

Pirámides de Gauss y Laplace

Tarea 1, Procesamiento Avanzado de Imágenes

Integrantes: Benjamín A. Irarrázabal T.

Profesor: Javier Ruiz del Solar

Auxiliar: Patricio Loncomilla

Ayudantes: José Díaz V.

Danilo Moreira

Javier Mosnaim Z.

Jhon Pilataxi

Fecha de realización: 04 de septiembre de 2022

Fecha de entrega: 04 de septiembre de 2022

Santiago de Chile

Índice de Contenidos

1. Introducción	1
2. Marco Teórico	2
2.1. Pirámide de Gauss	2
2.2. Pirámide de Laplace	3
2.3. Filtrado Pasa-Altos	4
3. Desarrollo	4
3.1. Resultados	5
3.1.1. Pirámide de Gauss	5
3.1.2. Pirámide de Laplace	7
3.1.2.1 Reconstrucción	9
3.1.3. Filtrado Pasa-Altos	11
3.1.3.1 Laplacian of Gaussian	11
3.1.3.2 Derivative of Gaussian	13
3.1.3.3 Comparación con pirámide de Laplace	15
3.2. Análisis de Resultados	19
3.2.1. Pirámide de Gauss	20
3.2.2. Pirámide de Laplace	20
3.2.2.1 Reconstrucción	20
3.2.3. Filtrado Pasa-Altos	20
3.2.3.1 Filtros	20
3.2.3.2 Filtros v/s Laplace	20
4. Conclusiones	21
5. Bibliografía	22
Referencias	22
6. Anexos	23
6.1. Imágenes de prueba	23
6.2. Código implementado	23

Índice de Figuras

1. Máscara gaussiana computada	2
2. Pirámide de Gauss para corteza_2022.jpg	5
3. Pirámide de Gauss para origami_2022.jpg	6
4. Pirámide de Gauss para techo_falso_2022.jpg	6
5. Pirámide de Gauss para dali_2022.jpg	7
6. Pirámide de Laplace para corteza_2022.jpg	8
7. Pirámide de Laplace para origami_2022.jpg	8
8. Pirámide de Laplace para techo_falso_2022.jpg	9

9.	Pirámide de Laplace para dali_2022.jpg	9
10.	corteza_2022.jpg luego de aplicar reconstrucción	10
11.	origami_2022.jpg luego de aplicar reconstrucción	10
12.	techo_falso_2022.jpg luego de aplicar reconstrucción	11
13.	dali_2022.jpg luego de aplicar reconstrucción	11
14.	Filtro pasa-altos LoG para corteza_2022.jpg	12
15.	Filtro pasa-altos LoG para origami_2022.jpg	12
16.	Filtro pasa-altos LoG para techo_falso_2022.jpg	13
17.	Filtro pasa-altos LoG para dali_2022.jpg	13
18.	Filtro pasa-altos DoG para corteza_2022.jpg	14
19.	Filtro pasa-altos DoG para origami_2022.jpg	14
20.	Filtro pasa-altos DoG para techo_falso_2022.jpg	15
21.	Filtro pasa-altos DoG para dali_2022.jpg	15
22.	Comparación entre pirámide de Laplace y filtro LoG	16
23.	Comparación entre pirámide de Laplace y filtro DoG	16
24.	Comparación entre pirámide de Laplace y filtro LoG	17
25.	Comparación entre pirámide de Laplace y filtro DoG	17
26.	Comparación entre pirámide de Laplace y filtro LoG	18
27.	Comparación entre pirámide de Laplace y filtro DoG	18
28.	Comparación entre pirámide de Laplace y filtro LoG	19
29.	Comparación entre pirámide de Laplace y filtro DoG	19
30.	Imágenes de prueba	23

1. Introducción

El procesamiento de imágenes se compone por un conjunto de técnicas aplicadas a imágenes digitales para poder modificar su calidad o buscar información dentro de esta, por ejemplo, colores, bordes, entre otros. Estos métodos han tenido un gran impacto en variadas áreas tales como medicina, telecomunicaciones, industria e incluso entretenimiento. Dentro de estos procedimientos, se encuentran las **Pirámides de Gauss y de Laplace**, las cuales son representaciones multi-resolución calculadas a partir de una imagen de entrada (Input). La primera, contiene distintas resoluciones de la misma imagen, por otro lado, la pirámide de Laplace guarda la información que se va perdiendo al ir generando la pirámide de Gauss a causa de la convolución de matrices.

El presente informe, tiene como objetivo el desarrollo de la primera tarea del curso Procesamiento Avanzado de Imágenes y busca, usando herramientas computacionales, el cálculo de pirámides de Gauss y Laplace de imágenes de prueba, aplicando además, un filtrado pasa-alto para detectar bordes dentro de estas. Para esto, se comenzará con un Marco Teórico comentando sobre los principales algoritmos usados, tales como convolución matricial, y los asociados a las pirámides. Posteriormente, se detallará el desarrollo de la experiencia finalizando con las conclusiones más importantes de esta. Cabe destacar, que al final de este, se adjuntan los respectivos códigos en lenguaje Python, que fueron utilizados para resolver los problemas asociados.

2. Marco Teórico

La siguiente sección, tiene como objetivo describir los algoritmos utilizados para resolver los problemas asociados al desarrollo de la experiencia, cabe destacar, que existen funciones que se utilizan en ambos casos, como por ejemplo convolution_cython(), entre otras. A continuación se detalla lo solicitado.

2.1. Pirámide de Gauss

- **convolution_cython:**

Esta función busca implementar la convolución entre dos matrices “a mano”, es decir, sin utilizar las librerías provistas por el lenguaje utilizado. Esta recibe dos parámetros, una imagen de entrada y una máscara, para posteriormente realizar la convolución. Para esta, primero se ubica el centro del kernel en la posición adecuada evitando sobrepasar los límites. Finalmente se multiplica posición por posición, guardando en una variable la suma de estas operaciones. En otras palabras, la matriz (imagen) resultante, se construye a partir de combinaciones lineales entre el input y la máscara [1].

- **compute_gauss_mask_2d:**

En simples palabras, esta función se encarga de generar una máscara gaussiana cuadrada en dos dimensiones. Para esto, recibe un largo y desviación deseada. Con esto, toma el valor del largo y calcula un punto medio, para posteriormente crear un arreglo con **numpy**. Luego de esto, se aplica la ecuación 1 y después el producto externo entre este arreglo y sí mismo. Finalmente, se normaliza la máscara, dividiendo por la suma total de valores en su interior [2].

$$\text{Gauss}(x, y) = \frac{1}{2\pi\sigma^2} \cdot e^{-\frac{(x^2+y^2)}{2\sigma^2}} \quad (1)$$

A continuación, se presenta un ejemplo de máscara Gaussiana computada con la función anterior, esta tiene una desviación $\sigma = 2.0$ y un largo $width = 15$.

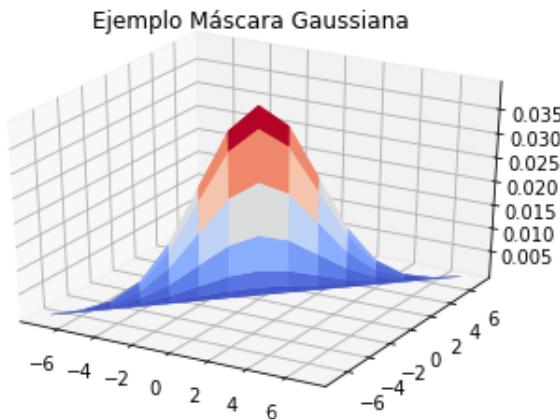


Figura 1: Máscara gaussiana computada

- **apply_blur:**

apply_blur recibe una imagen de entrada junto a un largo y una desviación. En primer lugar

crea una máscara utilizando la función anterior, para posteriormente aplicar la convolución entre la entrada y el kernel gaussiano. Finalmente, entrega la imagen resultante. Cabe destacar, que al aplicar la función de convolución, se generan pérdidas en la imagen en los extremos a causa de los límites con el kernel gaussiano, por lo cual, en los resultados que se verán más adelante, se observan bordes en los extremos de las imágenes (marco).

- **do_subsample:**

Esta función realiza un submuestreo de la imagen, con la finalidad de reducir su tamaño a la mitad. Para esto, toma una imagen de entrada y copia los valores de sus **posiciones pares** (es decir, (0,0), (0,2)...) en un nuevo arreglo. Esto se realiza ya que, los píxeles más cercanos entre sí contienen información relativamente similar, por lo que se asume que tomar una muestra de estos, no aumentará drásticamente las pérdidas.

- **calc_gauss_pyramid:**

calc_gauss_pyramid une todas las funciones descritas anteriormente para entregar un arreglo con las imágenes correspondientes para los niveles definidos por el usuario. En primer lugar, toma la imagen de entrada (y la guarda en el arreglo), luego, aplica iterativamente la función apply_blur y do_subsample, guardando los resultados de la misma manera mencionada.

- **show_gauss_pyramid:**

Esta función únicamente toma el arreglo de entrada e iterativamente imprime en pantalla las imágenes correspondientes a la pirámide de Gauss.

2.2. Pirámide de Laplace

- **subtract:**

La función subtract recibe dos imágenes de entrada (asumidas del mismo tamaño), para aplicar la resta **píxel a píxel**, es decir, se recorren ambas imágenes al mismo tiempo y se restan los valores en sus píxeles con igual posición, guardando el resultado en una imagen de salida.

- **add:**

Análoga a la función anterior, add recorre ambas imágenes y aplica la suma píxel a píxel, generando una salida compuesta por la adición de valores en igual posición de las entradas.

- **calc_laplace_pyramid:**

Como se mencionó en la sección 1, la pirámide de Laplace guarda la información que se pierde a causa de la manipulación de las imágenes en la pirámide de Gauss. Para esto, luego de usar apply_blur, se utiliza la función subtract para poder almacenar los cambios en la imagen.

- **show_laplace_pyramid:**

En primer lugar, esta función recorre los píxeles de todas las imágenes (excepto de la del último nivel) y aplica la función valor absoluto (evitando así, valores negativos en la matriz), Finalmente y análogo al caso anterior, se recorre el arreglo recibido imprimiendo las imágenes en pantalla.

- **do_upsample:**

La función upsample crea una imagen del doble del tamaño con respecto a la de entrada. Luego, analiza según posiciones (par o impar) y calcula el promedio de los 4 valores correspondientes en el Input.

- **do_reconstruct**

Reconstruct, toma la última imagen de la pirámide de Laplace y comienza a aplicar iterativamente un upsample y una suma, para tratar de reconstruir la imagen a partir de una resolución menor.

2.3. Filtrado Pasa-Altos

- **LoG:**

Análoga a la función para computar una máscara gaussiana, esta genera un kernel basado en *Laplacian of Gaussian* (LoG). Ingresando una desviación y un largo deseado se entregan dos máscaras, una con respecto a “X” y otra con respecto a “Y”. A continuación, se presenta la ecuación de una máscara LoG en 2D [3].

$$\text{LoG}(x, y) = \frac{\partial^2 G}{\partial x^2} + \frac{\partial^2 G}{\partial y^2} \quad (2)$$

$$\text{LoG}(x, y) = \frac{1}{2\pi\sigma^4} \left(-2 + \frac{x^2 + y^2}{\sigma^2} \right) e^{-\frac{(x^2+y^2)}{2\sigma^2}} \quad (3)$$

- **apply_blur_Log:**

Idéntica a apply.blur, se diferencia en que en esta ocasión se crea una máscara LoG en vez de Gaussiana. Entrega una imagen filtrada pasa-altos.

- **DoG:**

Análogo a lo anterior, esta función computa una máscara basada en *Derivative of Gaussian* (DoG). Ingresando una desviación y un largo deseado se entrega un kernel normalizado (para eje X e Y). A continuación, se presentan las ecuaciones en 2D para este filtro [4].

$$\text{DoG}(x, y) = \frac{\partial G}{\partial x} + \frac{\partial G}{\partial y} \quad (4)$$

$$\text{DoG}(x, y) = - \left(\frac{x + y}{2\pi\sigma^4} \right) e^{-\frac{(x^2+y^2)}{2\sigma^2}} \quad (5)$$

- **apply_blur_DoG:**

Aplica la convolución utilizando una máscara DoG. Entrega una imagen filtrada pasa-altos.

No obstante, para el filtrado pasa-altos, se realizan aproximaciones siguiendo lo estudiado en la referencia [5], esto debido a problemas al aplicar la máscara LoG y DoG 2D directamente. Sin embargo, esta aproximación permite separar eficientemente la derivada en X de Y, luego aplicar cada máscara y combinar ambos resultados en un solo *output*.

3. Desarrollo

A continuación, se presentan los resultados post-aplicación de las funciones descritas anteriormente a las cuatro imágenes de prueba. Posteriormente, se comentará sobre estos, analizando la efectividad del algoritmo. Cabe destacar, que estas imágenes también se pueden visualizar (para observar de mejor forma los detalles) usando el Notebook adjunto a este informe.

3.1. Resultados

En la siguiente sección se presentan los resultados obtenidos para las pirámides de Gauss, Laplace y el Filtrado pasa-altos en base a LoG y DoG (ver sección 2).

3.1.1. Pirámide de Gauss

Para los siguientes resultados se utilizó una máscara Gaussiana computada usando la función explicitada en la sección 2, la cual se encuentra explícita en la sección 6. Esta se construyó usando una desviación $\sigma = 2.0$ y un largo $width = 7$.

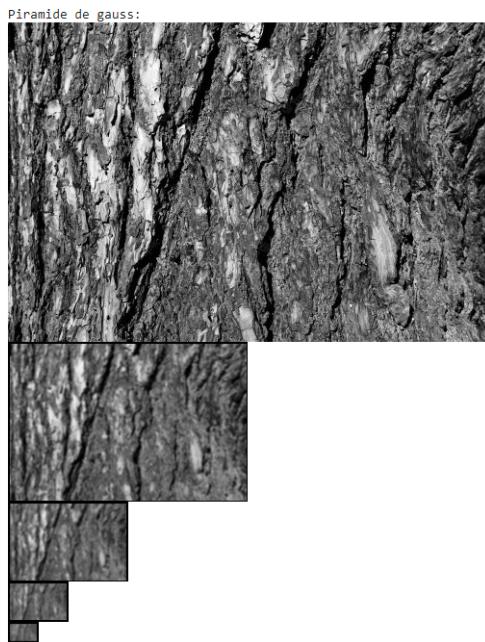


Figura 2: Pirámide de Gauss para corteza_2022.jpg

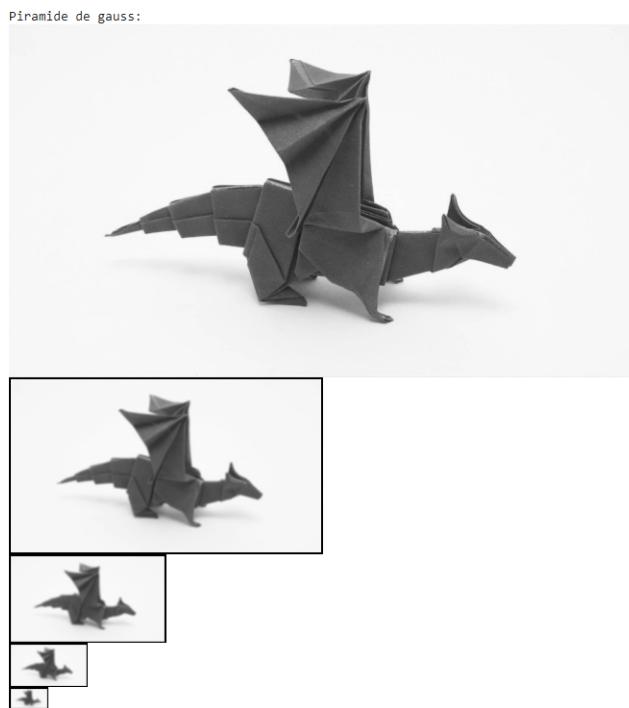


Figura 3: Pirámide de Gauss para origami_2022.jpg



Figura 4: Pirámide de Gauss para techo_falso_2022.jpg

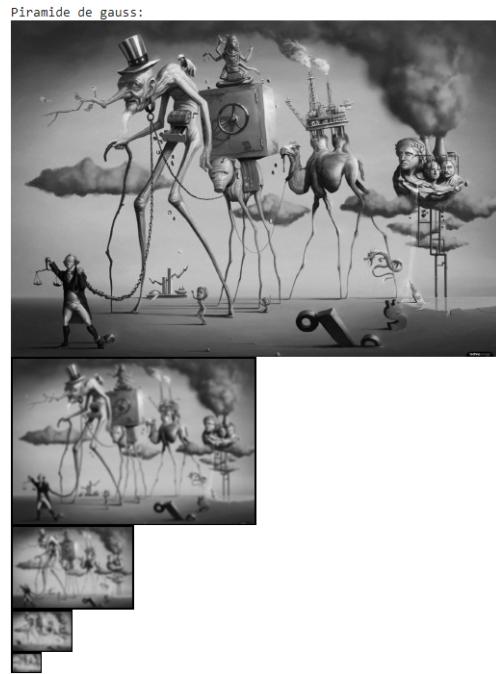


Figura 5: Pirámide de Gauss para dali_2022.jpg

3.1.2. Pirámide de Laplace

Al igual que en la sección anterior, se utilizó una máscara Gaussiana con desviación $\sigma = 2.0$ y un largo $width = 7$. Además, para mostrar las imágenes de la pirámide, se utilizó un factor constante de valor 6. Este se escogió realizando diferentes pruebas y seleccionando el que permitía mejorar la calidad visible de la imagen.

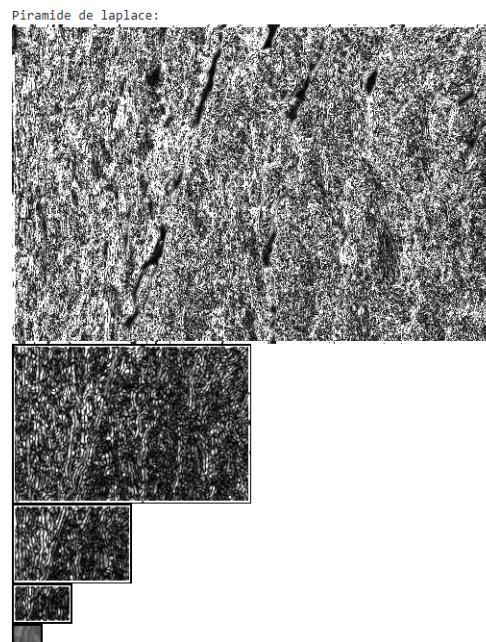


Figura 6: Pirámide de Laplace para corteza_2022.jpg

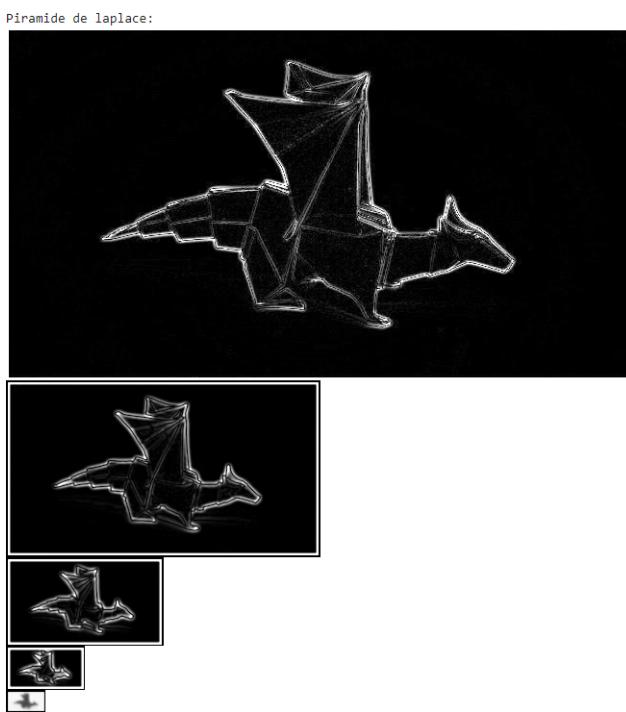


Figura 7: Pirámide de Laplace para origami_2022.jpg

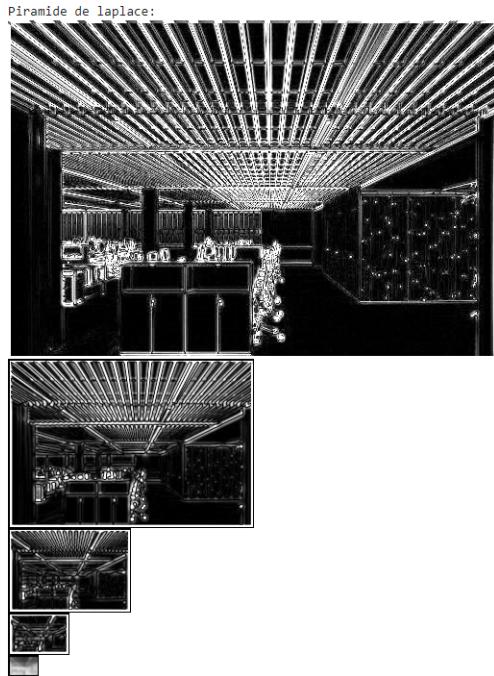


Figura 8: Pirámide de Laplace para techo_falso_2022.jpg

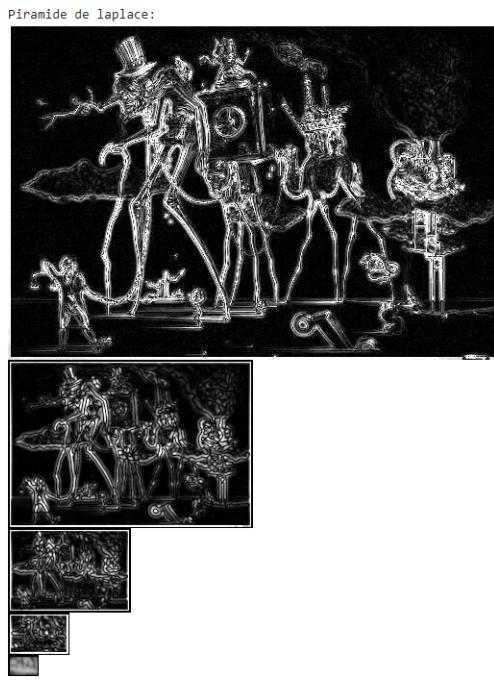


Figura 9: Pirámide de Laplace para dali_2022.jpg

3.1.2.1. Reconstrucción

En esta sección, se utilizan las funciones explicadas en la sección 2, las cuales corresponden a un *upsample* de la imagen de entrada y una reconstrucción usando la pirámide de Laplace. Los resultados

se presentan a continuación

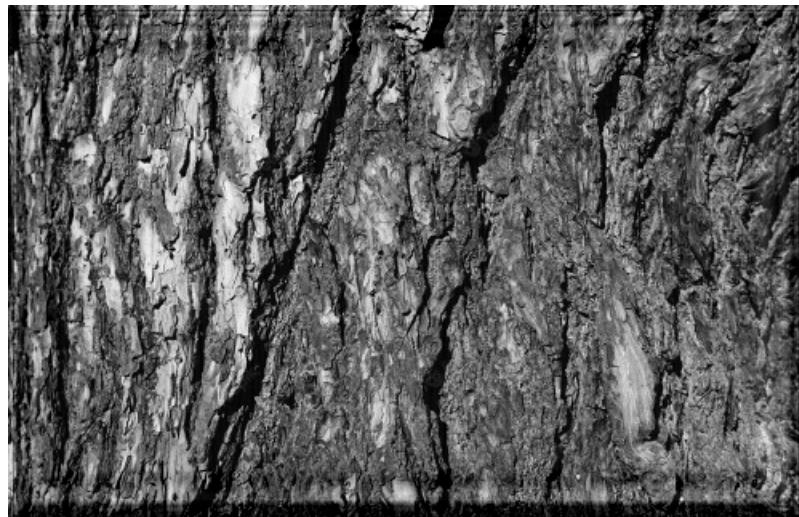


Figura 10: corteza_2022.jpg luego de aplicar reconstrucción

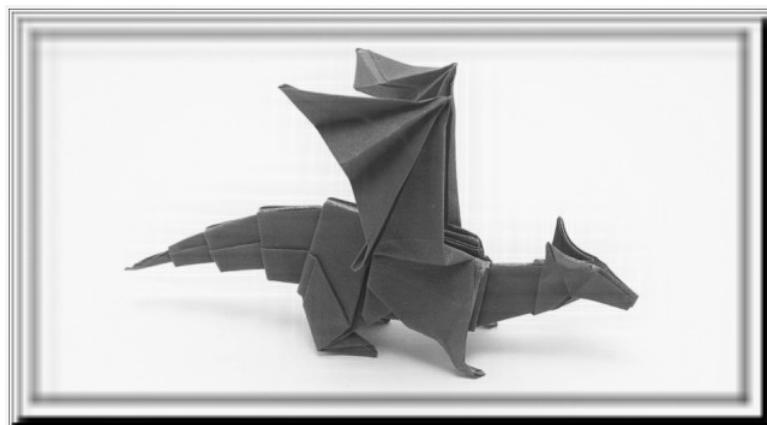


Figura 11: origami_2022.jpg luego de aplicar reconstrucción



Figura 12: techo_falso_2022.jpg luego de aplicar reconstrucción

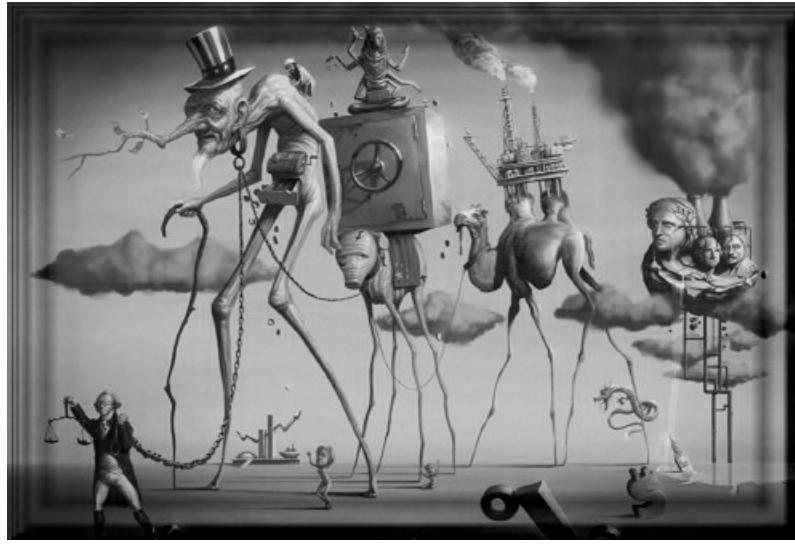


Figura 13: dali_2022.jpg luego de aplicar reconstrucción

3.1.3. Filtrado Pasa-Altos

El filtrado pasa-altos se realiza utilizando dos métodos distintos (LoG y DoG). Cabe destacar, que los parámetros utilizados (desviación y largo) fueron los mejores encontrados en base a la experimentación, ya que con otros valores, la imagen perdía visibilidad o simplemente no filtraba.

3.1.3.1. Laplacian of Gaussian

Los siguientes resultados, muestran tres imágenes correspondientes a la entrada, la salida (post convolución con la máscara) y una resta entre estas para ver los cambios. Para estas imágenes, se usó una máscara LoG con una desviación $\sigma = 1.4$ y un largo $width = 15$.

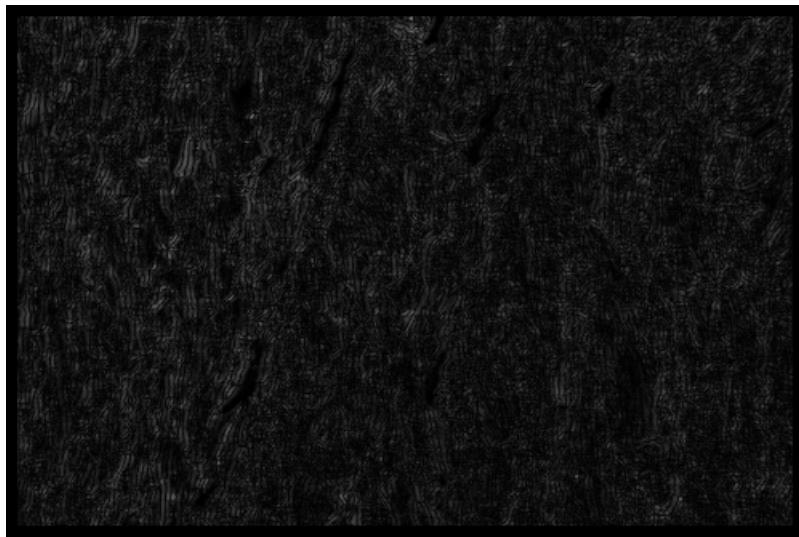


Figura 14: Filtro pasa-altos LoG para corteza_2022.jpg

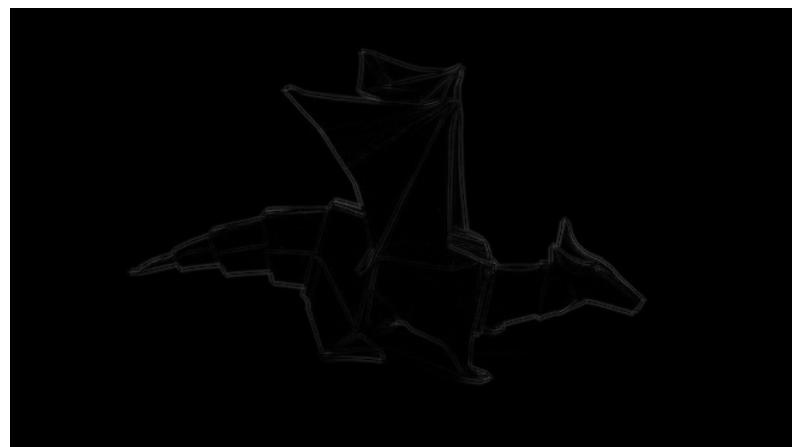


Figura 15: Filtro pasa-altos LoG para origami_2022.jpg

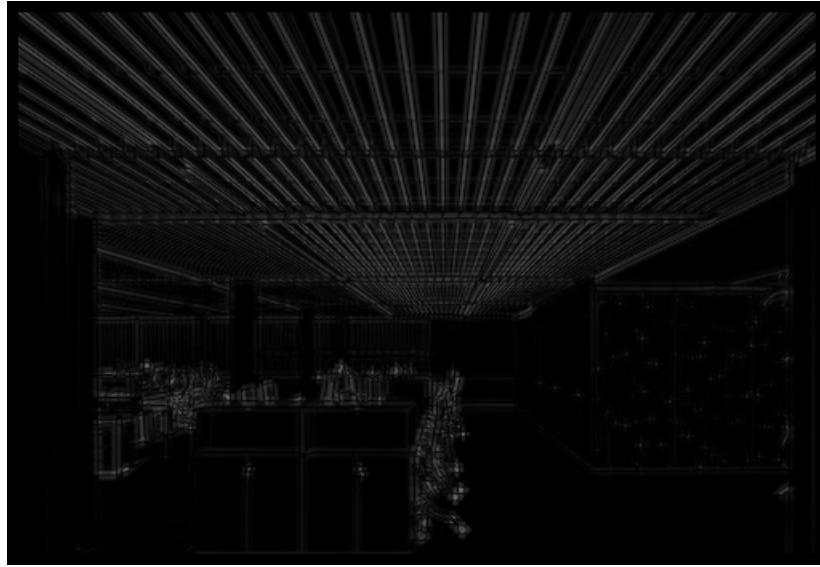


Figura 16: Filtro pasa-altos LoG para techo_falso_2022.jpg



Figura 17: Filtro pasa-altos LoG para dali_2022.jpg

3.1.3.2. Derivative of Gaussian

Análogo a la sección anterior, los resultados que se presentan a continuación, muestran tres imágenes. Para estas se utilizó una máscara DoG con una desviación $\sigma = 1.4$ y un largo $width = 9$

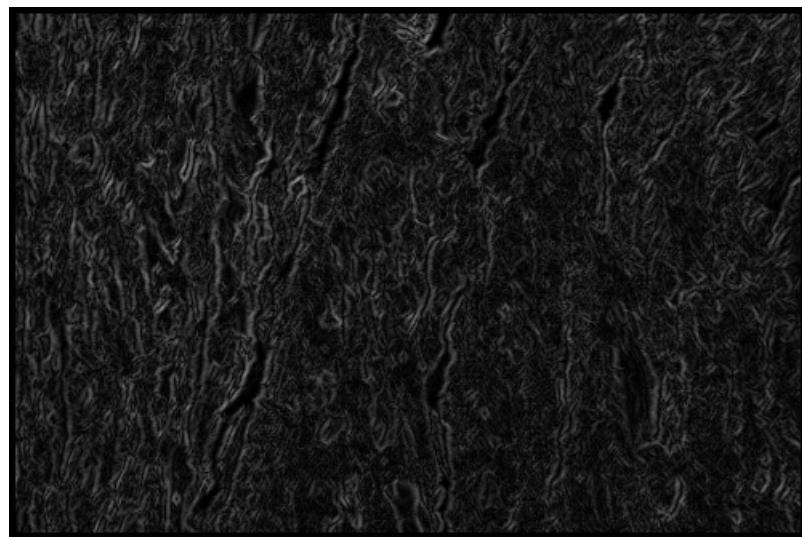


Figura 18: Filtro pasa-altos DoG para corteza_2022.jpg

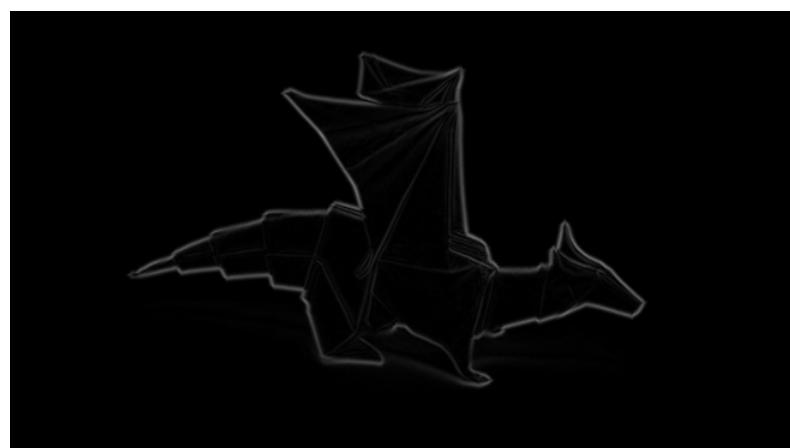


Figura 19: Filtro pasa-altos DoG para origami_2022.jpg



Figura 20: Filtro pasa-altos DoG para techo_falso_2022.jpg



Figura 21: Filtro pasa-altos DoG para dali_2022.jpg

3.1.3.3. Comparación con pirámide de Laplace

Para realizar la debida comparación, se aplican ambos filtros a cada piso de la pirámide de Gauss. Para esto, se utilizan los valores especificados anteriormente para el tamaño del filtro y la desviación, es decir:

- Parámetros LoG:

$$\sigma = 1.4, \text{ width} = 15 \quad (6)$$

- Parámetros DoG:

$$\sigma = 1.4, \text{ width} = 9 \quad (7)$$

Estos parámetros fueron escogidos en base a la experimentación, analizando numéricamente las más caras generadas en conjunto a los resultados obtenidos. Estos últimos se presentan a continuación.

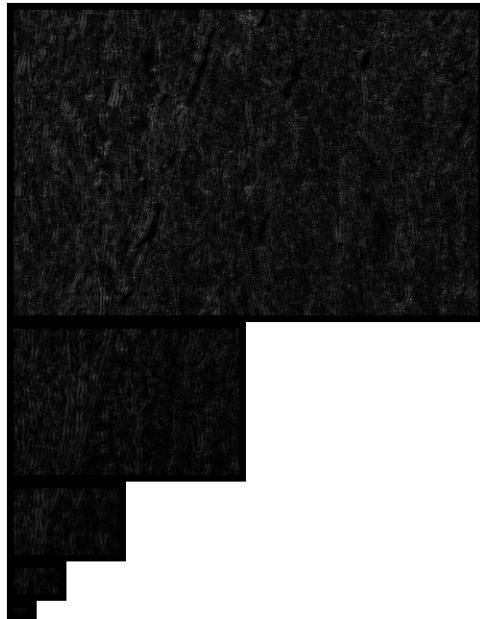


Figura 22: Comparación entre pirámide de Laplace y filtro LoG



Figura 23: Comparación entre pirámide de Laplace y filtro DoG



Figura 24: Comparación entre pirámide de Laplace y filtro LoG

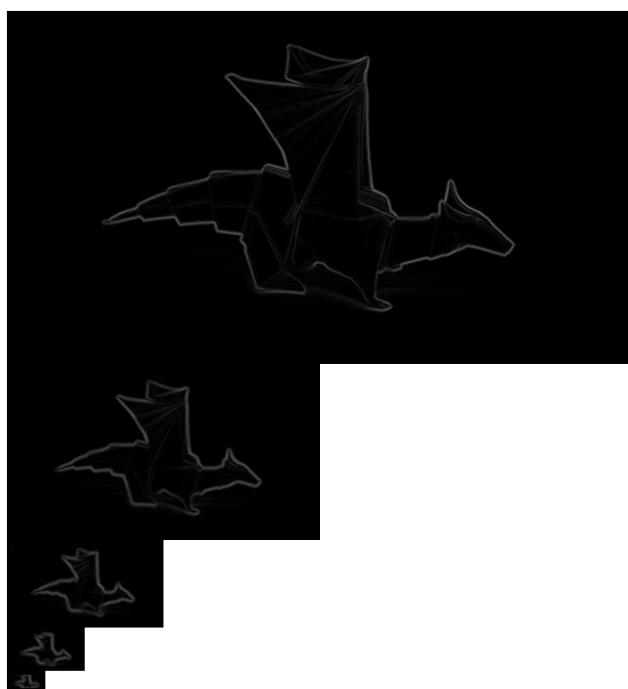


Figura 25: Comparación entre pirámide de Laplace y filtro DoG

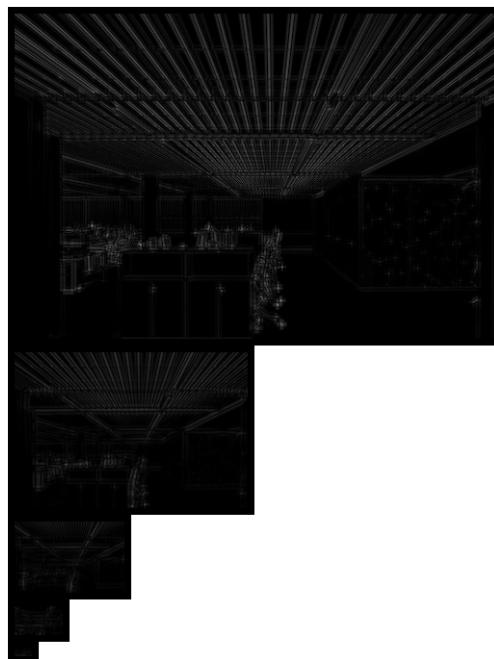


Figura 26: Comparación entre pirámide de Laplace y filtro LoG

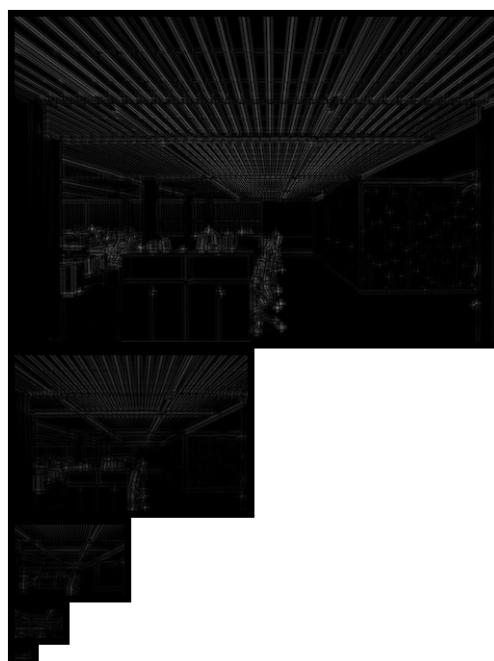


Figura 27: Comparación entre pirámide de Laplace y filtro DoG

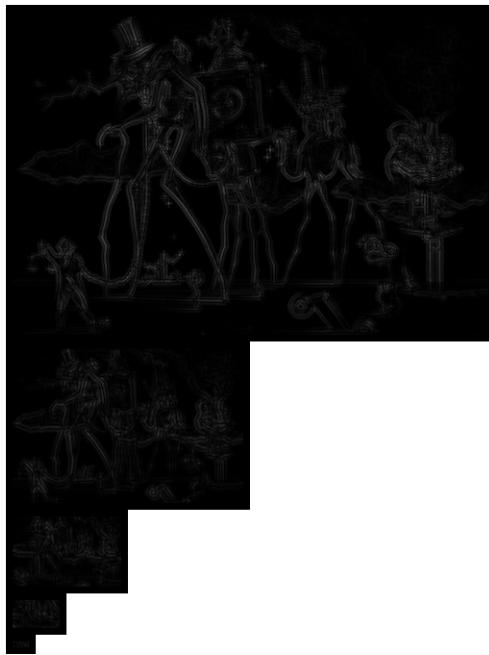


Figura 28: Comparación entre pirámide de Laplace y filtro LoG



Figura 29: Comparación entre pirámide de Laplace y filtro DoG

3.2. Análisis de Resultados

En la siguiente sección se realizarán los comentarios principales sobre los resultados de la sección 3.1. Estos se realizarán en base a lo obtenido y su contraste con la teoría.

3.2.1. Pirámide de Gauss

A partir de las figuras 2, 3, 4 y 5, se puede notar que se obtiene una vista multiresolución de las imágenes de prueba, con un filtro gaussiano aplicado a los niveles 2, ..., 5. Este último, produce una imagen borrosa, en la cual, los bordes de los objetos dentro de esta se pierden ligeramente. Esto, permite que en la sección siguiente se pueda encontrar la pirámide de Laplace.

3.2.2. Pirámide de Laplace

Posteriormente, en las figuras 6, 7, 8 y 9, se observa que se pueden rescatar eficientemente los bordes de los objetos dentro de las imágenes. No obstante, se observa que esto, es más efectivo para imágenes donde el objeto contrasta con el fondo. Por ejemplo, para la figura dali_2022, los bordes en algunos casos se pueden ver menos destacados que en la figura origami_2022.

3.2.2.1. Reconstrucción

A partir de las imágenes 10, 11, 12 y 13, se puede observar que la imagen se logra reconstruir con ciertos errores en algunos pixeles cercanos a los extremos de la imagen. Lo anterior, posiblemente causado por las posiciones con valor en 0 en los extremos de la pirámide, debido a la convolución realizada.

3.2.3. Filtrado Pasa-Altos

3.2.3.1. Filtros

En las figuras 14,..., 21, se presentan los resultados luego de aplicado los filtros tanto LoG como DoG. Para ambos casos, visualmente se obtienen imágenes similares con respecto a las pirámides de Laplace. No obstante, como se verá en la siguiente sección, difieren en algunos pixeles.

Cabe destacar, que se probó con distintos valores de sigma y width para analizar los cambios en la salida de los filtros, no obstante, con algunos valores se obtuvieron imágenes sin filtrar o con un filtrado no eficiente. En otros casos, estos cambios no eran notorios, siendo muy semejantes entre sí. Además, para escoger los valores óptimos, se visualizaron las máscaras obtenidas, analizando su gráfico asociado.

3.2.3.2. Filtros v/s Laplace

Como fue mencionado en la sección anterior, al aplicar los filtros se obtienen imágenes muy similares en términos visuales a las de la pirámide de Laplace, esto fue comprobado realizando el procedimiento de la sección 3.1.3.3, donde se obtuvieron las figuras 22, ..., 29.

Para las imágenes con bordes bien definidos, como origami y techo falso, se obtiene un resultado claro, donde se observan de buena manera los contornos en esta. Por otro lado, para las imágenes corteza y dali, que presentan bordes más “confusos”, la diferencia entre ambos métodos no es tan clara y las imágenes resultantes muestran pixeles de valor cercano a 0.

Finalmente, se observa que en los últimos niveles de las pirámides, la mayoría de los pixeles son negros, perdiendo la capacidad de filtrado a medida que se disminuye la resolución de la imagen.

4. Conclusiones

En base a los resultados obtenidos, se puede afirmar que se cumple con el objetivo principal de la experiencia, el cual se enfocaba en implementar un algoritmo capaz de computar las pirámides de Gauss y Laplace junto a un filtro pasa altos de cuatro imágenes de prueba. Dentro de esto, fue complejo crear las funciones que realizan la reconstrucción de la imagen a partir del último piso de la pirámide de Laplace, esto debido a que existen complicaciones con los tamaños de las imágenes. Además, se presentaron dificultades al momento de aplicar los filtros LoG y DoG, debido a la elección de un sigma y un width no óptimo.

Si bien, en general los resultados fueron bastante positivos, en la sección de reconstrucción se obtuvieron imágenes con ciertos errores causados por el “marco” generado por la convolución. Esto se pudo corregir realizando una función que analizara estos casos de borde y los trabajara correctamente, sumando los pixeles correspondientes y evitando algunos errores de arrastre en cada piso.

Con lo anterior, se pudo comprender la importancia de estos filtros y sus aplicaciones dentro del procesamiento de imágenes. Pudiendo notar, cómo estas herramientas sirven para encontrar bordes y realizar análisis posteriores.

Finalmente, se debe destacar que el uso de pirámides de Gauss o Laplace, permite tener visualizaciones de menor resolución y por ende, menor espacio ocupado en memoria, lo cual puede ser de gran utilidad al momento de trabajar por ejemplo, con una gran base de datos de imágenes.

5. Bibliografía

Referencias

- [1] Soto P. Convolución Matricial Aplicado al Procesamiento de Imágenes. Instituto Tecnológico de Costa Rica. Disponible en: https://www.tec.ac.cr/sites/default/files/media/uploads/presentacion_pablosoto.pdf
- [2] Stackoverflow. How to calculate a Gaussian kernel matrix efficiently in numpy?. Septiembre 2021. Disponible en: <https://stackoverflow.com/questions/29731726/how-to-calculate-a-gaussian-kernel-matrix-efficiently-in-numpy>
- [3] Hypermedia Image Processing Reference. Laplacian/Laplacian of Gaussian. Disponible en: <https://homepages.inf.ed.ac.uk/rbf/HIPR2/log.htm>
- [4] Technische Universität München. 1D and 2D Gaussian Derivatives. Mayo 2010. Disponible en: <https://campar.in.tum.de/Chair/HaukeHeibelGaussianDerivatives>
- [5] Collins R. LoG and DoG Filters. Disponible en: <https://www.cse.psu.edu/~rtc12/CSE486/lecture11.pdf>

6. Anexos

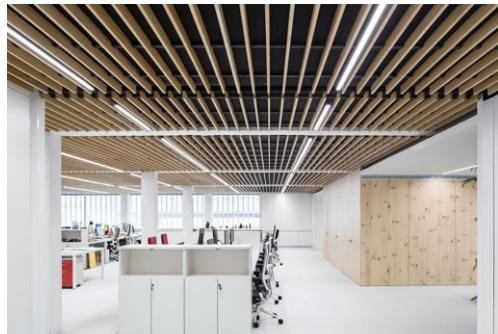
6.1. Imágenes de prueba



(a) corteza



(b) origami



(c) techo falso



(d) dali

Figura 30: Imágenes de prueba

6.2. Código implementado

A continuación, se presenta el código implementado en el desarrollo de la tarea, cabe destacar, que al momento de imprimir (mostrar) los resultados, sólo se copia el fragmento utilizado para una imagen, ya que, para las demás simplemente se cambia el nombre.

```

1 from google.colab import files
2 uploaded = files.upload()
3
4 # Mostrar archivos en la carpeta del notebook
5 !ls
6
7 # Para medir tiempo de ejecucion
8 !pip install ipython-autotime
9
10 # Commented out IPython magic to ensure Python compatibility.
11 # Extensiones
12 # \%load_ext Cython
13 # \%load_ext autotime
14

```

```
15 # Paquetes a ser usados
16 import numpy as np
17 import cv2
18 import cython
19 import numpy as np
20 import math
21 import matplotlib.pyplot as plt
22 from matplotlib import cm
23 # Este paquete solo se debe usar si se usa colaboratory
24 from google.colab.patches import cv2_imshow
25
26 """# Pirámide de Gauss"""
27
28 # Commented out IPython magic to ensure Python compatibility.
29 # %cython
30 # import cython
31 # import numpy as np
32 # cimport numpy as np
33 #
34 # La convolucion debe ser implementada usando cython (solo esta funcion en cython)
35 # #@cython.boundscheck(False)
36 # cpdef np.ndarray[np.float32_t, ndim=2] convolution_cython(np.ndarray[np.float32_t, ndim=2]
37 #     ↪ input, np.ndarray[np.float32_t, ndim=2] mask):
38 #     cdef int row, col, rows, cols, mask_centerR, mask_centerC, i, j # se agregan 4 variables nuevas
39 #     ↪ para manejar las posiciones de la iteración.
40 #     cdef float sum
41 #     cdef np.ndarray[np.float32_t, ndim=2] output=np.zeros([input.shape[0], input.shape[1]], dtype =
42 #     ↪ np.float32)
43 #
44 #     # tamano de la imagen
45 #     rows = input.shape[0]
46 #     cols = input.shape[1]
47 #
48 #     # Buscamos el centro del kernel (R=row, C=column), para poder iterar la convolucion.
49 #     mask_centerR = mask.shape[0] // 2
50 #     mask_centerC = mask.shape[1] // 2
51 #
52 #
53 #     # A continuación se cambiaron los atributos de la función range para poder ubicar
54 #     # correctamente el centro del kernel y evitar errores.
55 #     for row in range(mask_centerR, rows - mask_centerR):
56 #         for col in range(mask_centerC, cols - mask_centerC):
57 #             sum = 0 # se define sum dentro de la iteración para luego ser modificado únicamente con las
58 #             ↪ multiplicaciones entre input y kernel.
59 #             for i in range(mask.shape[0]):
60 #                 for j in range(mask.shape[1]):
```

```
61 # return output
62 #
63 """
64 Test de la función de convolución, se crea un arreglo matricial de input (simulando una imagen de
65     ↪ entrada) y una máscara sencilla para comprobar el funcionamiento.
66 Se realiza un cambio de tipo a float32 de los arreglos para corregir errores y se comprueba el
67     ↪ funcionamiento.
68 Finalmente, se observa que se realiza correctamente la convolución, obteniendo una salida del mismo
69     ↪ tamaño que la entrada con ceros en los extremos.
70 """
71
72 input = np.arange(1,82)
73 input = input.reshape(9,9)
74 kernel = np.array([[0,0,0],[0,1,0],[0,0,0]])
75 input = np.float32(input)
76 kernel = np.float32(kernel)
77 C = convolution_cython(input, kernel)
78 print(C)
79
80 def compute_gauss_mask_2d(sigma, width):
81     gmask = np.zeros((width, width), np.float32)
82
83     center = width // 2 # centro de la máscara
84     ax = np.linspace(-center, center, width) # se define un arreglo
85     for i in range(len(ax)): # Filas
86         for j in range(len(ax)): # Columnas
87             # se definen variables auxiliares
88             x = ax[i]
89             y = ax[j]
90             gmask[i][j] = (1/(2*np.pi*np.square(sigma))) * (np.exp(-0.5 * (np.square(x)+np.square(y)) / np.
91                 ↪ square(sigma))) # se computan los píxeles de la máscara
92
93     return gmask / np.sum(gmask) # Se normaliza dividiendo por la suma total dentro de la matriz
94
95 """
96 Test de la función generadora de una máscara gaussiana a partir del largo (width) y el sigma (
97     ↪ desviación).
98 """
99
100 width = 15
101 sigma = 2.0
102 gmask = compute_gauss_mask_2d(sigma, width)
103 #print(gmask, np.sum(gmask))
104 width = gmask.shape[0]
105 center = width // 2
106
107 fig = plt.figure()
108 ax = fig.add_subplot(111, projection='3d')
109
110 x = np.linspace(-center, center, width)
111 y = np.linspace(-center, center, width)
```

```
107 ax.plot_surface(x, y, gmask, cmap = cm.coolwarm)
108
109 plt.title('Ejemplo Máscara Gaussiana')
110
111 plt.show()
112
113 def apply_blur(input, sigma, width):
114     # Se computa una máscara gaussiana tomando los atributos de entrada
115     gmask = compute_gauss_mask_2d(sigma, width)
116     # Se aplica una convolución matricial usando la máscara generada en la línea anterior
117     return convolution_cython(input, gmask)
118
119 def do_subsample(img):
120     r,c = img.shape
121     output = np.zeros((r//2, c//2), np.float32)
122     for i in range(output.shape[0]):
123         for j in range(output.shape[1]):
124             output[i][j] = img[2*i][2*j]
125     return output
126
127 def calc_gauss_pyramid(input, levels):
128     gausspyr = [] # Lista que guarda las imágenes de la pirámide
129
130     current = np.copy(input) # Primera imagen (sin modificaciones)
131     gausspyr.append(current)
132
133     for i in range(1,levels):
134         current = apply_blur(gausspyr[i-1], 2.0, 7) # Se aplica una convolución a la imagen en i-1
135         current = do_subsample(current) # Se realiza submuestreo del resultado anterior
136         gausspyr.append(current) # Se agrega a la lista
137     return gausspyr # Se retorna la lista
138
139 def show_gauss_pyramid(pyramid):
140     # Recorre la lista y muestra las imágenes hacia abajo
141     for imagen in pyramid:
142         cv2_imshow(imagen)
143
144 """# Pirámide de Laplace"""
145
146 def subtract(input1, input2):
147     # Por hacer: calcular la resta entre input1 e input2, pixel a pixel
148     x = input1.shape[0]
149     y = input1.shape[1]
150     output = np.zeros((x,y), np.float32)
151     for i in range(x):
152         for j in range(y):
153             output[i][j] = input1[i][j] - input2[i][j]
154     return output
155
156 def add(input1, input2):
157     x, y = input1.shape
```

```
158 output = np.zeros((x,y), np.float32)
159 for i in range(x):
160     for j in range(y):
161         try:
162             output[i][j] += (input1[i][j] + input2[i][j])
163         except:
164             pass
165 return output
166
167 def calc_laplace_pyramid(input, levels):
168     gausspyr = []
169     laplacepyr = []
170     current = np.copy(input)
171     gausspyr.append(current)
172     for i in range(1, levels):
173         # 1) Aplicar apply_blur( ) a la imagen gausspyr[i-1], con sigma 2.0 y ancho 7
174         current = apply_blur(gausspyr[i-1], 2.0, 7)
175         # 2) Guardar en laplacepyr el resultado de restar gausspyr[i-1] y la imagen calculada en (1)
176         resta = subtract(gausspyr[i-1], current)
177         laplacepyr.append(resta)
178         # 3) Submuestrear la imagen calculada en (1), guardar el resultado en current
179         current = do_subsample(current)
180         gausspyr.append(current)
181     laplacepyr.append(current) # Se agrega el ultimo piso de la piramide de Laplace
182 return laplacepyr
183
184 def show_laplace_pyramid(pyramid):
185     """
186     pyramid: arreglo de imágenes
187
188     La función toma este arreglo de entrada y lo recorre píxel a píxel hasta la penúltima
189     posición aplicando el valor absoluto, lo anterior debido a que la última imagen es idéntica a la ú
190     ↪ ltima de la pirámide de Gauss y no
191     requiere de la aplicación de esta función. Finalmente, se recorre nuevamente el arreglo mostrando
192     ↪ las imágenes en pantalla.
193     """
194     pyramid2 = np.copy(pyramid)
195     for k in range(len(pyramid2)-1):
196         for i in range(pyramid2[k].shape[0]):
197             for j in range(pyramid2[k].shape[1]):
198                 (pyramid2[k])[i][j] = np.abs((pyramid2[k])[i][j]) * 6# por factor
199     for imagen in pyramid2:
200         cv2_imshow(imagen)
201
202     """## Reconstrucción"""
203
204     def do_upsample(img):
205         x, y = img.shape
206         upsample = np.zeros((2*x, 2*y), np.float32)
```

```
207     for j in range(2*y-2):
208         if (i\%2==0) and (j\%2==0):
209             upsample[i][j] = img[i//2][j//2]
210         elif (i\%2==0) and (j\%2!=0):
211             upsample[i][j] = (img[i//2][j//2] + img[i//2][(j+1)//2]) / 2
212         elif (i\%2!=0) and (j\%2==0):
213             upsample[i][j] = (img[i//2][j//2] + img[(i+1)//2][j//2]) / 2
214         else:
215             upsample[i][j] = (img[i//2][j//2] + img[(i+1)//2][j//2] + img[i//2][(j+1)//2] + img[(i+1)//2][(j+1)//2]) / 4
216
217     return upsample
218
219 def do_reconstruct(laplacepyr):
220     output = np.copy( laplacepyr[len(laplacepyr)-1] )
221     for i in range(1, len(laplacepyr)):
222         level = int(len(laplacepyr)) - i - 1
223
224         output = do_upsample(output)
225
226         output = add(output, laplacepyr[level])
227         output = output
228     return output
229
230 """# Resultados parte A y B"""
231
232 originalBGR = cv2.imread('corteza_2022.jpg') #Leer imagen
233
234 if originalBGR is None:
235     assert False, 'Imagen no encontrada'
236
237 if len(originalBGR.shape) == 3:
238     original = cv2.cvtColor(originalBGR, cv2.COLOR_BGR2GRAY)
239 else:
240     original = originalBGR
241
242 input = np.float32( original )
243
244
245 print('Piramide de gauss:')
246 gausspyramid = calc_gauss_pyramid(input, 5)
247 show_gauss_pyramid(gausspyramid)
248
249
250
251 laplacepyramid = calc_laplace_pyramid(input, 5)
252
253 print('reconstruida:')
254 reconstr = do_reconstruct(laplacepyramid)
255 cv2_imshow(reconstr)
256
```

```
257
258 print('Piramide de laplace:')
259 show_laplace_pyramid(laplacepyramid)
260
261 """# Filtrado Pasa-Altos
262
263 ## Laplacian of Gaussian (LoG)
264 """
265
266 def LoG(sigma, width):
267     LoG_X = np.zeros((width, width), np.float32)
268     LoG_Y = np.zeros((width, width), np.float32)
269     center = width // 2
270     ax = np.linspace(-center, center, width)
271     for j in range(len(ax)):
272         x = ax[j]
273         LoG_X[center][j] = (((x**2) / (sigma**4)) - (1/(sigma**2))) * np.exp(-0.5 * (x**2) / (sigma**2))
274     for i in range(len(ax)):
275         y = ax[i]
276         LoG_Y[i][center] = (((y**2) / (sigma**4)) - (1/(sigma**2))) * np.exp(-0.5 * (y**2) / (sigma**2))
277     return LoG_X / np.sum(np.abs(LoG_X)), LoG_Y / np.sum(np.abs(LoG_Y))
278
279 def apply_LoG(img, sigma, width):
280     r, c = img.shape
281     output = np.zeros((r,c), np.float32)
282
283     LoG_X, LoG_Y = LoG(sigma, width)
284
285     Dx = convolution_cython(img, LoG_X)
286     Dy = convolution_cython(img, LoG_Y)
287
288     output = np.sqrt(np.square(Dx) + np.square(Dy))
289     return output
290
291 originalBGR = cv2.imread('corteza_2022.jpg') #Leer imagen
292
293 if originalBGR is None:
294     assert False, 'Imagen no encontrada'
295
296 if len(originalBGR.shape) == 3:
297     original = cv2.cvtColor(originalBGR, cv2.COLOR_BGR2GRAY)
298 else:
299     original = originalBGR
300
301 input = np.float32( original )
302 out = apply_LoG(input, 1.4, 15)
303 cv2_imshow(out)
304
305 """## Derivative of Gaussian (DoG)"""
306
307 def DoG(sigma, width):
```

```
308 DoG_X = np.zeros((width, width), np.float32)
309 DoG_Y = np.zeros((width, width), np.float32)
310 center = width // 2
311 ax = np.linspace(-center, center, width)
312 for j in range(len(ax)):
313     x = ax[j]
314     DoG_X[center][j] = (-x/(sigma**2)) * np.exp(-0.5 * (x**2) / (sigma**2))
315     for i in range(len(ax)):
316         y = ax[i]
317         DoG_Y[i][center] = (-y/(sigma**2)) * np.exp(-0.5 * (y**2) / (sigma**2))
318     return DoG_X / np.sum(np.abs(DoG_X)), DoG_Y / np.sum(np.abs(DoG_Y))
319
320 def apply_DoG(img, sigma, width):
321     r, c = img.shape
322     output = np.zeros((r,c), np.float32)
323
324     DoG_X, DoG_Y = DoG(sigma, width)
325
326     Dx = convolution_cython(img, DoG_X)
327     Dy = convolution_cython(img, DoG_Y)
328
329     output = np.sqrt(np.square(Dx) + np.square(Dy))
330     return output
331
332 originalBGR = cv2.imread('corteza_2022.jpg') #Leer imagen
333
334 if originalBGR is None:
335     assert False, 'Imagen no encontrada'
336
337 if len(originalBGR.shape) == 3:
338     original = cv2.cvtColor(originalBGR, cv2.COLOR_BGR2GRAY)
339 else:
340     original = originalBGR
341
342 input = np.float32( original )
343 out = apply_DoG(input, 1.4, 9)
344 cv2_imshow(out)
345
346 """## Filtrado v/s Laplace"""
347
348 originalBGR = cv2.imread('corteza_2022.jpg') #Leer imagen
349
350 if originalBGR is None:
351     assert False, 'Imagen no encontrada'
352
353 if len(originalBGR.shape) == 3:
354     original = cv2.cvtColor(originalBGR, cv2.COLOR_BGR2GRAY)
355 else:
356     original = originalBGR
357
358 input = np.float32( original )
```

```
359 gauss = calc_gauss_pyramid(input, 5)
360 #laplace = calc_laplace_pyramid(input, 5)
361
362 P_filtrada_LoG = []
363 P_filtrada_DoG = []
364
365 for i in range(len(gauss)):
366     P_filtrada_LoG.append(apply_LoG(gauss[i], 1.4, 15))
367     P_filtrada_DoG.append(apply_DoG(gauss[i], 1.4, 9))
368
369 print('Pirámide Filtrada LoG')
370 for img in P_filtrada_LoG:
371     cv2_imshow(img)
372
373 print('Pirámide Filtrada DoG')
374 for img in P_filtrada_DoG:
375     cv2_imshow(img)
```