

DenoisingDeblurring_Tarea5

March 31, 2024

1 Tarea 5. Variational models for image processing

Guillermo Segura Gómez

1.1 Exercice 1. Implementation of basic operators

Define functions : - **forward-derivative-x** which implements the forward partial derivative with respect to x of a color image. - **forward-derivative-y** which implements the forward partial derivative with respect to y of a color image. - **backward-derivative-x** which implements the backward partial derivative with respect to x of a color image. - **backward-derivative-y** which implements the backward partial derivative with respect to y of a color image.

Tip : Use matrix operations instead of convolutions with kernels to implement these methods. It makes their executions faster.

From these operators define : - a function gradient which constructs the gradient of a color image using forward derivatives. - a function divergence which constructs the divergence operator using backward derivatives.

Podemos considerar una imagen como una función bidimensional $f(x, y)$, donde x y y son las coordenadas espaciales de cada píxel, y el valor de f en cualquier punto (x, y) da la intensidad del píxel en esa ubicación. Cuando trabajamos con imágenes en color, esta función es vectorial, ya que tenemos tres valores de intensidad en cada punto en el caso de RGB, uno para cada canal de color: rojo, verde y azul, por lo que tendríamos una función $f(\mathbb{R}^2) \rightarrow \mathbb{R}^3$

La **derivada parcial hacia adelante** mide el cambio en intensidad al moverse de un píxel a su vecino inmediato en la dirección que se especifica.

La derivada parcial hacia adelante de una función $f(x, y)$ con respecto a x se define como:

$$f_x^+(x, y) = \lim_{\Delta x \rightarrow 0} \frac{f(x + \Delta x, y) - f(x, y)}{\Delta x}$$

En una imagen digital no podemos tener límites infinitesimales, por lo que esta derivada se aproxima como la diferencia entre los valores de los píxeles adyacentes, lo que nos da:

$$f_x^+(x, y) \approx f(x + 1, y) - f(x, y)$$

Similarmente, la derivada parcial hacia adelante con respecto a y se define como:

$$f_y^+(x, y) \approx f(x, y + 1) - f(x, y)$$

La **derivada parcial hacia atrás** mide el cambio en intensidad al moverse en la dirección opuesta, es decir, del píxel actual al anterior.

La derivada parcial hacia atrás se define de manera similar, pero en lugar de mirar hacia el pixel siguiente, miramos hacia el anterior. Para x , se define como:

$$f_x^-(x, y) = \lim_{\Delta x \rightarrow 0} \frac{f(x, y) - f(x - \Delta x, y)}{\Delta x}$$

De la misma forma, para una imagen, esto se convierte en:

$$f_x^-(x, y) \approx f(x, y) - f(x - 1, y)$$

Y para y , la derivada hacia atrás es:

$$f_y^-(x, y) \approx f(x, y) - f(x, y - 1)$$

Podemos programar estas funciones de manera sencilla utilizando python. Como se revisó al principio de la discusión de este ejercicio, las imágenes pueden ser vistas como funciones de dos dimensiones. En este caso sería una matriz de $n \times n$ y cada elemento tendría tres elementos a su vez ya que es una imagen a color. Utilizamos la escritura de las matrices de *numpy* para poder desplazar las matrices y hacer las operaciones de las funciones.

```
[1]: import numpy as np

def forward_derivative_x(image):
    # Añadir cero al final en la dirección x para mantener las dimensiones originales
    padded_image = np.pad(image, ((0, 0), (0, 1), (0, 0)), mode='edge')
    # Calcular la derivada hacia adelante en x
    derivative = padded_image[:, 1:, :] - padded_image[:, :-1, :]
    return derivative

def forward_derivative_y(image):
    # Añadir cero al final en la dirección y para mantener las dimensiones originales
    padded_image = np.pad(image, ((0, 1), (0, 0), (0, 0)), mode='edge')
    # Calcular la derivada hacia adelante en y
    derivative = padded_image[1:, :, :] - padded_image[:-1, :, :]
    return derivative

def backward_derivative_x(image):
    # Añadir cero al inicio en la dirección x para mantener las dimensiones originales
    padded_image = np.pad(image, ((0, 0), (1, 0), (0, 0)), mode='edge')
```

```

# Calcular la derivada hacia atrás en x
derivative = padded_image[:, 1:, :] - padded_image[:, :-1, :]
return derivative

def backward_derivative_y(image):
    # Añadir cero al inicio en la dirección y para mantener las dimensiones originales
    padded_image = np.pad(image, ((1, 0), (0, 0), (0, 0)), mode='edge')
    # Calcular la derivada hacia atrás en y
    derivative = padded_image[1:, :, :] - padded_image[:-1, :, :]
    return derivative

```

Ahora probamos las funciones de derivación en alguna imagen.

```

[2]: import matplotlib.pyplot as plt
import glob
import cv2

# Path de la imagen
img_path = glob.glob('/Users/guillermo_sego/Desktop/Segundo Semestre/CVision/Tarea 5/img2.png')[0]

# Lee la imagen en color
image_bgr = cv2.imread(img_path, cv2.IMREAD_COLOR)
# Convierte la imagen a RGB
image_rgb = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2RGB)

# Aplicar las funciones de derivadas a la imagen RGB
fdx = forward_derivative_x(image_rgb)
fdy = forward_derivative_y(image_rgb)
bdx = backward_derivative_x(image_rgb)
bdy = backward_derivative_y(image_rgb)

# Imprimir resultados
fig, axes = plt.subplots(2, 3, figsize=(15, 10))
axes[0, 0].imshow(image_rgb)
axes[0, 0].set_title('Imagen Original')
axes[0, 0].axis('off')

axes[0, 1].imshow(np.abs(fdx).astype(np.uint8))
axes[0, 1].set_title('Derivada hacia adelante en X')
axes[0, 1].axis('off')

axes[0, 2].imshow(np.abs(fdy).astype(np.uint8))
axes[0, 2].set_title('Derivada hacia adelante en Y')
axes[0, 2].axis('off')

```

```

axes[1, 0].imshow(image_rgb)
axes[1, 0].set_title('Imagen Original')
axes[1, 0].axis('off')

axes[1, 1].imshow(np.abs(bdx).astype(np.uint8))
axes[1, 1].set_title('Derivada hacia atrás en X')
axes[1, 1].axis('off')

axes[1, 2].imshow(np.abs(bdy).astype(np.uint8))
axes[1, 2].set_title('Derivada hacia atrás en Y')
axes[1, 2].axis('off')

plt.show()

```



Ahora, definimos las funciones gradiente y divergencia en función de las derivadas que construimos.

El **gradiente** se define como:

$$\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right)$$

Donde $\frac{\partial f}{\partial x}$ y $\frac{\partial f}{\partial y}$ son las derivadas parciales de f con respecto a x e y , respectivamente.

En una imagen a color, que tiene múltiples canales, el gradiente se puede calcular por separado para cada canal, resultando en un vector gradiente para cada punto en cada canal.

La **divergencia** en un punto da una medida de cuánto un campo vectorial se está “expandiendo” desde ese punto, y se define como la suma de las derivadas parciales de las componentes del vector con respecto a sus dimensiones correspondientes. Para un campo vectorial $\mathbf{v} = (v_x, v_y)$, la divergencia se define como:

$$\text{div}(\mathbf{v}) = \nabla \cdot \mathbf{v} = \frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y}$$

En el caso del campo de gradientes de una imagen, esto significaría tomar el gradiente de los gradientes. Esencialmente, se está mirando cómo cambian los gradientes en la imagen, lo que puede indicar áreas donde los cambios en la intensidad aumentan o disminuyen rápidamente.

```
[3]: def gradient(image):
    # Calcula las derivadas hacia adelante en x e y
    grad_x = forward_derivative_x(image)
    grad_y = forward_derivative_y(image)

    # Combina las derivadas para formar el vector gradiente
    gradient = np.stack((grad_x, grad_y), axis=-1)

    return gradient

def divergence(gradient):
    # Extrae los componentes del gradiente en x e y
    grad_x = gradient[:, 0] # Todos los componentes x del gradiente
    grad_y = gradient[:, 1] # Todos los componentes y del gradiente

    # Calcula las derivadas hacia atrás en x e y para cada componente del
    # gradiente
    div_x = backward_derivative_x(grad_x)
    div_y = backward_derivative_y(grad_y)

    # La divergencia es la suma de estas dos derivadas
    div = div_x + div_y

    return div
```

```
[4]: # Calcula el gradiente de la imagen
img_gradient = gradient(image_rgb)

# Calcula la divergencia del gradiente
img_divergence = divergence(img_gradient)

# Imprimir resultados
plt.figure(figsize=(15, 10))
```

```

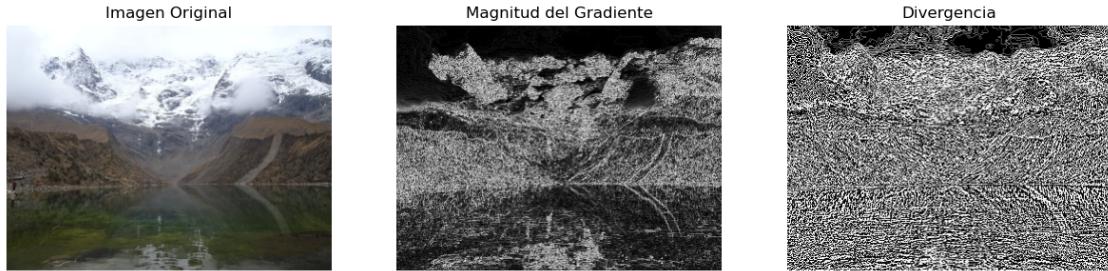
# Imagen original
plt.subplot(1, 3, 1)
plt.imshow(image_rgb)
plt.title('Imagen Original')
plt.axis('off')

# Magnitud del gradiente
gradient_magnitude = np.sqrt(np.sum(img_gradient**2, axis=-1))
plt.subplot(1, 3, 2)
plt.imshow(np.mean(gradient_magnitude, axis=-1), cmap='gray')
plt.title('Magnitud del Gradiente')
plt.axis('off')

# Divergencia
plt.subplot(1, 3, 3)
plt.imshow(np.mean(img_divergence, axis=-1), cmap='gray')
plt.title('Divergencia')
plt.axis('off')

plt.show()

```



1.2 Exercice 2 (Joint denoising and deblurring)

img1-degradation1.png is the result of applying Gaussian blur of standard deviation 2 and additive white Gaussian noise of standard deviation 5 to **img1.png**.

1. Implement the Algorithm 3 seen in class by using the operators constructed in Exercise
2. Use **fast-gaussian-convolution** to implement the component-wise convolution with a Gaussian kernel.
3. Apply the algorithm to **img1-degradation1.png** with the following parameters : $\eta = 0.0001$, $\epsilon = 0.001$, $\alpha = 0.01$, $K = 15000$. You will test different values of the regularization parameter λ and select the one providing the best PSNR with respect to **img1.png**.

1.2.1 Eliminación del ruido

Revisando las notas de la clase, en el apartado de restauración de imágenes podemos trabajar con la eliminación de ruido y desenfoque. Es común trabajar con imágenes que poseen “ruido” y “desenfoque”. El ruido puede deberse a la calidad de la cámara, las condiciones de iluminación, etc., mientras que el desenfoque puede ocurrir por movimiento o enfoque incorrecto durante la captura de la imagen. Nuestro objetivo es “limpiar” estas imágenes para recuperar la imagen original \underline{u} tanto como sea posible, eliminando el ruido y corrigiendo el desenfoque. Esto se conoce como **denoising** eliminación de ruido y **deblurring** eliminación de desenfoque.

Para poder plantear esto, comenzamos con un modelo simple que describe cómo se ve afectada la imagen original \underline{u} por el ruido n , dando como resultado la imagen degradada que observamos u_0 . En el caso más simple de **solo denoising**, nuestro modelo de degradación se simplifica a la siguiente relación:

$$u_0 = \underline{u} + n,$$

lo que significa que el operador de degradación A es simplemente la identidad, y la imagen observada u_0 es la suma de la imagen original sin ruido \underline{u} y el ruido n . En otras palabras, el ruido se suma simplemente a nuestra imagen original.

Para “limpiar” la imagen, utilizamos modelos matemáticos que nos ayudan a encontrar la versión más probable de la imagen original sin ruido. Estos modelos tratan de encontrar una imagen u que sea lo más cercana posible a u_0 (en términos de fidelidad a los datos) pero que también sea suave (para eliminar el ruido). Esto se logra mediante un balance, controlado por un parámetro λ , entre hacer que u se parezca a u_0 y mantener u suave. Tenemos dos variantes de estos modelos, diferenciándose en cómo definen la “suavidad” de la imagen.

El primero se describe como la solución al problema de minimización:

$$\arg \min_u \frac{1}{2} \int_{\Omega} \|u(x) - u_0(x)\|^2 + \lambda \|\nabla u(x)\|^2 dx,$$

donde el término $\frac{1}{2} \int_{\Omega} \|u(x) - u_0(x)\|^2 dx$ mide la fidelidad de la imagen restaurada u con respecto a la imagen observada u_0 , y $\lambda \|\nabla u(x)\|^2$ es un término de regularización que favorece soluciones suaves, con λ controlando el balance entre ambos.

El segundo modelo introduce un término de regularización ligeramente diferente y se define como:

$$\arg \min_u \frac{1}{2} \int_{\Omega} \|u(x) - u_0(x)\|^2 + \lambda \sqrt{\eta + \|\nabla u(x)\|^2} dx,$$

donde η es un parámetro pequeño que asegura que la expresión bajo la raíz cuadrada nunca sea cero, evitando singularidades en la derivada.

Para encontrar la imagen u que minimiza nuestras funciones objetivo (energías), calculamos el “gradiente de energía”, que nos dice en qué dirección cambiar u para hacerla más similar a la imagen original mientras reducimos el ruido. Luego aplicamos un método de minimización como el descenso de gradiente para ajustar u paso a paso hasta que encontramos la mejor solución.

El gradiente de la energía $E(u)$ para el primer modelo se da por:

$$\nabla E(u) = u - u_0 - \lambda \Delta u,$$

donde Δu es el laplaciano de u , representando la suma de las segundas derivadas parciales, que indica la dispersión de u .

Para el segundo modelo, el gradiente de $E_\eta(u)$ es:

$$\nabla E_\eta(u) = u - u_0 - \lambda \operatorname{div} \left(\frac{\nabla u}{\sqrt{\eta + \|\nabla u\|^2}} \right),$$

donde div es el operador de divergencia, que en este contexto, toma el gradiente de u y lo normaliza, dandonos una dirección en la que actualizar u para minimizar $E_\eta(u)$.

1.2.2 Eliminación de ruido y desenfoque

Cuando también queremos corregir el desenfoque, nuestro modelo de degradación se vuelve más complejo, incluyendo un operador A que representa cómo la imagen se desenfoca (a menudo mediante una convolución con un “kernel” o filtro específico). Nuestro objetivo sigue siendo el mismo: encontrar la mejor u , pero esta vez también necesitamos deshacer el efecto del desenfoque representado por A . Entonces, nuestro modelo de degradación toma la forma:

$$u_0 = A\underline{u} + n,$$

donde A es un operador que representa una convolución con algún núcleo específico. Basándonos en los resultados de la eliminación de ruido, preferimos el siguiente modelo variacional para nuestros experimentos:

$$E_\eta(u) = \frac{1}{2} \int_{\Omega} \|Au(x) - u_0(x)\|^2 + \lambda \sqrt{\eta + \|\nabla u(x)\|^2} dx,$$

y el gradiente de esta energía en u es:

$$\nabla E_\eta(u) = A^T (Au - u_0) - \lambda \operatorname{div} \left(\frac{\nabla u}{\sqrt{\eta + \|\nabla u\|^2}} \right),$$

donde A^T es el operador adjunto de A . Esto nos permite aplicar un algoritmo de descenso por gradiente (algoritmo 3) para encontrar numéricamente la solución del modelo variacional, es decir, restaurar la imagen u minimizando $E_\eta(u)$.

Algorithm 3 Joined Deblurring + denoising: Model 2

Initialization: Take $u^{(0)} = u_0$, and choose $\epsilon > 0$ the stopping criteria, $\alpha > 0$ the step size, and K the maximum number of iterations.

while $\sqrt{\operatorname{MSE}(u^{(k+1)}, u^{(k)})} > \epsilon$ and $k < K$

do

$$u^{(k+1)} = u^{(k)} - \alpha \left(A^* (Au^{(k)} - u_0) - \lambda \operatorname{div} \left(\frac{\nabla u^{(k)}}{\sqrt{\eta + \|\nabla u^{(k)}\|^2}} \right) \right)$$

end while

Para este caso consideramos el modelo

$$u_0 = A\underline{u} + n,$$

donde $A = G^*$ es un blur gaussiano de varianza σ_b y n un ruido blanco gaussiano de varianza σ_n .

El objetivo de este ejercicio consiste en implementar el algoritmo 3 para eliminación de ruido y desenfoque. Lo primero, definimos la función A que vamos a utilizar. Se nos proporciona la convolución rápida gaussiana en la función adjunta al ejercicio.

(Revisar capítulo 2 del libro de *Paragios. Handbook of mathematical models in computer vision* [1] y notas de clase.)

Es importante destacar que para el caso de $A = G_{\sigma^*}$ la convolución $A = A^*$ es simétrica respecto a su adjunta, por lo que solo necesitamos la siguiente función para implementar el algoritmo.

```
[5]: from scipy import signal

def fast_gaussian_convolution(u, sigma):
    # Crear el kernel gaussiano
    gaussian_kernel_1d_1 = np.expand_dims(signal.gaussian(u.shape[0], sigma), axis=1)
    gaussian_kernel_1d_2 = np.expand_dims(signal.gaussian(u.shape[1], sigma), axis=1)
    gaussian_kernel_2d = np.matmul(gaussian_kernel_1d_1, np.
        transpose(gaussian_kernel_1d_2))
    gaussian_kernel_2d /= np.sum(gaussian_kernel_2d) # Normalizar el kernel

    # Verificar si la imagen es en escala de grises o en color
    if u.ndim == 3: # Imagen en color
        result = np.zeros_like(u)
        for i in range(3): # Aplicar la convolución a cada canal
            result[:, :, i] = np.real(np.fft.fftshift(np.fft.ifft2(np.fft.
                ifft2(u[:, :, i]) * np.fft.fft2(gaussian_kernel_2d))))
    else: # Imagen en escala de grises
        result = np.real(np.fft.fftshift(np.fft.ifft2(np.fft.fft2(u) * np.fft.
            ifft2(gaussian_kernel_2d)))))

    return result
```

Algoritmo (3) para deblurring y denoising combinados.

```
[6]: from tqdm import tqdm # Para barras de progreso

def DeblurringDenoising(A, div, grad, u0, lambdak, eta, alpha, epsilon, nMax, sigma,
    delta_mse_threshold=1e-6):
    # Inicialización
    uk = np.array(u0)
    mse_prev = None # Almacenar el valor de MSE de la iteración anterior
```

```

# Crear la barra de progreso de tqdm
pbar = tqdm(total=nMax, desc="Deblurring and Denoising")

# Barra de progreso en el bucle for
for k in range(nMax):
    # Aplicar A y A^* usando fast_gaussian_convolution
    Au_k = A(uk, sigma)
    Astar_diff = A(Au_k - u0, sigma)

    # Denominador del término de la divergencia
    deno = eta + np.linalg.norm(grad(uk), axis=-1)**2

    # Añadir una dimensión extra a 'deno' para hacerla compatible con la
    # forma de 'grad(uk)'
    deno_expanded = np.expand_dims(deno, axis=-1) # Añade una nueva
    # dimensión al final

    # Realizar la división
    aux = grad(uk) / deno_expanded

    # Calcular el gradiente de la energía en uk
    gk = Astar_diff - lambdak * div(aux)

    # Actualizar uk para la siguiente iteración
    uk_next = uk - alpha * gk

    # print(uk_next)
    # plt.imshow(uk_next.astype(np.uint8))
    # break

    # Calcular MSE entre iteraciones sucesivas para el criterio de parada
    mse = np.mean((uk_next - uk)**2)

    # Verificar la condición de parada basada en el cambio de MSE
    if mse_prev is not None and abs(mse - mse_prev) < delta_mse_threshold:
        print(f"\nMSE no cambio en la iteración {k+1}")
        pbar.close()
        return uk_next, k + 1, False

    mse_prev = mse # Actualizar el valor anterior de MSE para la próxima
    # iteración

    # Actualizar la barra de progreso con el MSE actual
    pbar.set_description(f"Deblurring and Denoising Iter: {k+1}, MSE: {mse:.5f}")
    pbar.update(1) # Avanzar la barra de progreso en una unidad

```

```

# Verificar la condición de parada
if np.sqrt(mse) < epsilon or np.sqrt(mse) > 10:
    print(f"Convergencia alcanzada en la iteración {k+1}")
    pbar.close()
    return uk_next, k + 1, True

uk = uk_next # Preparar para la siguiente iteración

pbar.close()
return uk, nMax, False

```

Necesitamos además definir una función para poder calcular el PSNE.

El PSNR, o *Peak Signal-to-Noise Ratio* (Relación Pico Señal-Ruido), es una métrica utilizada para medir la calidad de reconstrucción de imágenes y videos comprimidos en comparación con sus originales sin comprimir o no distorsionados.

El PSNR se define en términos del MSE (Mean Squared Error, o Error Cuadrático Medio), que es la media de los cuadrados de las diferencias entre los valores de píxeles correspondientes de la imagen original y la imagen distorsionada o reconstruida [2]. La fórmula para calcular el PSNR es:

$$\text{PSNR} = 10 \cdot \log_{10} \left(\frac{\text{MAX}^2}{\text{MSE}} \right)$$

donde: - **MAX** es el valor máximo posible de un píxel en la imagen. Para imágenes de 8 bits por canal, este valor es típicamente 255. - **MSE** se calcula como:

$$\text{MSE} = \frac{1}{mn} \sum_{i=1}^m \sum_{j=1}^n [I(i, j) - K(i, j)]^2$$

donde I es la imagen original, K es la imagen distorsionada o reconstruida, y m y n son las dimensiones de la imagen.

El PSNR se expresa en decibelios (dB) y proporciona una medida de la relación entre el máximo poder posible de una señal y el poder del ruido corruptor que afecta la calidad de su representación. Cuanto mayor es el valor del PSNR, mejor es la calidad de la imagen reconstruida o comprimida en comparación con la original.

```
[7]: # Función para calcular el PSNR
def calculate_psnr(img1, img2):
    mse = np.mean((img1 - img2) ** 2)
    if mse == 0:
        return 100
    MAX_PIXEL_VALUE = 255.0
    return 20 * np.log10(MAX_PIXEL_VALUE / np.sqrt(mse))
```

Probamos la función para la imagen `img1_degradation.png`

```
[8]: # Define los parámetros del algoritmo
eta = 0.0001
```

```

epsilon = 0.001
alpha = 0.01
nMax = 3000
lambdak_values = [0.01, 0.1, 1, 10, 100] # Valores a probar
sigma = 5

# Imagen
u_path = glob.glob('/Users/guillermo_sego/Desktop/Segundo Semestre/CVision/
↳Tarea 5/img1_degradation1.png')[0]

# Lee la imagen en color
u_bgr = cv2.imread(u_path, cv2.IMREAD_COLOR)
# Convierte la imagen a RGB
u_rgb = cv2.cvtColor(u_bgr, cv2.COLOR_BGR2RGB)

# Configurar los subplots
fig, axs = plt.subplots(1, len(lambdak_values) + 1, figsize=(15, 5))
axs[0].imshow(u_rgb)
axs[0].set_title('Imagen original')
axs[0].axis('off')

# Aplicación del algoritmo para cada valor de lambdak y cálculo de PSNR
for i, lambdak in enumerate(lambdak_values, start=1):
    restored_img, k, converged = DeblurringDenoising(
        fast_gaussian_convolution, divergence, gradient, u_rgb, lambdak, eta,
        ↳alpha, epsilon, nMax, sigma
    )
    psnr = calculate_psnr(u_rgb, restored_img)
    print(f"={lambdak}, PSNR: {psnr:.2f}")

    # Mostrar la imagen restaurada y el PSNR
    axs[i].imshow(restored_img.astype(np.uint8))
    axs[i].set_title(f'={lambdak}\nPSNR: {psnr:.2f}')
    axs[i].axis('off')

plt.tight_layout()
plt.show()

```

```

Deblurring and Denoising: 0% | 0/3000 [00:00<?, ?it/s]/var/folders/9
t/s_zg1kn954126k1btcnp7lfm0000gn/T/ipykernel_30077/2053258905.py:5:
DeprecationWarning: Importing gaussian from 'scipy.signal' is deprecated and
will raise an error in SciPy 1.13.0. Please use 'scipy.signal.windows.gaussian'
or the convenience function 'scipy.signal.get_window' instead.
    gaussian_kernel_1d_1 = np.expand_dims(signal.gaussian(u.shape[0], sigma),
axis=1)
/var/folders/9t/s_zg1kn954126k1btcnp7lfm0000gn/T/ipykernel_30077/2053258905.py:6
: DeprecationWarning: Importing gaussian from 'scipy.signal' is deprecated and

```

```
will raise an error in SciPy 1.13.0. Please use 'scipy.signal.windows.gaussian'  
or the convenience function 'scipy.signal.get_window' instead.
```

```
gaussian_kernel_1d_2 = np.expand_dims(signal.gaussian(u.shape[1], sigma),  
axis=1)  
Deblurring and Denoising Iter: 417, MSE: 0.00038: 14% | 417/3000  
[00:40<04:08, 10.38it/s]
```

MSE no cambio en la iteración 418
=0.01, PSNR: 26.50

```
Deblurring and Denoising Iter: 290, MSE: 0.00057: 10% | 290/3000  
[00:26<04:04, 11.08it/s]
```

MSE no cambio en la iteración 291
=0.1, PSNR: 28.55

```
Deblurring and Denoising Iter: 13, MSE: 0.00280: 0% | 13/3000  
[00:01<04:51, 10.25it/s]
```

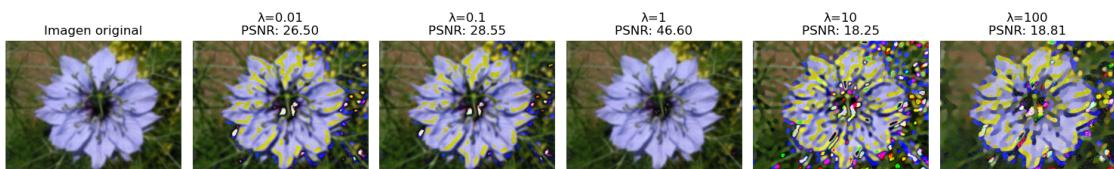
MSE no cambio en la iteración 14
=1, PSNR: 46.60

```
Deblurring and Denoising Iter: 3000, MSE: 0.12685: 100% | 3000/3000  
[04:39<00:00, 10.74it/s]
```

=10, PSNR: 18.25

```
Deblurring and Denoising Iter: 3000, MSE: 1.71777: 100% | 3000/3000  
[04:30<00:00, 11.09it/s]
```

=100, PSNR: 18.81



El valor que mejor ajustó la imagen fue el de $\lambda = 1$. Para valores altos de λ encontramos que se disminuye la exactitud ya que el PSNR calculado es menor. Es útil también revisar algún otro valor de sigma para ver si podemos lograr mejores resultados. El valor de $\sigma = 5$, se escogió ya que el algoritmo encontraba mejores resultados partiendo de una imagen con desenfoque. Esto por lo revisando en la discusión inicial, comenzamos con una imagen desenfocada y tratamos de ajustarla minimizando el gradiente de energía. De cierta forma, haciendo mas evidente el desenfoque, se lograba una convergencia y una disminución del MSE mucho mejor.

Ahora probamos el algoritmo con la imagen **img1.png** con el mejor valor de lambda seleccionado.

```
[9]: # Define los parámetros del algoritmo
eta = 0.0001
epsilon = 0.001
alpha = 0.01
nMax = 3000
lambda_dak_values = [1]
sigma = 5

# Imagen
u_path = glob.glob('/Users/guillermo_sego/Desktop/Segundo Semestre/CVision/
↳Tarea 5/img1.png')[0]

# Lee la imagen en color
u_bgr = cv2.imread(u_path, cv2.IMREAD_COLOR)
# Convierte la imagen a RGB
u_rgb = cv2.cvtColor(u_bgr, cv2.COLOR_BGR2RGB)

# Configurar los subplots
fig, axs = plt.subplots(1, len(lambda_dak_values) + 1, figsize=(15, 5))
axs[0].imshow(u_rgb)
axs[0].set_title('Imagen original')
axs[0].axis('off')

# Aplicación del algoritmo para cada valor de lambda_dak y cálculo de PSNR
for i, lambda_dak in enumerate(lambda_dak_values, start=1):
    restored_img, k, converged = DeblurringDenoising(
        fast_gaussian_convolution, divergence, gradient, u_rgb, lambda_dak, eta,
        ↳alpha, epsilon, nMax, sigma
    )
    psnr = calculate_psnr(u_rgb, restored_img)
    print(f"={lambda_dak}, PSNR: {psnr:.2f}")

    # Mostrar la imagen restaurada y el PSNR
    axs[i].imshow(restored_img.astype(np.uint8))
    axs[i].set_title(f'={lambda_dak}\nPSNR: {psnr:.2f}')
    axs[i].axis('off')

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

```
Deblurring and Denoising: 0% | 0/3000 [00:00<?, ?it/s]/var/folders/9
t/s_zg1kn954126k1btcnp71fm0000gn/T/ipykernel_30077/2053258905.py:5:
DeprecationWarning: Importing gaussian from 'scipy.signal' is deprecated and
will raise an error in SciPy 1.13.0. Please use 'scipy.signal.windows.gaussian'
or the convenience function 'scipy.signal.get_window' instead.
    gaussian_kernel_1d_1 = np.expand_dims(signal.gaussian(u.shape[0], sigma),
axis=1)
/var/folders/9t/s_zg1kn954126k1btcnp71fm0000gn/T/ipykernel_30077/2053258905.py:6
```

```

: DeprecationWarning: Importing gaussian from 'scipy.signal' is deprecated and
will raise an error in SciPy 1.13.0. Please use 'scipy.signal.windows.gaussian'
or the convenience function 'scipy.signal.get_window' instead.
gaussian_kernel_1d_2 = np.expand_dims(signal.gaussian(u.shape[1], sigma),
axis=1)
Deblurring and Denoising Iter: 36, MSE: 0.00363: 1%| | 36/3000
[00:03<04:36, 10.70it/s]

```

MSE no cambio en la iteración 37
 $\lambda=1$, PSNR: 42.27



A mi parecer, mejoró la calidad de la imagen. :D (Salvo por los puntos azules, pero lo demás se me menos degradada)

1.3 Exersice 3 (Impulse noise removal)

We would like to test the following model

$$\arg \min_u \frac{1}{2} \int_{\Omega} \|u(x) - u_0(x)\|^2 + \lambda \|\nabla u(x)\|^2 dx, \quad (0.1)$$

to remove impulse noise in the color image $u_0 : \Omega \subset \mathbb{R}^2 \rightarrow \mathbb{R}^3$.

1. Write a differentiable approximation of the functional in (0.1) and determine its gradient.
2. Adapt the Algorithm 2 seen in class to the minimization problem (0.1).
3. Apply the algorithm to **img1—degradation2.png** with the following parameters : $\lambda = 1$, $\eta = 0.0001$, $\epsilon = 0.001$, $\alpha = 0.01$, $K = 15000$.
4. What PSNR value do you obtain ? (with respect to img1.png).
5. Test Algorithm 2 with the following parameters : $\lambda = 100$, $\eta = 0.0001$, $\epsilon = 0.001$, $\alpha = 0.0001$, $K = 15000$. Which algorithm is the best for impulse noise removal ?

El valor del funcional 0.1 es bastante similar al revisado en el modelo 1 del inciso anterior.

$$\arg \min_u \frac{1}{2} \int_{\Omega} \|u(x) - u_0(x)\|^2 + \lambda \|\nabla u(x)\|^2 dx,$$

donde el gradiente de energía tenía la forma

$$\nabla E(u) = u - u_0 - \lambda \Delta u,$$

La diferencia entre ambos funcionales radica en que uno de ellos posee términos cuadráticos y el otro no. Para trabajar con este funcional, necesitamos una versión diferenciable del mismo. La dificultad esta en la no diferenciabilidad de los términos $\|u(x) - u_0(x)\|$ y $\|\nabla u(x)\|$ en cero. Una aproximación que podemos realizar es agregar un pequeño término η positivo para asegurar la diferenciabilidad:

$$\sqrt{\eta + \|u(x) - u_0(x)\|^2} \quad \text{y} \quad \sqrt{\eta + \|\nabla u(x)\|^2},$$

Por lo tanto, el funcional aproximado y diferenciable sería:

$$E(u) \approx \int_{\Omega} \sqrt{\eta + \frac{1}{2} \|u(x) - u_0(x)\|^2 + \lambda \sqrt{\eta + \|\nabla u(x)\|^2}} dx.$$

Se sigue determinando el gradiente del funcional aproximado respecto a u .

Para la primera parte, el gradiente sería:

$$\frac{u(x) - u_0(x)}{\sqrt{\eta + \frac{1}{2} \|u(x) - u_0(x)\|^2}},$$

y para la regularización de variación total, el gradiente de $\sqrt{\eta + \|\nabla u(x)\|^2}$ es más complejo y se relaciona con la divergencia del gradiente normalizado de u , que puede escribirse como:

$$\lambda \operatorname{div} \left(\frac{\nabla u(x)}{\sqrt{\eta + \|\nabla u(x)\|^2}} \right),$$

Por lo tanto, para el funcional 0.1, el gradiente sería

$$\nabla E(u) = \frac{u(x) - u_0(x)}{\sqrt{\eta + \frac{1}{2} \|u(x) - u_0(x)\|^2}} + \lambda \operatorname{div} \left(\frac{\nabla u(x)}{\sqrt{\eta + \|\nabla u(x)\|^2}} \right)$$

Con esto podemos construir el algoritmo 2 adaptado al gradiente calculado.

Algorithm 2 Denoising (Model 2)

Initialization: Choose $u^{(0)} = u_0$, $\epsilon > 0$ for the stopping criteria, a step size $\alpha > 0$, a number of max iterations K

```

while  $\sqrt{\text{MSE}(u^{(k+1)}, u^{(k)})} > \epsilon$  and  $k < K$ 
do
 $u^{(k+1)} = u^{(k)} - \alpha \nabla E(u)$ 
end while

```

```

[23]: def Denoising(div, grad, u0, lambdak, eta, alpha, epsilon, nMax, sigma,
    ↪delta_mse_threshold=1e-6):
    # Inicialización
    uk = np.array(u0)
    mse_prev = None # Almacenar el valor de MSE de la iteración anterior

    # Crear la barra de progreso de tqdm
    pbar = tqdm(total=nMax, desc="Denoising")

    # Barra de progreso en el bucle for
    for k in range(nMax):

        # Denominador del término de la divergencia
        deno = eta + np.linalg.norm(grad(uk), axis=-1)**2

        # Añadir una dimensión extra a 'deno' para hacerla compatible con la
        ↪forma de 'grad(uk)'
        deno_expanded = np.expand_dims(deno, axis=-1) # Añade una nueva
        ↪dimensión al final

        # Realizar la división
        aux = grad(uk) / deno_expanded

        # Calcular el gradiente de la energía en uk
        gk = (uk - u0) / np.sqrt(eta + (uk - u0)**2) + lambdak * div(aux)

        # Actualizar uk para la siguiente iteración
        uk_next = uk - alpha * gk
        u0 = uk

        # print(uk_next)
        # plt.imshow(uk_next.astype(np.uint8))
        # break

        # Calcular MSE entre iteraciones sucesivas para el criterio de parada
        mse = np.mean((uk_next - uk)**2)

        # Verificar la condición de parada basada en el cambio de MSE
        if mse_prev is not None and abs(mse - mse_prev) < delta_mse_threshold:
            print(f"\nMSE no cambio en la iteración {k+1}")
            pbar.close()

```

```

        return uk_next, k + 1, False

    mse_prev = mse # Actualizar el valor anterior de MSE para la próxima
    ↵iteración

    # Actualizar la barra de progreso con el MSE actual
    pbar.set_description(f"Denoising Iter: {k+1}, MSE: {mse:.5f}")
    pbar.update(1) # Avanzar la barra de progreso en una unidad

    # Verificar la condición de parada
    if np.sqrt(mse) < epsilon or np.sqrt(mse) > 10:
        print(f"Convergencia alcanzada en la iteración {k+1}")
        pbar.close()
        return uk_next, k + 1, True

    uk = uk_next # Preparar para la siguiente iteración

    pbar.close()
    return uk, nMax, False

```

Ahora probamos el algoritmo con la imagen `img1_degradation2.png`

```
[24]: # Define los parámetros del algoritmo
eta = 0.0001
epsilon = 0.001
alpha = 0.01
nMax = 3000
lambdak_values = [1.0] # Valores a probar
sigma = 5

# Imagen
u_path = glob.glob('/Users/guillermo_sego/Desktop/Segundo Semestre/CVision/
    ↵Tarea 5/img1_degradation2.png')[0]

# Lee la imagen en color
u_bgr = cv2.imread(u_path, cv2.IMREAD_COLOR)
# Convierte la imagen a RGB
u_rgb = cv2.cvtColor(u_bgr, cv2.COLOR_BGR2RGB)

# Configurar los subplots
fig, axs = plt.subplots(1, len(lambdak_values) + 1, figsize=(15, 5))
axs[0].imshow(u_rgb)
axs[0].set_title('Imagen original')
axs[0].axis('off')

# Aplicación del algoritmo para cada valor de lambdak y cálculo de PSNR
for i, lambdak in enumerate(lambdak_values, start=1):
    u_denoised, nMax, error = denoising(u_path, lambdak, eta, epsilon, alpha, sigma)
    psnr = calculate_psnr(u_bgr, u_denoised)
    print(f'PSNR para lambdak {lambdak}: {psnr:.2f} dB')
```

```

restored_img, k, converged = Denoising(
    divergence, gradient, u_rgb, lambdak, eta, alpha, epsilon, nMax, sigma
)
psnr = calculate_psnr(u_rgb, restored_img)
print(f"λ={lambdak}, PSNR: {psnr:.2f}")

# Mostrar la imagen restaurada y el PSNR
axs[i].imshow(restored_img.astype(np.uint8))
axs[i].set_title(f"λ={lambdak}\nPSNR: {psnr:.2f}")
axs[i].axis('off')

plt.tight_layout()
plt.show()

```

Denoising Iter: 5, MSE: 0.00001: 0% | 5/3000 [00:00<02:21, 21.21it/s]

MSE no cambio en la iteración 6
=1.0, PSNR: 71.62



El valor calculado del PSNR es bastante alto, mucho mejor que los resultados obtenidos en el algoritmo pasado. Ahora probamos el algoritmo con respecto a la imagen **img1.png** para comparar los valores del PSNR.

```
[12]: # Define los parámetros del algoritmo
eta = 0.0001
epsilon = 0.001
alpha = 0.0001
nMax = 3000
lambdak_values = [100.0] # Valores a probar
sigma = 5

# Imagen
```

```

u_path = glob.glob('/Users/guillermo_sego/Desktop/Segundo Semestre/CVision/
↳Tarea 5/img1.png')[0]

# Lee la imagen en color
u_bgr = cv2.imread(u_path, cv2.IMREAD_COLOR)
# Convierte la imagen a RGB
u_rgb = cv2.cvtColor(u_bgr, cv2.COLOR_BGR2RGB)

# Configurar los subplots
fig, axs = plt.subplots(1, len(lambdak_values) + 1, figsize=(15, 5))
axs[0].imshow(u_rgb)
axs[0].set_title('Imagen original')
axs[0].axis('off')

# Aplicación del algoritmo para cada valor de lambdak y cálculo de PSNR
for i, lambdak in enumerate(lambdak_values, start=1):
    restored_img, k, converged = Denoising(
        divergence, gradient, u_rgb, lambdak, eta, alpha, epsilon, nMax, sigma
    )
    psnr = calculate_psnr(u_rgb, restored_img)
    print(f"={lambdak}, PSNR: {psnr:.2f}")

    # Mostrar la imagen restaurada y el PSNR
    axs[i].imshow(restored_img.astype(np.uint8))
    axs[i].set_title(f'={lambdak}\nPSNR: {psnr:.2f}')
    axs[i].axis('off')

plt.tight_layout()
plt.show()

```

Denoising Iter: 6, MSE: 0.00002: 0% | 6/3000 [00:00<01:22,
36.07it/s]

MSE no cambio en la iteración 7
=100.0, PSNR: 69.02



El algoritmo 2 con el gradiente modificado obtuvo un PSNR mas alto que el algoritmo 3. La imagen también se ve mucho mejor visualmente. Por lo tanto el algoritmo que funcionó mejor para la tarea de eliminación de ruido fue el algoritmo 2 **Denoising**.

1.4 Exercice 4 (Contrast processing)

Let $u_0 : \Omega \subset \mathbb{R}^2 \rightarrow \mathbb{R}^n$ be a continuous function, and the functional E given by

$$E(u) = \frac{1}{2} \int_{\Omega} \|u(x) - u_0(x)\|^2 dx - \frac{\lambda}{4} \int_{\Omega} \int_{\Omega} w(x, y) \|u(x) - u(y)\|^2 dxdy, \quad (0.2)$$

where w is a normalized Gaussian kernel of standard deviation σ .

1. Show that the gradient $\nabla E(u)$ of E at u is

$$\nabla E(u) = u - u_0 - \lambda(u - w * u),$$

where $*$ stands for the component-wise convolution.

Para resolver este ejercicio, necesitamos calcular el gradiente $\nabla E(u)$ de la funcional $E(u)$.

La funcional $E(u)$ tiene dos términos:

1. El primer término es un término de fidelidad, $\frac{1}{2} \int_{\Omega} \|u(x) - u_0(x)\|^2 dx$, que mide la diferencia entre la función u y una función dada u_0 .
2. El segundo término es un término de regularización, $-\frac{\lambda}{4} \int_{\Omega} \int_{\Omega} w(x, y) \|u(x) - u(y)\|^2 dxdy$, que mide la diferencia entre los valores de u en dos puntos distintos x y y , ponderada por una función de peso $w(x, y)$.

Para calcular el gradiente de $E(u)$, derivaremos cada término por separado con respecto a u y luego sumaremos los resultados.

El gradiente del primer término, $\frac{1}{2} \int_{\Omega} \|u(x) - u_0(x)\|^2 dx$, con respecto a u es simplemente $u - u_0$.

El segundo término es más complicado debido a la presencia de la integral doble y la función de peso $w(x, y)$. La derivación de este término respecto a u implica aplicar el cálculo de variaciones, considerando la diferencia $u(x) - u(y)$ y su relación con la convolución $w * u$, donde $*$ denota la convolución componente a componente. La convolución $w * u$ en un punto x puede interpretarse como una suma ponderada de los valores de u alrededor de x , donde los pesos están dados por w . Podemos ver el resultado de esta integral en la p 27 de libro [1].

Juntando ambos términos y considerando el signo negativo en el segundo término, el gradiente de $E(u)$ es:

$$\nabla E(u) = u - u_0 - \lambda(u - w * u),$$

donde $\lambda(u - w * u)$ representa el efecto de la regularización que penaliza las grandes diferencias entre u y su versión suavizada $w * u$.

2. Apply the gradient descent algorithm associated to the functional (0.2) in order to enhance simultaneously the local and the global contrasts of img2.png.

Parameters of the models : - Local contrast enhancement model : $\lambda_1 = 0.5$ and $\sigma_1 = 5$. - Global contrast enhancement model : $\lambda_2 = 0.75$ and $\sigma_2 = 3000$.

Parameters of the algorithm : - Stopping criteria : MSE between two consecutive images is less than $\epsilon = 0.001$ - Step size : $\alpha = 0.01$ - Maximum number of iterations : $K = 15000$

Tip : Use the function fast_gaussian-convolution to implement the component-wise convolution with a Gaussian kernel.

Para realizar esto, nos basamos en la función construida en el ejercicio 2.

```
[25]: def Contrast(w, u0, lambdak, alpha, epsilon, nMax, sigma, mse_prev, delta_mse_threshold=1e-6):
    # Inicialización
    uk = np.array(u0)
    mse_prev = None # Almacenar el valor de MSE de la iteración anterior

    # Crear la barra de progreso de tqdm
    pbar = tqdm(total=nMax, desc="Contrast")

    # Barra de progreso en el bucle for
    for k in range(nMax):

        # Calcular el gradiente de la energía en uk
        gk = (uk - u0) - lambdak * (uk - w(uk, sigma)) * uk

        # Actualizar uk para la siguiente iteración
        uk_next = uk - alpha * gk
        uk = uk

        # print(uk_next)
        # plt.imshow(uk_next.astype(np.uint8))
        # break

        # Calcular MSE entre iteraciones sucesivas para el criterio de parada
        mse = np.mean((uk_next - uk)**2)

        # Verificar la condición de parada basada en el cambio de MSE
        if mse_prev is not None and abs(mse - mse_prev) < delta_mse_threshold:
            print(f"\nMSE no cambio en la iteración {k+1}")
            pbar.close()
            return uk_next, k + 1, False

        mse_prev = mse # Actualizar el valor anterior de MSE para la próxima iteración

        # Actualizar la barra de progreso con el MSE actual
```

```

    pbar.set_description(f"Contrast Iter: {k+1}, MSE: {mse:.5f}")
    pbar.update(1) # Avanzar la barra de progreso en una unidad

    # Verificar la condición de parada
    if np.sqrt(mse) < epsilon or np.sqrt(mse) > 10:
        print(f"Convergencia alcanzada en la iteración {k+1}")
        pbar.close()
        return uk_next, k + 1, True

    uk = uk_next # Preparar para la siguiente iteración

    pbar.close()
    return uk, nMax, False

```

Probamos el algoritmo para el contraste local.

```
[26]: # Define los parámetros del algoritmo
epsilon = 0.001
alpha = 0.01
nMax = 3000
lambdak_values = [0.5] # Valores a probar
sigma = 5

# Imagen
u_path = glob.glob('/Users/guillermo_sego/Desktop/Segundo Semestre/CVision/
↳Tarea 5/img2.png')[0]

# Lee la imagen en color
u_bgr = cv2.imread(u_path, cv2.IMREAD_COLOR)
# Convierte la imagen a RGB
u_rgb = cv2.cvtColor(u_bgr, cv2.COLOR_BGR2RGB)

# Configurar los subplots
fig, axs = plt.subplots(1, len(lambdak_values) + 1, figsize=(15, 5))
axs[0].imshow(u_rgb)
axs[0].set_title('Imagen original')
axs[0].axis('off')

# Aplicación del algoritmo para cada valor de lambdak y cálculo de PSNR
for i, lambdak in enumerate(lambdak_values, start=1):
    restored_img, k, converged = Contrast(
        fast_gaussian_convolution, u_rgb, lambdak, alpha, epsilon, nMax, sigma
    )
    psnr = calculate_psnr(u_rgb, restored_img)
    print(f"={lambdak}, PSNR: {psnr:.2f}")

    # Mostrar la imagen restaurada y el PSNR

```

```

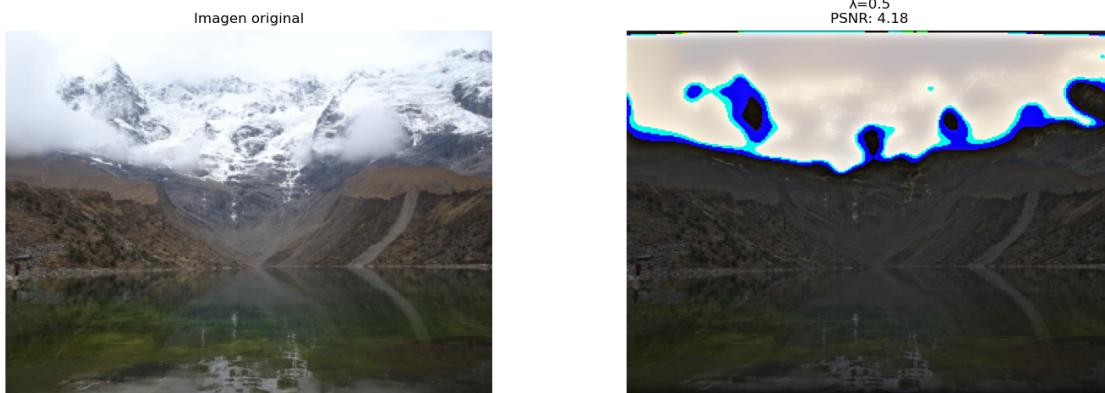
    axs[i].imshow(restored_img.astype(np.uint8))
    axs[i].set_title(f' ={lambdak}\nPSNR: {psnr:.2f}')
    axs[i].axis('off')

plt.tight_layout()
plt.show()

Contrast:  0%| 0/3000 [00:00<?, ?it/s]/var/folders/9t/s_zg1kn954l26k
1btcnp71fm0000gn/T/ipykernel_30077/2053258905.py:5: DeprecationWarning:
Importing gaussian from 'scipy.signal' is deprecated and will raise an error in
SciPy 1.13.0. Please use 'scipy.signal.windows.gaussian' or the convenience
function 'scipy.signal.get_window' instead.
    gaussian_kernel_1d_1 = np.expand_dims(signal.gaussian(u.shape[0], sigma),
axis=1)
/var/folders/9t/s_zg1kn954l26k1btcnp71fm0000gn/T/ipykernel_30077/2053258905.py:6
: DeprecationWarning: Importing gaussian from 'scipy.signal' is deprecated and
will raise an error in SciPy 1.13.0. Please use 'scipy.signal.windows.gaussian'
or the convenience function 'scipy.signal.get_window' instead.
    gaussian_kernel_1d_2 = np.expand_dims(signal.gaussian(u.shape[1], sigma),
axis=1)
Contrast Iter: 2, MSE: 25029.14677:  0%| 2/3000 [00:00<01:19,
37.65it/s]

Convergencia alcanzada en la iteración 2
=0.5, PSNR: 4.18

```



Probamos el algoritmo para el contraste global.

```
[33]: # Define los parámetros del algoritmo
epsilon = 0.001
alpha = 0.01
nMax = 3000
lambdak_values = [0.75]
sigma = 30
```

```

# Imagen
u_path = glob.glob('/Users/guillermo_sego/Desktop/Segundo Semestre/CVision/
↳Tarea 5/img2.png')[0]

# Lee la imagen en color
u_bgr = cv2.imread(u_path, cv2.IMREAD_COLOR)
# Convierte la imagen a RGB
u_rgb = cv2.cvtColor(u_bgr, cv2.COLOR_BGR2RGB)

# Configurar los subplots
fig, axs = plt.subplots(1, len(lambdak_values) + 1, figsize=(15, 5))
axs[0].imshow(u_rgb)
axs[0].set_title('Imagen original')
axs[0].axis('off')

# Aplicación del algoritmo para cada valor de lambdak y cálculo de PSNR
for i, lambdak in enumerate(lambdak_values, start=1):
    restored_img, k, converged = Contrast(
        fast_gaussian_convolution, u_rgb, lambdak, alpha, epsilon, nMax, sigma
    )
    psnr = calculate_psnr(u_rgb, restored_img)
    print(f"={lambdak}, PSNR: {psnr:.2f}")

# Mostrar la imagen restaurada y el PSNR
axs[i].imshow(restored_img.astype(np.uint8))
axs[i].set_title(f'={lambdak}\nPSNR: {psnr:.2f}')
axs[i].axis('off')

plt.tight_layout()
plt.show()

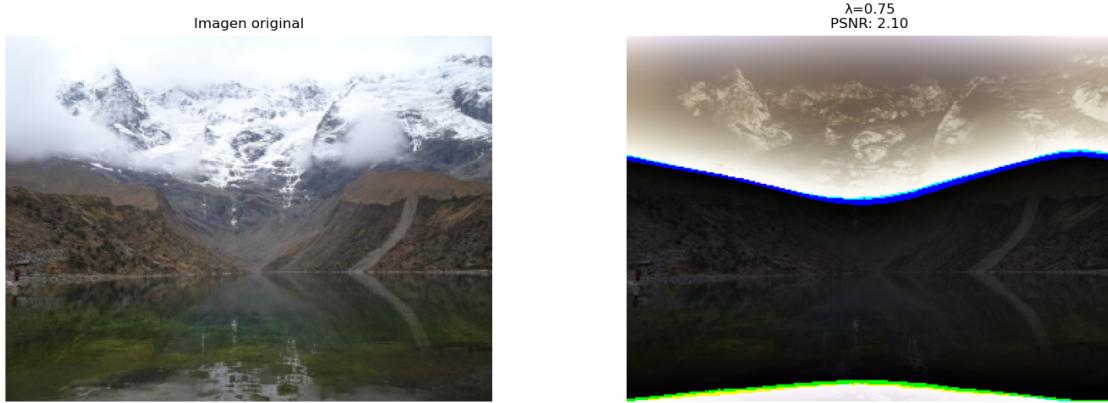
```

```

Contrast: 0% | 0/3000 [00:00<?, ?it/s]/var/folders/9t/s_zg1kn954l26k
1btcnp7lfm0000gn/T/ipykernel_30077/2053258905.py:5: DeprecationWarning:
Importing gaussian from 'scipy.signal' is deprecated and will raise an error in
SciPy 1.13.0. Please use 'scipy.signal.windows.gaussian' or the convenience
function 'scipy.signal.get_window' instead.
    gaussian_kernel_1d_1 = np.expand_dims(signal.gaussian(u.shape[0], sigma),
axis=1)
/var/folders/9t/s_zg1kn954l26k1btcnp7lfm0000gn/T/ipykernel_30077/2053258905.py:6
: DeprecationWarning: Importing gaussian from 'scipy.signal' is deprecated and
will raise an error in SciPy 1.13.0. Please use 'scipy.signal.windows.gaussian'
or the convenience function 'scipy.signal.get_window' instead.
    gaussian_kernel_1d_2 = np.expand_dims(signal.gaussian(u.shape[1], sigma),
axis=1)
Contrast Iter: 2, MSE: 40405.61762: 0% | 2/3000 [00:00<01:15,
39.84it/s]

```

Convergencia alcanzada en la iteración 2
 $\lambda = 0.75$, PSNR: 2.10



Utilizamos un valor de sigma menor para el global debido a que la imagen se saturaba demasiado. El algoritmo para procesamiento de contraste no funcionó de la manera óptima para los casos de la imagen, parece resaltar las sombras en la parte cálida de la imagen, en lugar del total.

1.5 Bibliografía

- [1] Paragios, Nikos, Yunmei Chen, and Olivier D. Faugeras, eds. Handbook of mathematical models in computer vision. Springer Science & Business Media, 2006.
- [2] Castleman, Kenneth R. Digital image processing. Prentice Hall Press, 1996.