Práctica 2. Filtrado espacial de imágenes biomédicas

Gonzalo Mesas Aranda

10 de noviembre de 2023

1. Introducción

La siguiente práctica ha sido realizada en Python por decisión personal. Se detallará tanto el desarrollo de cada ejercicio como su solución. La **imagen** a estudiar es la número **8**.

2. Pasos previos

Antes de proceder a la resolución de la práctica, es necesario llevar a cabo unos sencillos pasos previos. Entre ellos se encuentran la importación de librerías y la apertura de la imagen correspondiente. La única diferencia respecto a la práctica anterior, es la adición de una nueva librería, **OpenCv**. Esta se usará para la implementación y comparación de alguno de los filtros.

```
import numpy as np
import pydicom as pdc
import matplotlib.pyplot as plt
import cv2

# Abrir la imagen
def abrir_imagen(num_imagen):
    return pdc.dcmread(fr"C:\Users\Gonzalo_MA\Desktop\Universidad\Tercero
    \Imagenes biomedicas\Practica 1/im{num_imagen}.dcm")

imagen = abrir_imagen(8)

# Guardar la imagen en una matriz de datos indicando los bits por pixel
def imagen_bits(imagen, bpp = 12):
    if bpp == 12:
        sol = np.array(imagen.pixel_array)
    elif bpp == 8:
        sol = (np.array(imagen.pixel_array) * (255/4095)).astype(np.uint8)
    return sol, bpp

# Creaamos la matriz de la imagen con 12 bpp
imagen_pixeles, bpp = imagen_bits(imagen, 12)
```

Listing 1: Librerías y pasos previos.

3. Ejercicio 1

3.1. Enunciado

Realizar un filtrado espacial de la imagen y visualizar y valorar el resultado. Para el filtrado, deberá utilizar dos algoritmos:

1. Utilizando la función imfilter(...) de matlab. Los algoritmos a emplear deberán ser:

   a) Filtro paso bajo 5x5, todos los coeficientes del filtro valen 1.

   b) Filtro de Sobel.

   c) Filtro de Prewitt.

   d) Filtro gaussiano de 5x5, con sigma=1.

   e) Filtro laplaciano de 3x3.

   f) Filtro logaritmo de una gaussiana (LoG) de 5x5, con sigma=1.

   Una vez obtenida la imagen filtrada, busque la manera de visualizar de forma óptima el resultado obtenido para cada una de los seis imágenes resultado (use los comandos imshow() o imadjust() que ha aprendido a usar anteriormente) y seleccione el que ofrece una mejor visualización subjetiva de la información de interés. Nota: El histograma de la imagen le puede resultar de utilidad, así como las herramientas aprendidas en la práctica primera.

2. Para el filtro que considere que permite una mejor visualización de la zona de interés, codifique un algoritmo propio que obtenga un resultado similar al de la rutina de Matlab.

3.2. Filtro paso bajo

Antes de explicar en que consiste este filtro, es importante aclarar el concepto de **convolución**. En el ámbito de procesado de imagen, aplicar una convolución consiste en hacer el producto escalar del kernel y la submatriz que se corresponda con el kernel en cada momento. A continuación, el resultado se almacena en el píxel en el que está centrado el kernel.

Los pasos seguidos para la aplicación del filtro son los siguientes:

- Creación del **kernel** del tamaño dado como una matriz de unos.
- Aumentar el tamaño de la imagen por cada lado los píxeles correspondientes, evaluados en cero, para poder aplicar la convolución del kernel en los extremos **(Zero-padding)**.
- Inicializar una matriz para almacenar el resultado.

- Aplicar la convolución.
- Devolver el resultado.

En la siguiente propuesta para el algoritmo, se recorren arrays numpy con el uso de bucles. Esta práctica no es recomendada, ya que la eficiencia se ve afectada considerablemente. Su uso es debido a que, en general los tiempos de ejecución en la práctica no son considerables y la práctica tiene una finalidad didáctica. Por este motivo no se ha usado una librería auxiliar, por lo que se ha priorizado la programación a más bajo nivel.

```
def filtro_pb(imagen, tam_kernel = 5):
    # Creamos el kernel
    kernel = np.ones((tam_kernel, tam_kernel))

    # Pad de la imagen e inicialización
    t = tam_kernel//2
    im_pad = np.pad(imagen, t)
    im_filtrada = np.zeros((512, 512))

    # Aplicamos la convolucion
    for x in range(t, 512+t):
        for y in range(t, 512+t):
            im_filtrada[x-t, y-t] = (np.sum(kernel
            * im_pad[x-t:x+t+1, y-t:y+t+1]))/(tam_kernel**2)

    return im_filtrada.astype(np.uint16)
```

Listing 2: Código del filtro paso bajo.

El filtro ha sido aplicado tanto a la imagen original como a la imagen con ventana de tejido blando (Ver Figura1), zona de interés de la imagen. Para ello se ha hecho uso de técnicas estudiadas en la práctica uno. Para ver el código en su totalidad, se puede acceder al repositorio (Ver 6).

3.3. Conclusión de filtro paso bajo

El resultado después de aplicar el filtro es el suavizado de la imagen a la par de una reducción de ruido. Se puede observar con más claridad en la imagen de la ventana, ya que la original de por sí tiene poco ruido y pocos detalles.

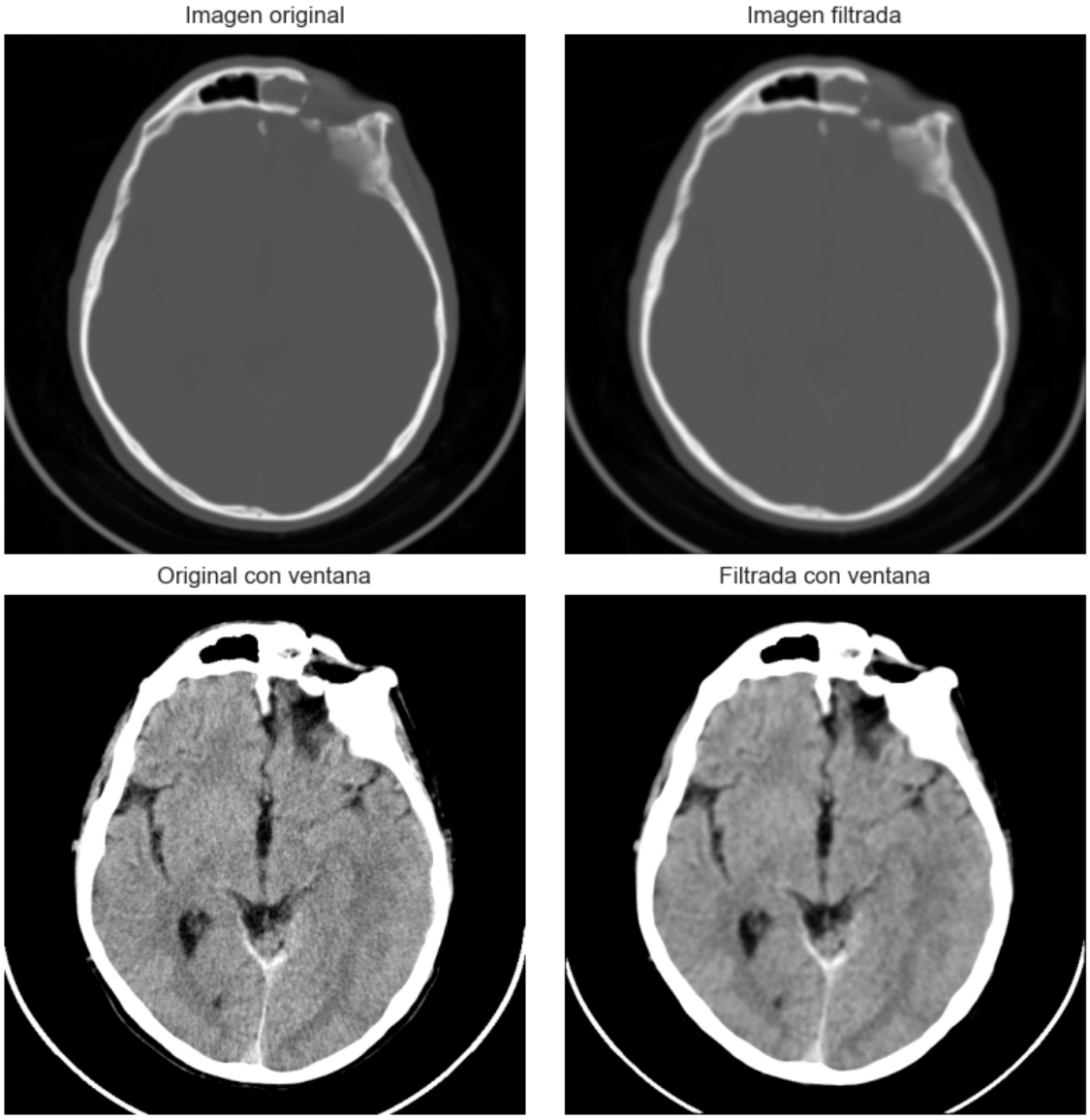

Figura 1: Visualización del filtro paso bajo 5x5.

3.4. Filtro de Sobel

El filtro de Sobel se basa en la aproximación discreta del gradiente de la función de intensidad de la imagen. Para ello emplea dos kernels, uno en cada dirección del espacio. A partir de estas dos componentes, se puede calcular el módulo del gradiente de la siguiente manera:

$$G = \sqrt{G_x^2 + G_y^2}$$

Los pasos seguidos para la implementación son los siguientes:

- Creación de los **kernels**.
- **Zero-padding** de la imagen e inicialización de las matrices almacenadoras.
- **Convolución** con ambos kernels.
- Cálculo del **módulo del gradiente**.

Además, se puede calcular la fase, la cual ha sido implementada en la función aunque no se usa para este apartado. La propuesta de programación es la siguiente:

```
def sobel(imagen):
    # Pad de la imagen e inicialización
    im_pad = np.pad(imagen, 1)
    sobel_x = np.zeros((512, 512))
    sobel_y = np.empty((512, 512))

    # Creamos los kernels
    gx = np.array([[-1, 0, 1],
                   [-2, 0, 2],
                   [-1, 0, 1]])
    gy = gx.T

    # Aplicamos las convoluciones
    for x in range(1, 513):
        for y in range(1, 513):
            sobel_y[x-1, y-1] = np.sum(gx * im_pad[x-1:x+2, y-1:y+2])

    for x in range(1, 513):
        for y in range(1, 513):
            sobel_x[x-1, y-1] = np.sum(gy * im_pad[x-1:x+2, y-1:y+2])

    # Modulo y fase
    sobel_xy = np.sqrt(sobel_x**2 + sobel_y**2)
    fase = np.arctan2(sobel_y, sobel_x)

    return sobel_x, sobel_y, sobel_xy.astype(np.uint16), fase
```

Listing 3: Código del filtro de Sobel.

3.5. Conclusión Sobel

Un ejemplo completo de las diferentes partes del proceso que sigue el filtrado con Sobel, se puede ver reflejado en la figura 2. Se puede ver como las componentes del gradiente recogen principalmente información de su dirección. El resultado final refleja claramente los bordes del cráneo y los límites de los senos paranasales.

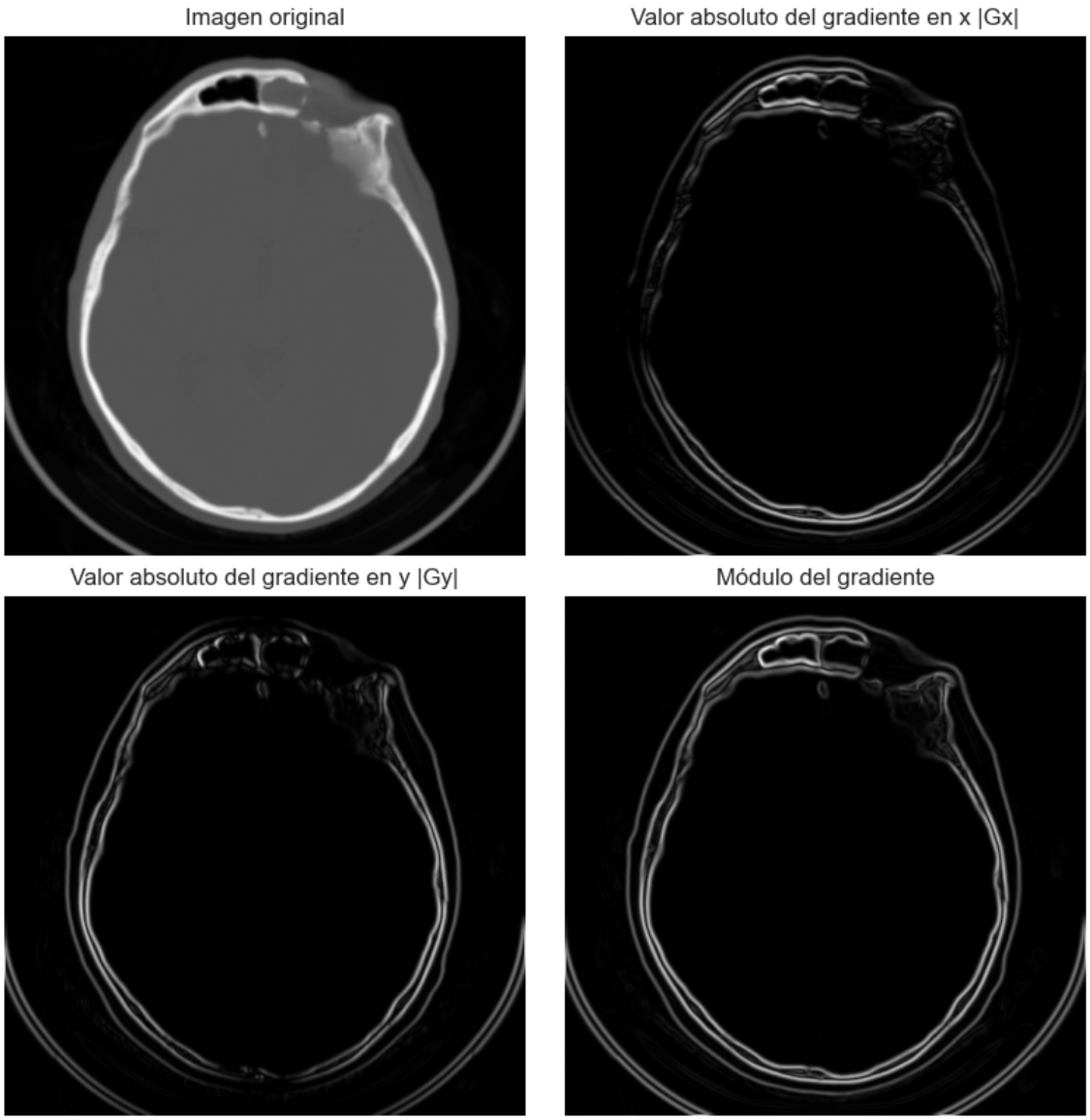

Figura 2: Visualización de la imagen original con filtro Sobel.

A efectos de comparación, se muestra el filtro de Sobel aplicado a diferentes imágenes, de la cual podemos destacar la ventana filtrada, en la que se muestran con

precisión los bordes de las estructuras. Hay que tener en cuenta que la imagen es de tomografía computarizada.

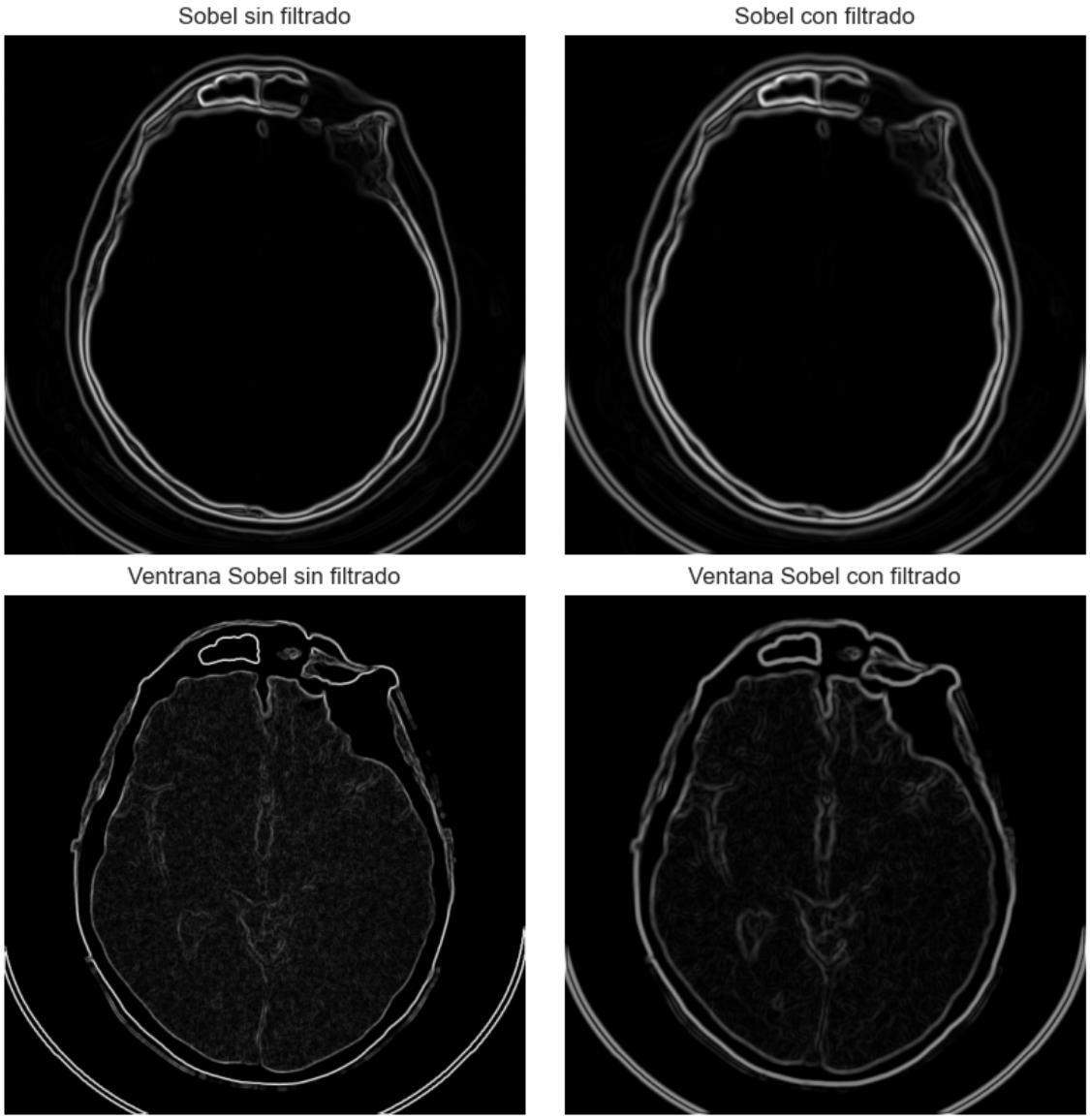

Figura 3: Comparación de imágenes con filtro Sobel.

3.6. Filtro Prewitt

El filtro de Prewitt tiene todas las características del filtro de Sobel (Ver 3.4). La única diferencia es la manera de aproximar el gradiente, por lo que los kernels cambian ligeramente. La diferencia es mínima pero puede presentar algunas desimilitudes en ciertos bordes por la ponderación que asigna a los píxeles.

Operador Sobel en la dirección x:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

Operador Sobel en la dirección y:

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Operador Prewitt en la dirección x:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

Operador Prewitt en la dirección y:

$$G_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

El código no ha sido incluido ya que es igual al filtro de Sobel, aunque con distintos kernels.

3.7. Conclusión Prewitt

A continuación se puede observar la gran similitud entre los resultados de ambos filtros. Para ver la comparación del filtro de Sobel, ver Figura 3.

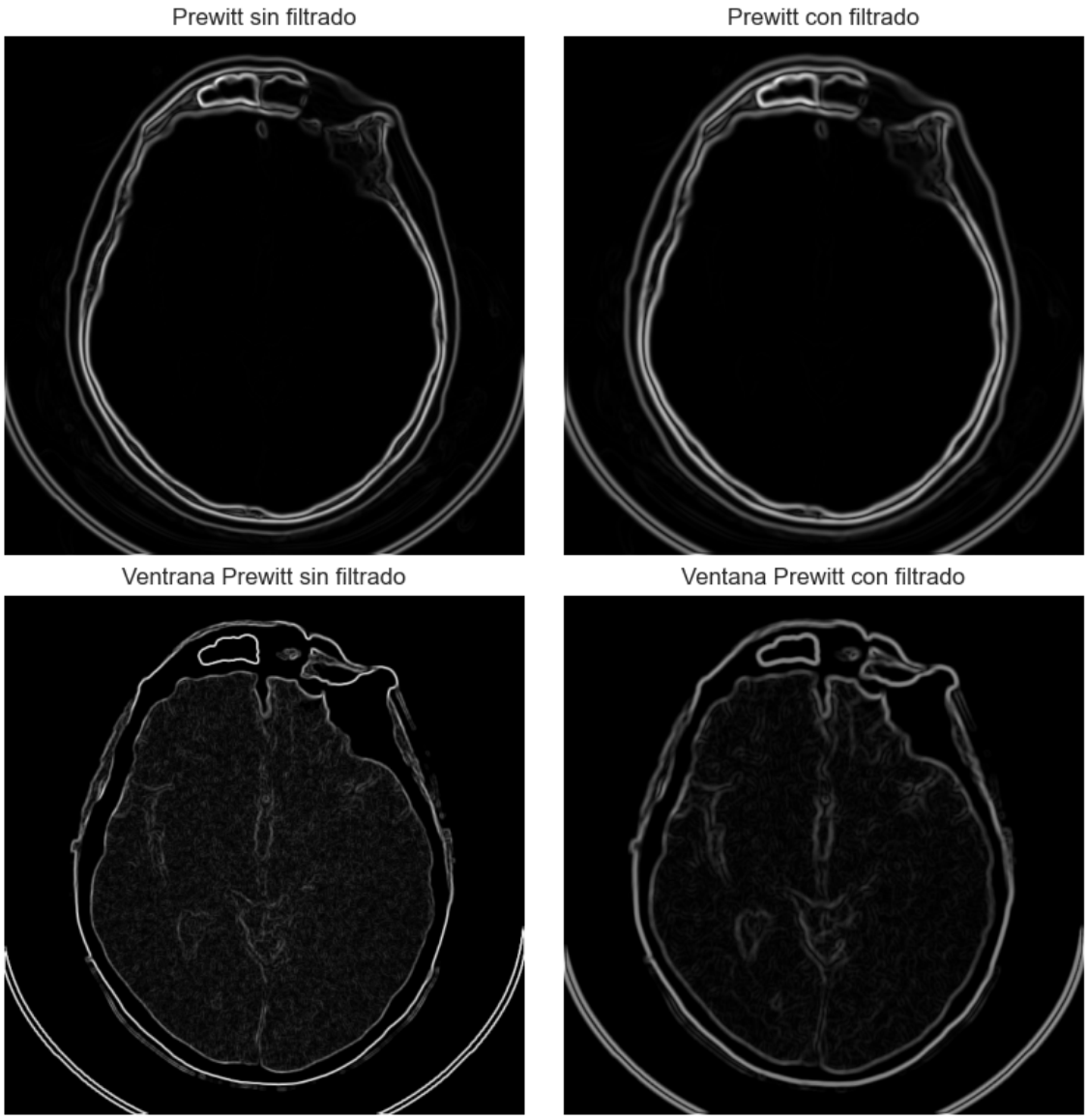

Figura 4: Comparación de imágenes con filtro Prewitt.

3.8. Filtro gaussiano

El filtro gaussiano tiene como objetivo suavizar la imagen, de igual forma que el filtro paso bajo visto anteriormente. La diferencia está en el uso de un kernel que sigue una distribución gaussiana, el cual se calcula a partir de la función gaussiana bidimensional en función del tamaño del kernel y la desviación. Además, debe estar centrada en el píxel central del kernel.

$$G(x,y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{(x-\mu)^2 + (y-\mu)^2}{2\sigma^2}\right) \tag{1}$$

El proceso para aplicar el filtro es igual al de los filtros anteriores. La propuesta para este filtro es la siguiente.

```
def gauss(imagen, tamano = 5, desviacion = 1):
    # Creamos el kernel
    t =  tamano//2
    kernel = np.fromfunction(lambda x, y:
                             (1/ (2 * np.pi * desviacion ** 2))
                             * np.exp(-((x - t) ** 2 + (y - t) ** 2)
                             / (2 * desviacion ** 2)),
                             (tamano, tamano))
    suma_kernel = np.sum(kernel)

    # Pad de la imagen e inicialización
    im_pad = np.pad(imagen, t)
    im_filtrada = np.zeros((512, 512))

    # Aplicamos la convolucion
    for x in range(t, 512+t):
        for y in range(t, 512+t):
            im_filtrada[x-t, y-t] = (np.sum(kernel
            * im_pad[x-t:x+t+1, y-t:y+t+1]))
            /(suma_kernel)

    return im_filtrada.astype(np.uint16)
```

Listing 4: Código del filtro gaussiano.

3.9. Conclusión de filtro gaussiano

El resultado es parecido al obtenido al aplicar el filtro paso bajo 5x5 (Ver Figura 1), aunque subjetivamente la imagen filtrada con ventana se ve mejor ya que el suavizado no es tan intenso y deja distinguir mejor las estructuras, sin perder su función. Una posible explicación a parte de la distinta naturaleza de los filtros, es que el filtro de Sobel es un filtro que aproxima el gradiente (primera derivada), la cual es más sensible al ruido que el filtro gaussiano que no aproxima ninguna derivada.

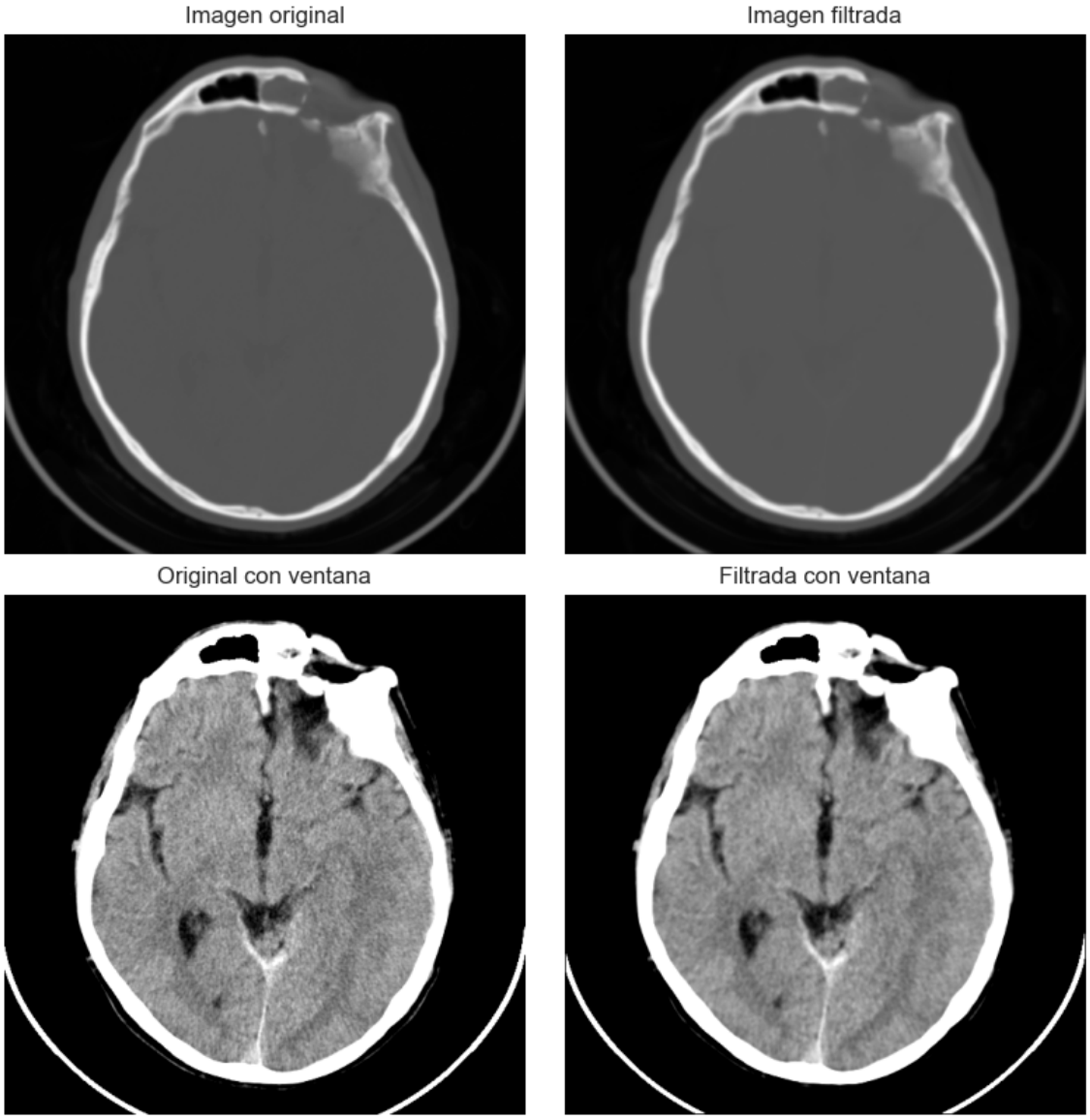

Figura 5: Comparación de imágenes con filtro gaussiano.

3.10. Filtro laplaciano

El filtro laplaciano se basa en la aproximación de la segunda derivada de la función de intensidad de la imagen. Para ello utiliza un kernel específico, que se mostrará a continuación.

Una vez obtenida la imagen filtrada, con información de los detalles finos, como bordes y otras características, se suma a la imagen original para que se vean resaltados estos detalles. Esta suma puede estar ponderada.

La propuesta para este filtro es la siguiente:

```
def laplaciano(imagen):
    # Creamos el kernel
    kernel = np.array([[0,1,0], [1,-4,1], [0,1,0]])

    # Pad de la imagen e inicialización
    im_pad = np.pad(imagen, 1)
    im_filtrada = np.zeros((512, 512), dtype=np.int32)

    # Aplicamos la convolucion
    for x in range(1, 513):
        for y in range(1, 513):
            im_filtrada[x-1, y-1] = (np.sum(kernel * im_pad[x-1:x+2, y-1:y+2]))

    im_orig_y_filtr = imagen.astype(np.int32) + im_filtrada
    im_orig_y_filtr = np.clip(im_orig_y_filtr, 0, (2**bpp)-1)

    return im_filtrada, im_orig_y_filtr.astype(np.uint16)
```

Listing 5: Código del filtro laplaciano.

3.11. Conclusión de filtro laplaciano

En la comparación (Ver Figura 6) se observa que el resultado no mejora ya que, al ser la segunda derivada una herramienta tan sensible al ruido, las imágenes finales solo se ven más borrosas y no se resaltan las zonas de interés. Una posible solución sería un previo filtrado paso bajo para eliminar el ruido. Además otro paso adicional que se podría computar es el cálculo del cruce por cero, que obtendría quizás una ubicación más precisa de los bordes.

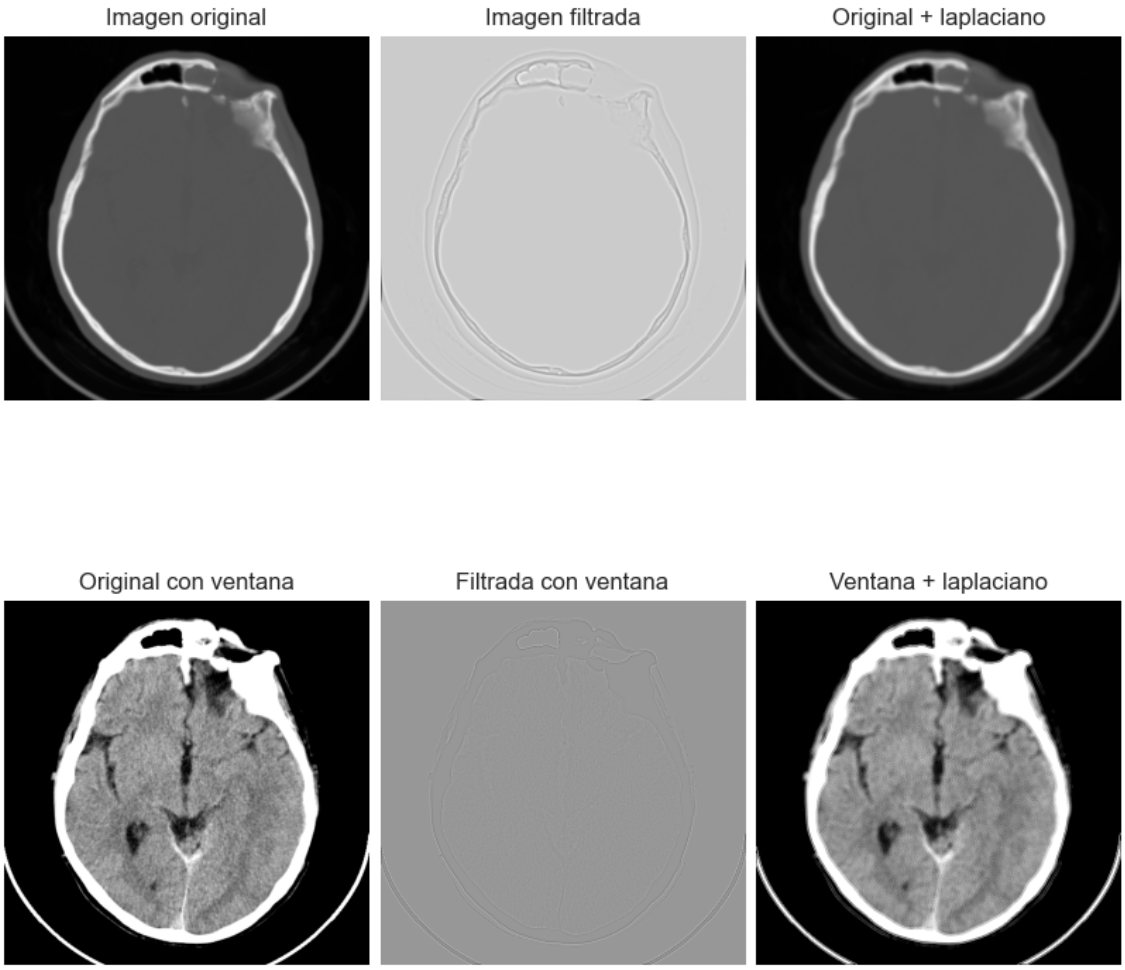

Figura 6: Comparación de imágenes con filtro laplaciano.

3.12. Filtro LoG

El filtro LoG (Logarithm of Gaussian), consiste en aplicar el filtro gaussiano y posteriormente el laplaciano. Ambos son lineales así que esto es posible. No obstante, hay una forma directa de calcular un kernel que aplique ambos al mismo tiempo. Esto se hace calculando la derivada del gradiente de la función gaussiana bidimensional (Ver Ecuación 1).

$$LoG(x,y) = \left(\sigma^2 - \frac{(x-\mu)^2 + (y-\mu)^2}{2\sigma^2}\right) \cdot e^{-\frac{(x-\mu)^2+(y-\mu)^2}{2\sigma^2}} \tag{2}$$

La función desarrollada para implementar este filtro utiliza la librería OpenCv.

```python
def LoG(imagen):
    # Aplica un filtro gaussiano
    imagen_suavizada = cv2.GaussianBlur(imagen, (5, 5), 1)

    # Aplica el operador Laplaciano
    filtro_log_cv2 = cv2.Laplacian(imagen_suavizada, cv2.CV_16U, ksize=5)

    return np.clip(v + 0.02*filtro_log_cv2, 0, 4095)
```

Listing 6: Código del filtro LoG.

3.13. Conclusión LoG

Con este filtro se ha solventado el problema del filtro laplaciano, la imagen final no es tan ruidosa. A parte de esta mejora, la aplicación del filtro no supone una diferencia significativa con la imagen original.

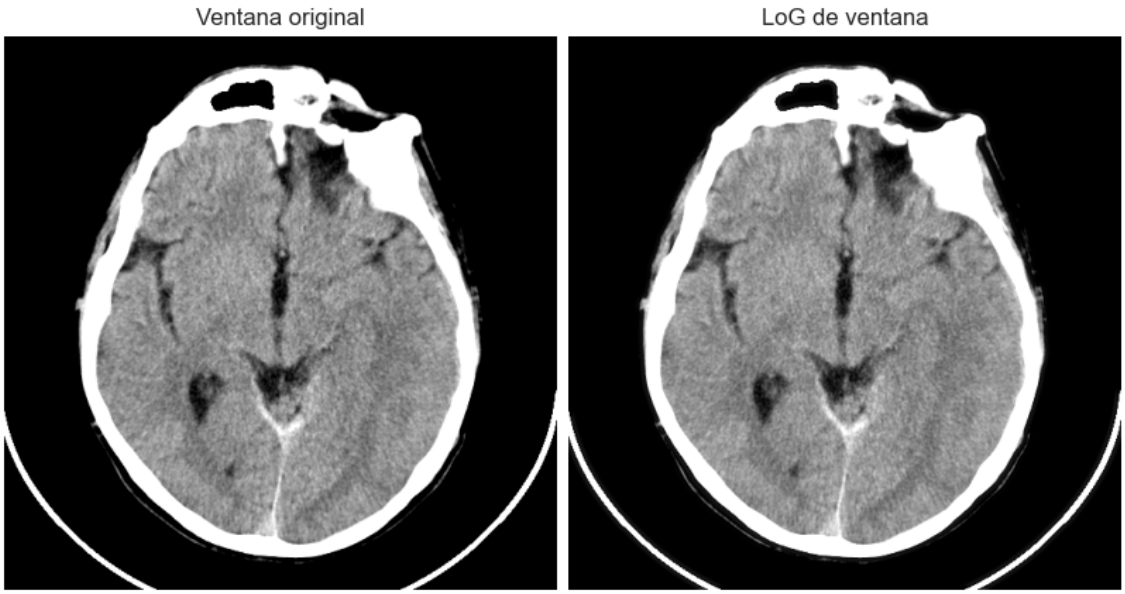

Figura 7: Comparación de imágenes con filtro LoG.

3.14. Conclusión

Para concluir el segundo apartado y por tanto el ejercicio, se pueden destacar de entre los filtros de detección de bordes, el filtro de Sobel (Ver Figura 3), aunque con mucha similitud con el de Prewitt. De los filtros de suavizado, el gaussiano es el que da mejor resultado.

Esta conclusión es subjetiva y está sujeta a la imagen estudiada. Además, los filtros han sido programados a mano, por lo que el apartado segundo de este primer ejercicio se da por concluido.

4. Ejercicio 2

4.1. Enunciado

Realizar una detección de los bordes de los objetos de interés en la imagen utilizando técnicas de gradiente.

1. Utilice la función edge(), en los siguientes algoritmos:

   a) Filtro de Sobel, con un umbral que deberá optimizar. Se recomienda probar varios umbrales y valorar el resultado obtenido.

   b) Filtro de Canny, con 3 parámetros que deberá optimizar.

2. Visualice los resultados obtenidos con los dos filtros de forma óptima, aplicando la técnica de modificación del contraste (práctica 1) que le parezca más adecuada.

3. Indique el método que obtenga mejor la información deseada, en el sentido de que la zona de interés está definida con claridad, desde su punto de vista subjetivo.

4.2. Binarización con Sobel

Para la binarización con el filtro de Sobel, se aplicará el filtro a la ventana de la imagen original, la cual contiene estructuras de interés. Una vez aplicado el filtro, se descartan todos los píxeles que no superen un umbral, el cual variaremos para ver el resultado óptimo. A continuación se muestra el proceso sobre la ventana original y posteriormente sobre la ventana filtrada.

```
umbrales = np.linspace(500, 2000, 9).tolist()
_, ax = plt.subplots(3, 3, figsize = (15, 15))

i = 0
for x in range(3):
    for y in range(3):
        umbral = umbrales[i]
        ax[x,y].imshow(1 - (s_xyvf < umbral), cmap = "gray")
        ax[x,y].axis("off")
        ax[x,y].set_title(f"Sobel con umbral {umbral}")
        i += 1
```

Listing 7: Código de binarización con sobel.

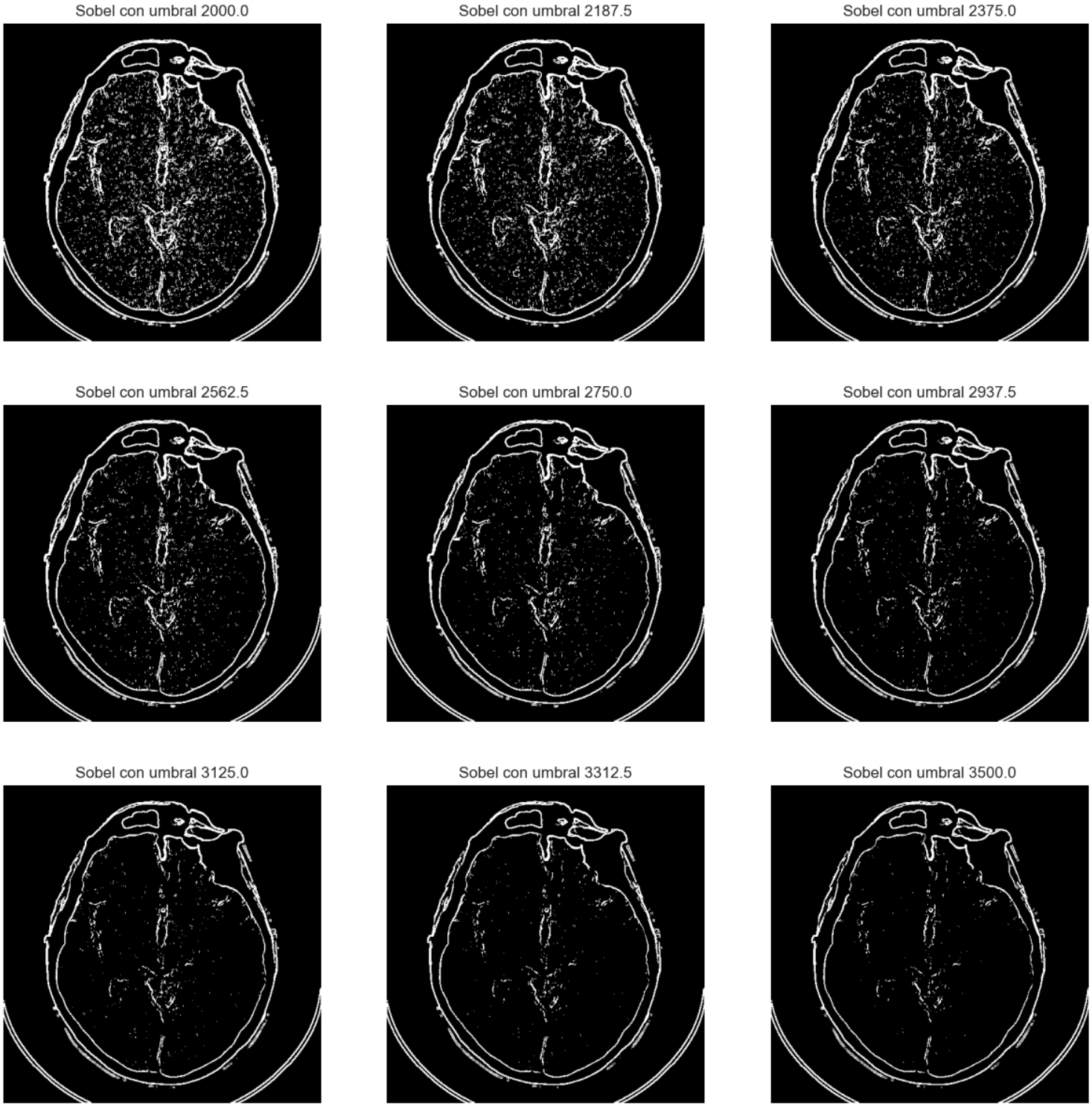

Figura 8: Binarización con Sobel sobre ventana original.

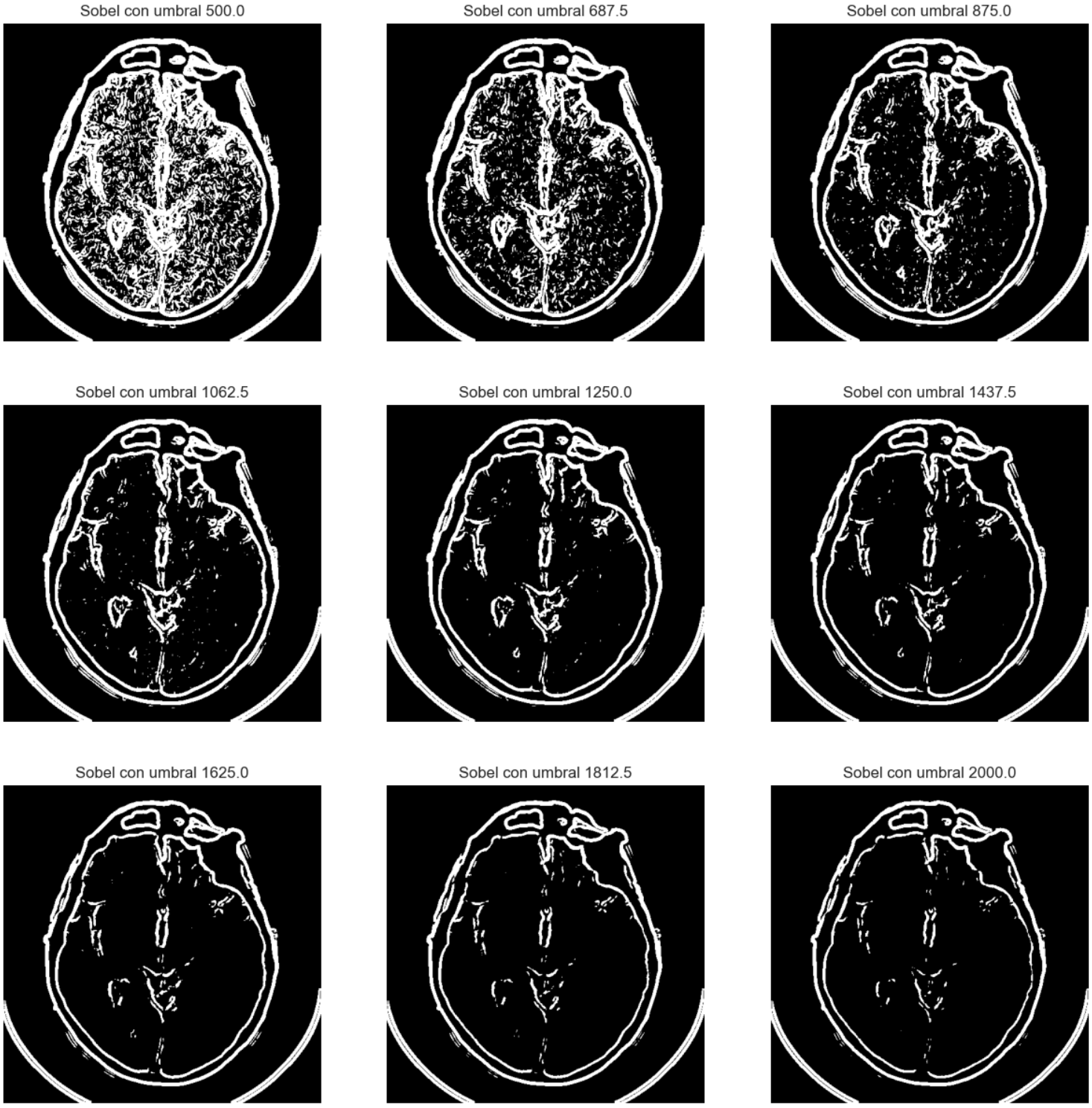

Figura 9: Binarización con Sobel sobre ventana filtrada.

A parir de los umbrales anteriores, se seleccionan los más adecuados de cada tipo.

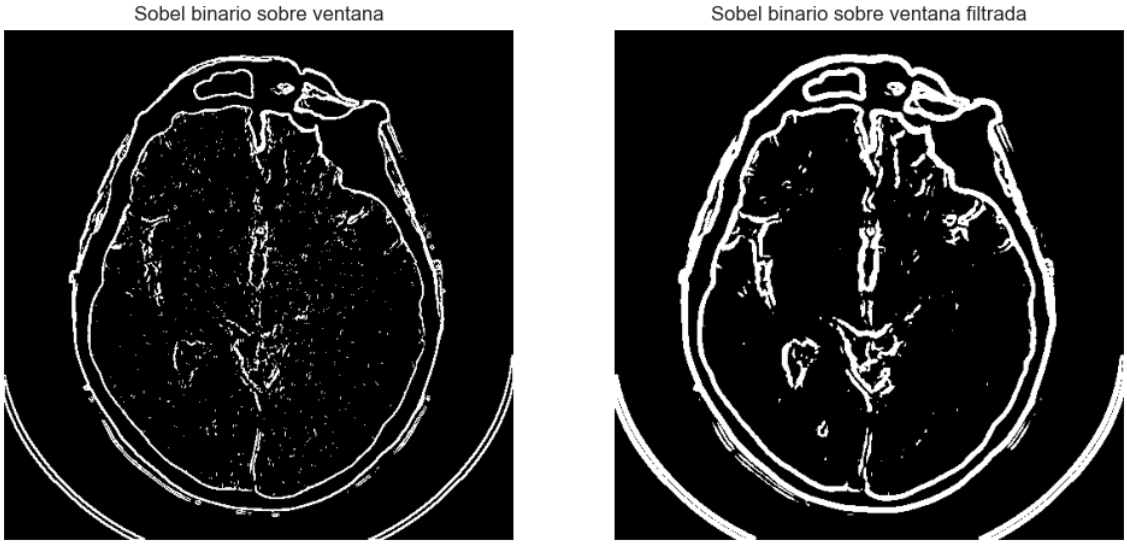

Figura 10: Comparación de binarización con Sobel.

4.3. Filtro de Canny

El filtro de Canny es usado para la detección de bordes y usa múltiples conceptos que lo hacen muy eficaz. Entre estas se encuentran:

- Cálculo del gradiente.
- Uso tanto de la magnitud como de la dirección del gradiente.
- Supresión de no máximos para adelgazar los bordes.
- Umbralización por histéresis, que clasifica los píxeles como fuertes, débiles o no relevantes.

Para su implementación se ha usado la librería OpenCv y se han optimizado los parámetros para su ideal visualización.

```
imagen_8bpp, bpp = imagen_bits(imagen, 8)
v_8bpp = (v * (255/4095)).astype(np.uint8)

imagen_original_8bpp_suavizada = cv2.GaussianBlur(imagen_8bpp, (5, 5), 1.0)
ventana_8bpp_suavizada = cv2.GaussianBlur(v_8bpp, (5, 5), 1.0)

imagen_original_8bpp_suavizada = cv2.convertScaleAbs(
imagen_original_8bpp_suavizada)
ventana_8bpp_suavizada = cv2.convertScaleAbs(ventana_8bpp_suavizada)

canny_original = cv2.Canny(imagen_original_8bpp_suavizada, 0, 40)
canny_ventana = cv2.Canny(ventana_8bpp_suavizada, 120, 250)
```

Listing 8: Código del filtro de Canny

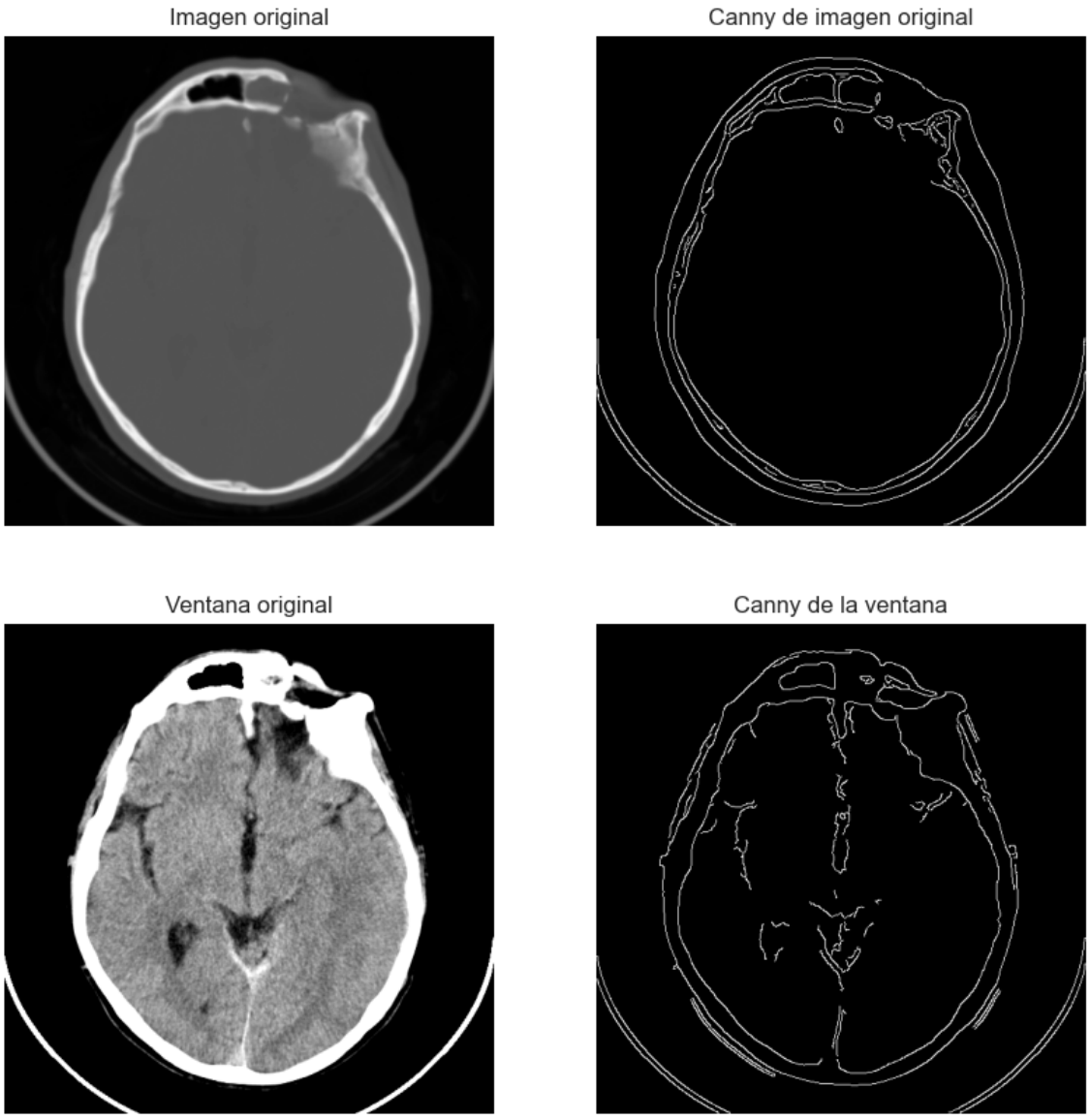

Figura 11: Filtro de canny en imagen original y ventana.

4.4. Conclusión

La binarización del filtro de Sobel, en concreto el de la ventana filtrada, es subjetivamente el mejor resultado obtenido. Incluso, se podría mejorar aplicando una técnica de thinning, ya que los bordes son algo gruesos.

Por otro lado, el filtro de Canny no obtiene un mal resultado pero faltan algunas estructuras de interés por resaltar y existen bordes que no cierran.

5. Ejercicio 3

5.1. Enunciado

Guardar todos los resultados de imágenes finales obtenidas para la memoria de prácticas. Use la función imwrite(...).

1. En formato png, con el mismo tipo de datos que la imagen original.
2. En formato jpg, tras reescalar la imagen al tipo uint8 (256 niveles de gris).
3. Adjuntar un fichero comprimido a la memoria con los ficheros de imagen obtenidos.

5.2. Guardado PNG y JPG

Este último paso es idéntico al realizado en la práctica anterior. Resumiendo, el entorno python utilizado (DataSpell), permite el guardado directamente en formato png, mientras que para guardar en formato jpg, se debe usar la función *plt.savefig()* e indicando jpg como formato.

6. Licencia y repositorio.

El código completo se encuentra en :mi repositorio de Github.
Licencia: MIT License