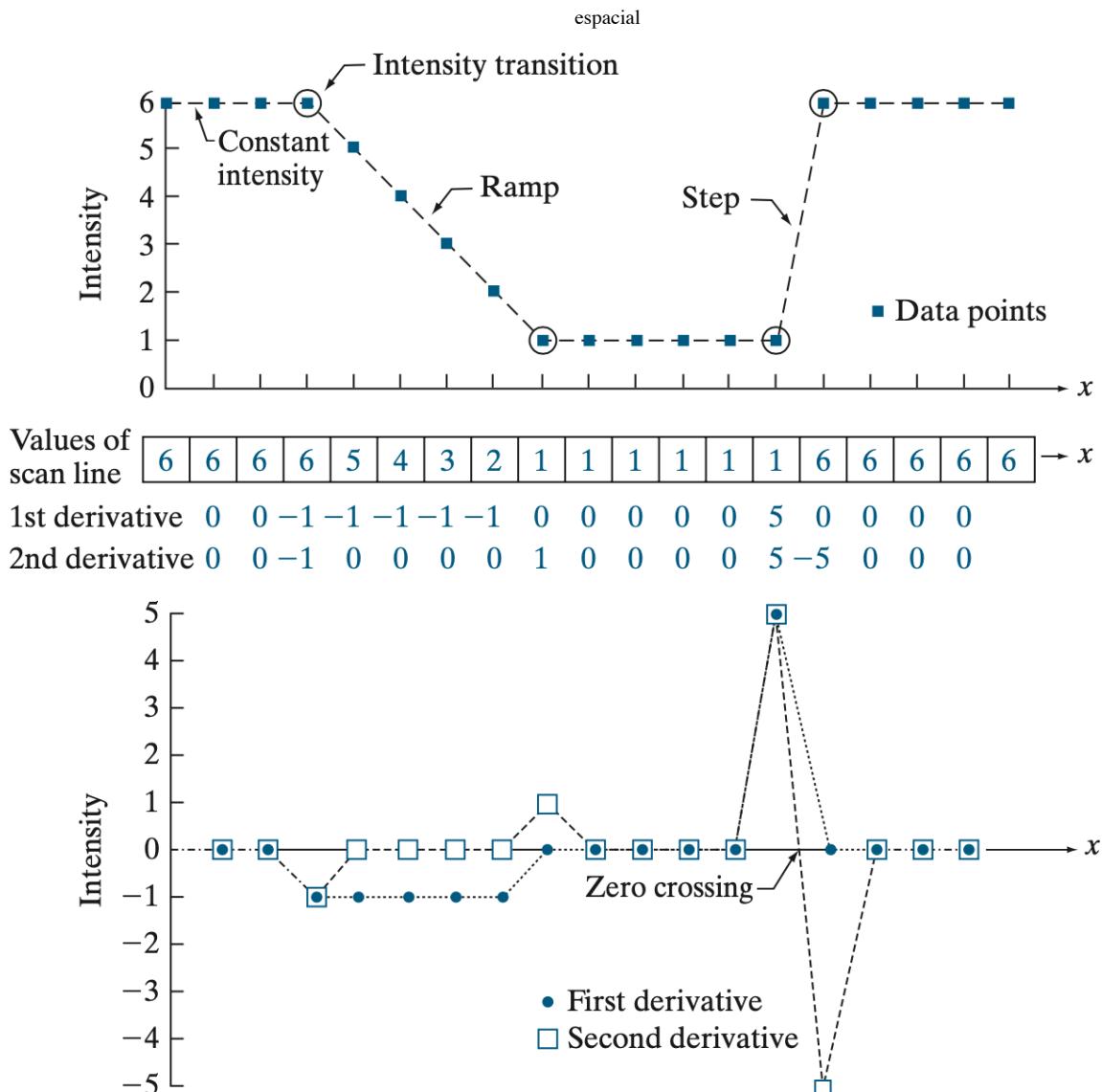
$$\frac{\partial f}{\partial x} = f(x + 1) - f(x)$$

Usamos aquí una derivada parcial para mantener la notación consistente cuando consideramos una función imagen de dos variables, $f(x, y)$, en cuyo momento trataremos con derivadas parciales a lo largo de los dos ejes espaciales.

Definimos la derivada de segundo orden de $f(x)$ como la diferencia

$$\frac{\partial^2 f}{\partial^2 x} = f(x + 1) + f(x - 1) - 2f(x)$$

Estas dos definiciones satisfacen las condiciones establecidas anteriormente, como ilustramos en la figura, donde también examinamos las similitudes y diferencias entre las derivadas de primer y segundo orden de una función digital.



3.6.2. Utilizando la segunda derivada para mejora de nitidez (el Laplaciano)

El enfoque consiste en definir una formulación discreta de la derivada de segundo orden y luego construir un núcleo de filtro basado en esa formulación.

Como en el caso de los núcleos de paso bajo gaussianos, aquí nos interesan los núcleos isotrópicos, cuya respuesta es independiente de la dirección de las discontinuidades de intensidad en la imagen a la que se aplica el filtro.

Se puede demostrar que el operador derivador isotrópico (núcleo) más simple es el laplaciano, que, para una función (imagen) $f(x, y)$ de dos variables, se define como:

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

Como las derivadas de cualquier orden son operaciones lineales, el laplaciano es un operador lineal.

Para expresar esta ecuación en forma discreta, usamos la definición de la ecuación previamente definida, teniendo en cuenta que ahora tenemos una segunda variable.

En la dirección x , tenemos:

$$\frac{\partial^2 f}{\partial x^2} = f(x+1, y) + f(x-1, y) + 2f(x, y)$$

y de manera similar en la dirección y :

$$\frac{\partial^2 f}{\partial y^2} = f(x, y+1) + f(x, y-1) + 2f(x, y)$$

De las tres ecuaciones anteriores se deduce que el laplaciano discreto de dos las variables son:

$$\nabla^2 f = f(x+1, y) + f(x-1, y) + f(x, y+1) + f(x, y-1) + 4f(x, y)$$

Esta ecuación se puede implementar usando convolución con el núcleo de la figura (a); por lo tanto, los mecanismos de filtrado para la nitidez de la imagen son los mismos que para el filtrado de paso bajo; simplemente estamos usando coeficientes diferentes aquí.

Las direcciones diagonales se pueden incorporar en la definición del laplaciano digital agregando cuatro términos más a la ecuación anterior. Debido a que cada término diagonal contendría un término $-2f(x, y)$, el total restado de los términos de diferencia ahora sería $-8f(x, y)$. La figura (b) muestra el núcleo utilizado para implementar esta nueva definición.

Los núcleos (c) y (d) también se utilizan para calcular el laplaciano. Se obtienen de definiciones de segundas derivadas que son negativas de las que usamos aquí. Producen resultados equivalentes, pero se debe tener en cuenta la diferencia de signo al combinar una imagen filtrada por Laplaciano con otra imagen.

0	1	0
1	-4	1
0	1	0

1	1	1
1	-8	1
1	1	1

0	-1	0
-1	4	-1
0	-1	0

-1	-1	-1
-1	8	-1
-1	-1	-1

a b c d

In [151...]

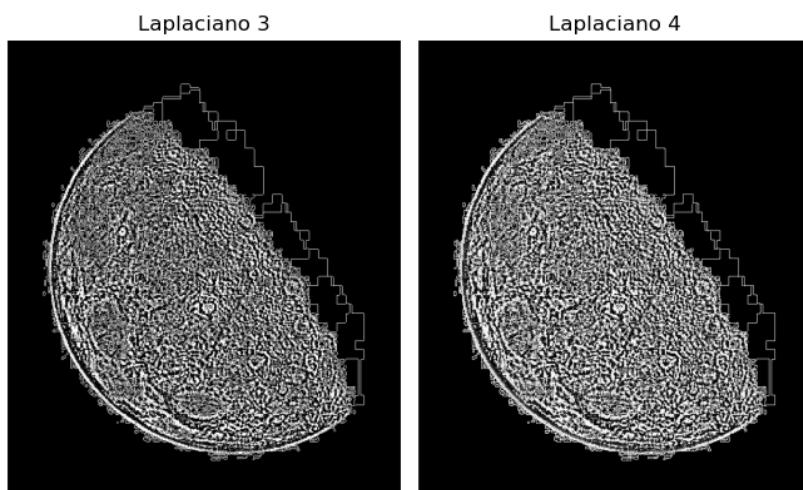
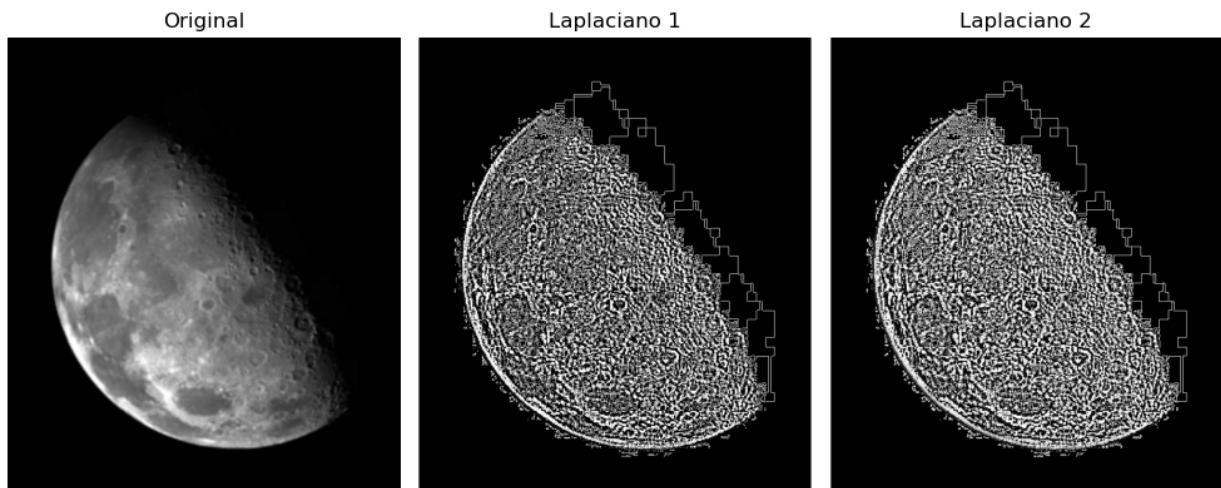
```
moon = imread('figs/moon.png', as_gray=True)
moon = moon * 255 // 1
```

```
moon = moon.astype(np.uint8)
```

```
In [121... laplaciano_1 = np.array([[0, 1, 0], [1, -4, 1], [0, 1, 0]])
laplaciano_2 = np.array([[1, 1, 1], [1, -8, 1], [1, 1, 1]])
laplaciano_3 = np.array([[0, -1, 0], [-1, 4, -1], [0, -1, 0]])
laplaciano_4 = np.array([[-1, -1, -1], [-1, 8, -1], [-1, -1, -1]])
```

```
In [122... moon_laplaciano_1 = aplicar_convolucion(moon, laplaciano_1)
moon_laplaciano_2 = aplicar_convolucion(moon, laplaciano_2)
moon_laplaciano_3 = aplicar_convolucion(moon, laplaciano_3)
moon_laplaciano_4 = aplicar_convolucion(moon, laplaciano_4)
```

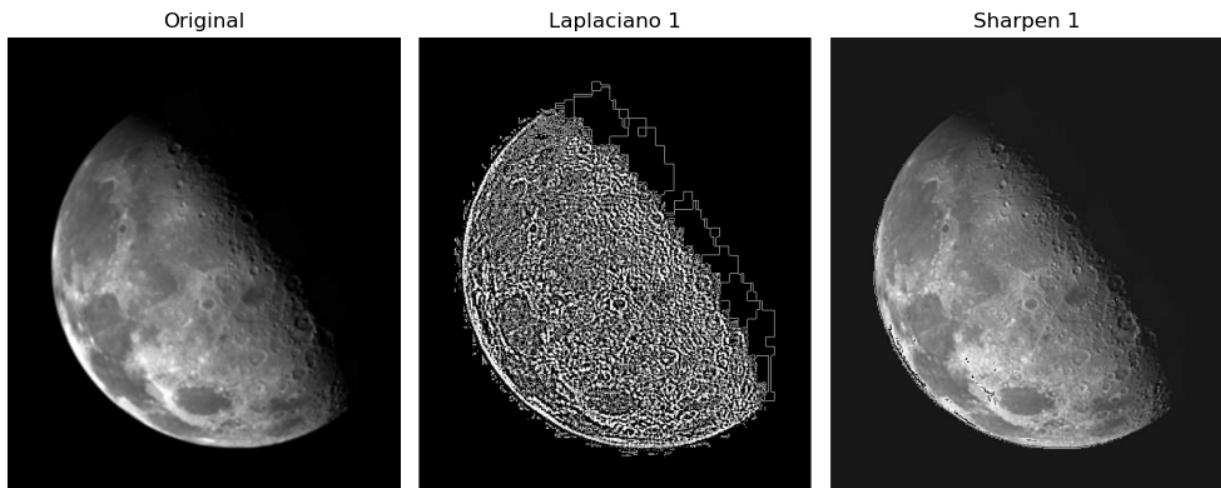
```
In [123... show_images(moon, moon_laplaciano_1, moon_laplaciano_2, moon_laplaciano_3, moon_laplaciano_4,
titles=['Original', 'Laplaciano 1', 'Laplaciano 2', 'Laplaciano 3',
cols=3,
figsize=(10, 10))
```



```
In [128... c = 2
moon_sharpen_2 = moon - c * moon_laplaciano_2

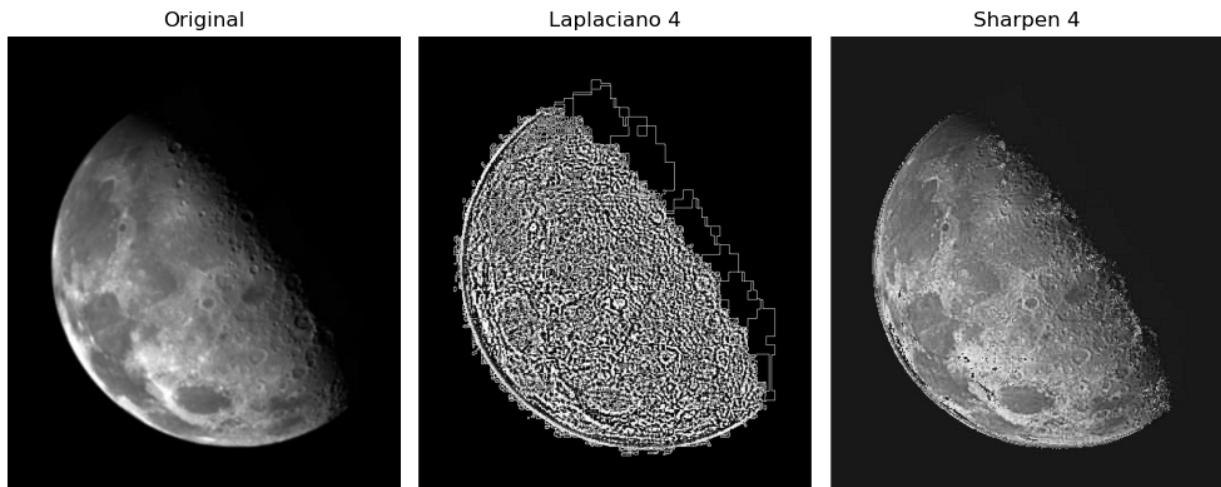
show_images(moon, moon_laplaciano_2, moon_sharpen_2,
titles=['Original', 'Laplaciano 1', 'Sharpen 1'],
```

```
cols=3,
figsize=(10, 10))
```



```
In [129]: moon_sharpen_4 = moon + c * moon_laplaciano_4
```

```
show_images(moon, moon_laplaciano_4, moon_sharpen_4,
            titles=['Original', 'Laplaciano 4', 'Sharpen 4'],
            cols=3,
            figsize=(10, 10))
```



3.6.3. Utilizando la primera derivada para mejora de nitidez (el gradiente)

Las primeras derivadas en el procesamiento de imágenes se implementan utilizando la magnitud del gradiente.

- El gradiente de una imagen f en las coordenadas (x, y) se define como el vector columna bidimensional:

$$\nabla f = \text{grad}(f) = \begin{bmatrix} g_x \\ g_y \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix}$$

Este vector tiene la importante propiedad geométrica de que apunta en la dirección de la mayor tasa de cambio de f en la ubicación (x, y) .

La magnitud (longitud) del vector ∇f , denotada como $M(x, y)$, donde:

$$M(x, y) = \text{mag}(\nabla f) = \sqrt{g_x^2 + g_y^2}$$

es el valor en (x, y) de la tasa de cambio en la dirección del vector gradiente.

Tenga en cuenta que $M(x, y)$ es una imagen del mismo tamaño que el original, creada cuando se permite que x e y varíen en todas las ubicaciones de píxeles en f .

Como en el caso del laplaciano, ahora definimos aproximaciones discretas a las ecuaciones anteriores y a partir de ellas formulamos los núcleos apropiados.

Para simplificar la discusión que sigue, usaremos la notación de la figura para denotar las intensidades de los píxeles en una región de 3×3 .

- Por ejemplo, el valor del punto central, z_5 , denota el valor de $f(x, y)$; z_1 denota el valor de $f(x - 1, y - 1)$; etcétera.

z_1	z_2	z_3
z_4	z_5	z_6
z_7	z_8	z_9

Las aproximaciones a g_x y g_y usando una vecindad de 3×3 centrada en z_5 son las siguientes:

$$g_x = \frac{\partial f}{\partial x} = (z_7 + 2z_8 + z_9) - (z_1 + 2z_2 + z_3)$$

y

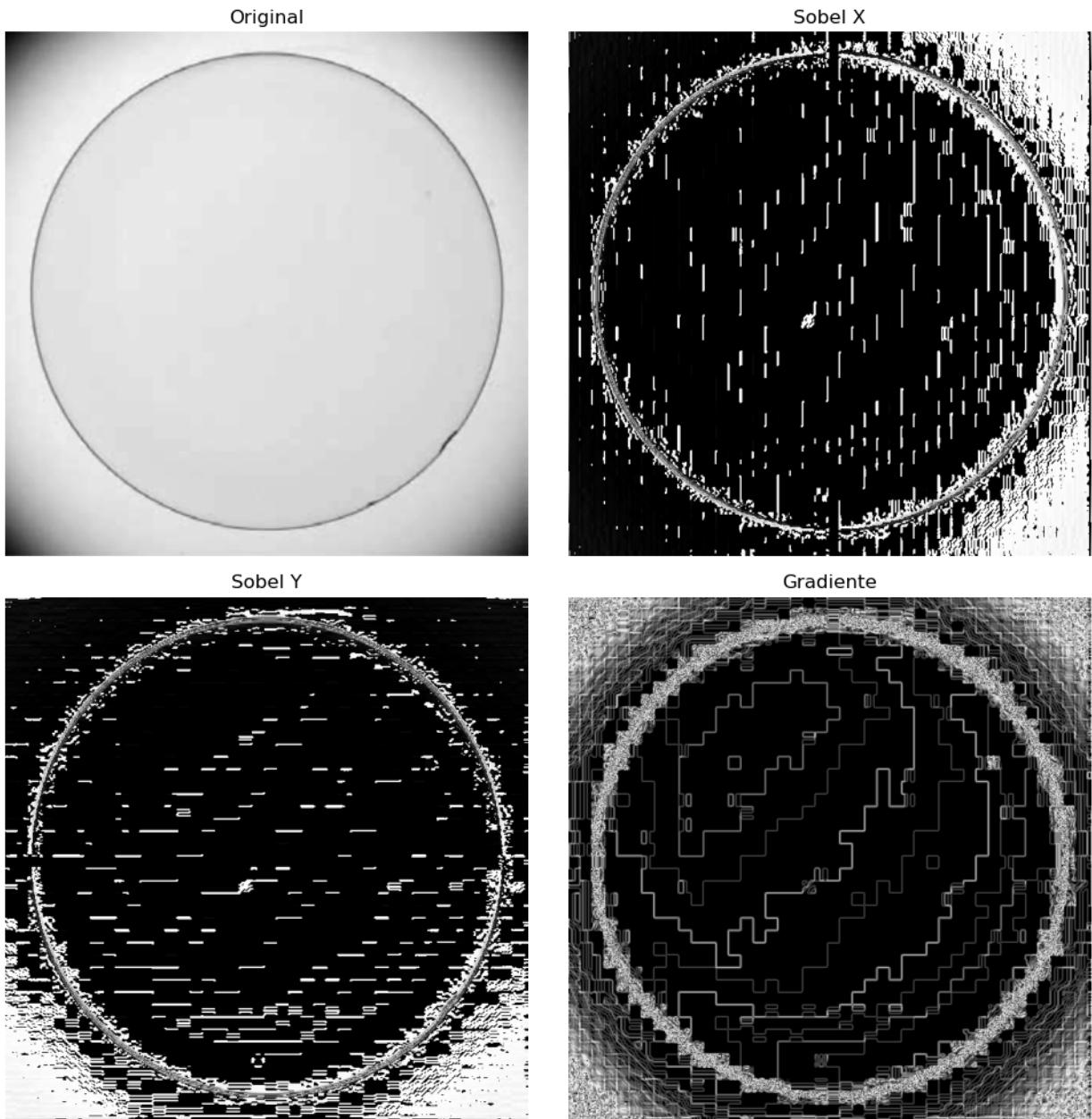
$$g_y = \frac{\partial f}{\partial y} = (z_3 + 2z_6 + z_9) - (z_1 + 2z_4 + z_7)$$

```
In [131... glass = imread('figs/glass.png', as_gray=True)
glass = glass * 255 // 1
glass = glass.astype(np.uint8)
```

```
In [143... sobel_x = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
sobel_y = np.array([[1, -2, -1], [0, 0, 0], [1, 2, 1]])

glass_sobel_x = aplicar_convolucion(glass, sobel_x)
glass_sobel_y = aplicar_convolucion(glass, sobel_y)
glass_gradient = np.sqrt(glass_sobel_x**2 + glass_sobel_y**2)
```

```
In [146... show_images(glass, glass_sobel_x, glass_sobel_y, glass_gradient,
titles=['Original', 'Sobel X', 'Sobel Y', 'Gradiente'],
cols=2,
figsize=(10, 10))
```



3.7. Filtros de pasa alto, rechazo de banda y pasa de banda a partir de filtros de pasa bajo

3.8. Combinación de métodos de mejora espacial