

# Tarea 1

Pirámides de Gauss y Laplace

Integrantes: Joaquín Zepeda  
Profesor: Javier Ruiz del Solar  
Auxiliar: Patricio Loncomilla  
Santiago de Chile

# Índice de Contenidos

<b>1. Introducción</b>	<b>1</b>
<b>2. Marco Teórico</b>	<b>2</b>
<b>3. Desarrollo</b>	<b>6</b>
3.1. Parte A: Cálculo de pirámides de Gauss . . . . .	6
3.2. Parte B: Cálculo de pirámides de Laplace . . . . .	8
3.2.1. Reconstrucción de una imagen a partir de las pirámides . . . . .	9
3.3. Parte C: Filtrados Pasaaltos . . . . .	9
3.3.1. Filtrado pasa altos imágenes de prueba . . . . .	10
3.3.2. Filtrado pasa altos Pirámides de Gauss . . . . .	12
<b>4. Conclusión</b>	<b>15</b>
<b>5. Anexos</b>	<b>16</b>
5.1. Parte A: Cálculo de pirámides de Gauss . . . . .	16
5.2. Parte B: Cálculo de pirámides de Laplace . . . . .	19
5.3. Reconstrucción de las imágenes . . . . .	22
5.4. Filtros . . . . .	23
5.5. Código . . . . .	25
<b>Referencias</b>	<b>36</b>

# Índice de Figuras

1. Distribuciones utilizadas para generar los filtros. Es importante destacar que los filtros se realizan con valores discretos de estas distribuciones, por lo que al graficarlos son menos suaves. . . . .	3
2. Kernel Gaussiano. . . . .	3
3. Ejemplo de los kernels generados DoG. . . . .	4
4. Ejemplo de los kernels generados LoG. . . . .	4
5. Pirámide de Gauss de la imagen Origami . . . . .	7
6. Pirámide de Laplace de la imagen Origami . . . . .	8
7. Imágenes reconstruidas. . . . .	9
8. Filtrado a la imagen Origami . . . . .	10
9. Filtrado a la imagen Corteza . . . . .	11
10. Filtrado a la imagen Dali . . . . .	11
11. Filtrado a la imagen Techo falso . . . . .	11
12. Pirámide filtrada utilizando el filtro pasa alto con DoG. Utilizando un filtro y una desviación grandes ( $\sigma = 5$ , $width = 100$ ) . . . . .	12
13. Pirámide filtrada utilizando el filtro pasa alto con DoG. Utilizando los tamaños finales de los filtros y de las desviaciones. . . . .	13
14. Dali filtrada pasa altas. . . . .	14
15. Pirámide de Gauss de la imagen Dali . . . . .	16

16.	Pirámide de Gauss de la imagen Corteza . . . . .	17
17.	Pirámide de Gauss de la imagen Techo . . . . .	18
18.	Pirámide de Laplace de la imagen Dali . . . . .	19
19.	Pirámide de Laplace de la imagen Corteza . . . . .	20
20.	Pirámide de Laplace de la imagen Techo . . . . .	21
21.	Reconstrucción de la imagen Origami. . . . .	22
22.	Pirámide de Gauss filtrada pasa altos. . . . .	23
23.	Pirámide de Gauss filtrada pasa altos. . . . .	24
24.	Pirámide de Gauss filtrada pasa altos. . . . .	25

## Índice de Códigos

1.	Convolución . . . . .	6
2.	Código filtro pasa alto . . . . .	9
3.	Código completo . . . . .	25

## 1. Introducción

El procesamiento digital de imágenes tiene un papel importante en la sociedad, siendo estas una base para el reconocimiento de objetos/personas, diagnosticar condiciones médicas, astronomía, etc. este corresponde a un conjunto de técnicas que permiten cambiar la información que contiene la imagen con el fin de tener una mejor representación de esta y/o resaltar o suprimir alguna característica. Esta disciplina dio sus comienzos cuando comenzaron a digitalizarse las imágenes, es decir, se representaron las imágenes mediante matrices las cuales se guardan en memoria. Dentro de computadoras se pueden operar técnicas matemáticas y cálculos los cuales en la actualidad se puede realizar medianamente rápido, convirtiéndose en una disciplina muy útil en múltiples áreas.

El objetivo de esta tarea es implementar representaciones multi-resolución de imágenes, en específico implementar las pirámides de Gauss y Laplace, y además realizar un filtrado de tipo pasa alto a diferentes imágenes, esto con el fin de poner en práctica los conceptos vistos en clases y poder apreciar en diferentes situaciones los resultados de estos métodos.

Esta tarea se desarrolla utilizando el lenguaje de programación Python, a continuación en la sección del Marco teórico se describen conceptos claves y los algoritmos que se utilizan en el desarrollo de la tarea, luego se muestran y analizan los resultados en la sección de Desarrollo para luego finalizar con las conclusiones.

## 2. Marco Teórico

A continuación se definen conceptos claves para comprender y dar contexto al desarrollo de esta Tarea:

- Imágenes digitales: Una imagen se puede representar de forma bidimensional por una matriz que viene dada por la siguiente:

$$p_{ij} = F(x_i, y_j) \quad (1)$$

en donde  $p_{ij}$  corresponde al valor de intensidad, también conocido como pixel y  $x_i$  e  $y_j$  son las coordenadas espaciales.

- Convolución: es una operación matemática con dos funciones

$$f(x) * h(x) = \int_{-\infty}^{\infty} f(x)h(u - x)dx \quad (2)$$

La convolución de matrices tiene un amplio uso en el filtrado de imágenes, consiste en modificar la matriz mediante convoluciones con otra matriz que se denominará kernel de convolución. Para poder realizar la convolución de matrices se modifica la ecuación 2 para trabajar con matrices discretas de 2 dimensiones. Dependiendo del kernel que se utilice, la imagen resultante presenta diferentes resultados.

- Filtros: Para el desarrollo de la tarea, es necesario generar un filtro Gaussiano (generado a partir de una distribución Gaussiana) y además la primera y segunda derivada de este filtro Gaussiano [1], para esto se presenta la distribución Gaussiana en una dimensión (1D) corresponde a la siguiente expresión:

$$g(x, \sigma) = \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{x^2}{2\sigma^2}} \quad (3)$$

Luego esta se puede llevar a dos dimensiones (2D) y además para que la indexación sea la correcta se le resta el centro “c” a cada elemento de la imagen.

$$g(x, y, \sigma) = \frac{1}{\sigma^2 2\pi} e^{-\frac{(x-c)^2 + (y-c)^2}{2\sigma^2}} \quad (4)$$

Para determinar el Derivative of Gaussian (DoG), se determinan por separado la derivada con respecto a “x” y la derivada con respecto a “y”.

$$\frac{\partial g(x, y, \sigma)}{\partial x} = \frac{-((x - center))e^{-\frac{(x-c)^2 + (y-c)^2}{2\sigma^2}}}{\sigma^4 2\pi} \quad (5)$$

$$\frac{\partial g(x, y, \sigma)}{\partial y} = \frac{-((y - center))e^{-\frac{(x-c)^2 + (y-c)^2}{2\sigma^2}}}{\sigma^4 2\pi} \quad (6)$$

Finalmente, para determinar el Laplacian of Gaussian (LoG), se determinan por separado la derivada con respecto a “x” y la derivada con respecto a “y”.

$$\frac{\partial^2 g(x, y, \sigma)}{\partial^2 x} = \frac{(-1 + \frac{(x - center)^2}{(\sigma^2)}) e^{-\frac{(x-c)^2+(y-c)^2}{2\sigma^2}}}{\sigma^4 2\pi} \quad (7)$$

$$\frac{\partial^2 g(x, y, \sigma)}{\partial^2 y} = \frac{(-1 + \frac{(y - center)^2}{(\sigma^2)}) e^{-\frac{(x-c)^2+(y-c)^2}{2\sigma^2}}}{\sigma^4 2\pi} \quad (8)$$

En base a estas ecuaciones se generan los filtros que luego se utilizan para el procesamiento de imágenes, a continuación se muestra una representación gráfica de la distribución de estos:

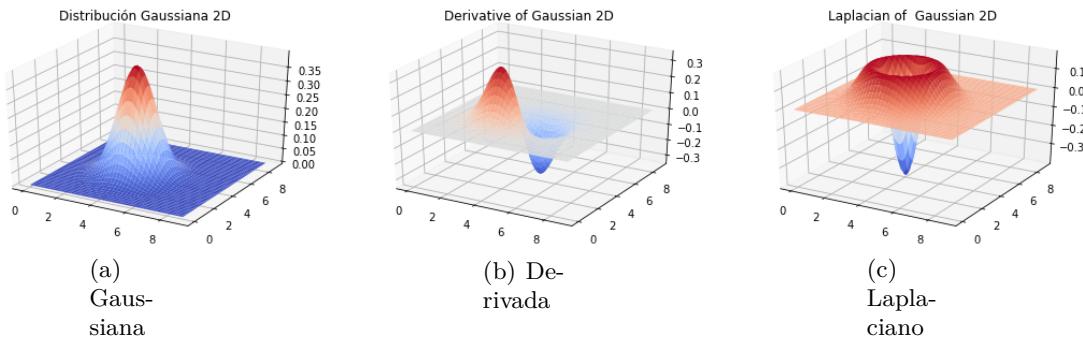


Figura 1: Distribuciones utilizadas para generar los filtros. Es importante destacar que los filtros se realizan con valores discretos de estas distribuciones, por lo que al graficarlos son menos suaves.

A partir de estas ecuaciones, se generan los kernels o filtros que corresponden a matrices que siguen una distribución correspondiente. Estos se pueden observar en las siguientes figuras:

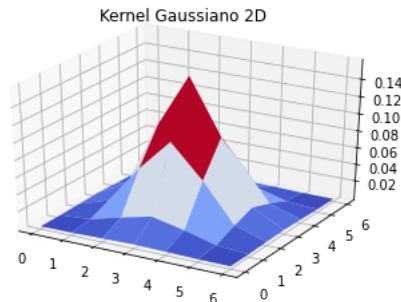


Figura 2: Kernel Gaussiano.

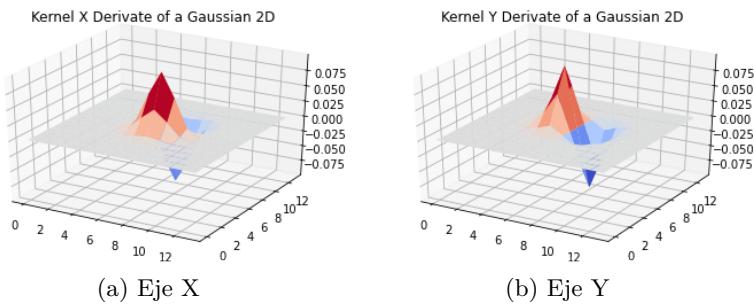


Figura 3: Ejemplo de los kernels generados DoG.

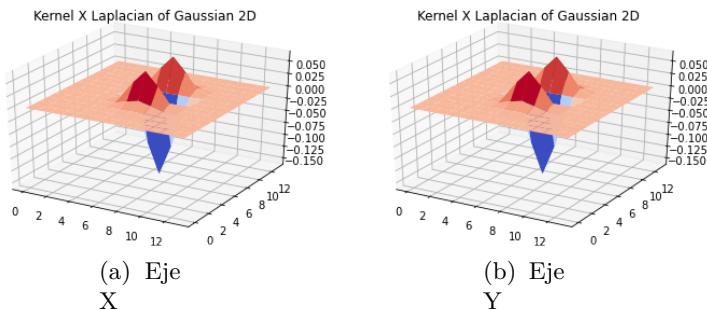


Figura 4: Ejemplo de los kernels generados LoG.

- Pirámide de Gauss: La pirámide de Gauss contiene varias imágenes que representan distintas resoluciones posibles para una misma imagen. Se le aplica un filtro Gaussiano para suavizar la imagen antes de submuestrear.
  - Pirámide de Laplace: contiene la información que se pierde al ir bajando la resolución de la imagen original en el proceso de formación de la pirámide de Gauss. Se le aplica un filtro Gaussiano a la imagen para luego restar la imagen suavizada con la imagen sin suavizar, dejando de esta manera una imagen que resalta las zonas no suaves, es decir, los bordes.
  - Algoritmo cálculo Pirámide de Gauss: primero se lleva la imagen a escala de grises, luego se suaviza la imagen aplicando un filtro Gaussiano (el cual se aplica realizando la convolución del filtro con la imagen), luego de esto se realiza un subsampleo de la imagen con la función 'do\_subsample', lo cual consiste en reducir el tamaño de esta seleccionando solo las filas y columnas pares (disminuyendo el tamaño de la imagen a la mitad), para finalmente almacenar la imagen resultante en la pirámide.
  - Algoritmo cálculo Pirámide de Laplace: A partir de una pirámide de Gauss, se selecciona un nivel de la pirámide, se suaviza la imagen seleccionada con un filtro gaussiano para luego generar una nueva imagen a partir de la resta entre la imagen sin suavizar y la imagen suavizada, de esta manera se resaltan los bordes de las imágenes. Finalmente se almacena la imagen en la pirámide de Laplace.
  - Reconstruir una Imagen: para reconstruir las imágenes a partir de la pirámide de Laplace se sigue el siguiente algoritmo:

1. Se selecciona el último piso de la pirámide de Laplace.
2. Se duplica el tamaño de la imagen con la función 'do\_upsample' para luego sumar la imagen correspondiente de la pirámide de laplace, esto se debe realizar con todas las imágenes de la piramide con el fin de poder reconstruir la imagen original.

### 3. Desarrollo

A continuación se muestran los resultados y los análisis de estos dentro del desarrollo de la Tarea, cabe destacar que se adjuntan algunas imágenes en el desarrollo y las demás imágenes que se obtuvieron se **adjuntan en el anexo**.

#### 3.1. Parte A: Cálculo de pirámides de Gauss

- Parte A.1 Convolución usando Cython: Para la parte A.1 se programó la función 'convolution\_cython' la cual recibe una imagen de entrada en escala de grises y una máscara, para luego calcular la convolución entre ambas para generar una imagen de salida.

```

1  %%cython
2  import cython
3  import numpy as np
4  cimport numpy as np
5
6  # La convolucion debe ser implementada usando cython (solo esta funcion en cython)
7  #@cython.boundscheck(False)
8  cpdef np.ndarray[np.float32_t, ndim=2] convolution_cython(np.ndarray[np.float32_t, ndim=2]
9      ↪ input, np.ndarray[np.float32_t, ndim=2] mask):
10     cdef int rows,cols,m_rows,m_cols
11     cdef int i,j,ii,jj
12     cdef int nn,mm,m_center_x,m_center_y,m,n
13     cdef float sum
14     cdef np.ndarray[np.float32_t, ndim=2] output = np.zeros([input.shape[0], input.shape[1]],
15      ↪ dtype = np.float32)
16
17     # tamano de la imagen
18     rows = input.shape[0]
19     cols = input.shape[1]
20     # tamano de la mascara
21     m_rows = mask.shape[0]
22     m_cols = mask.shape[1]
23     #centros
24     m_center_x = m_rows//2
25     m_center_y = m_cols//2
26     # Por hacer: implementar convolucion entre "input" y "mask"
27     for i in range(rows):
28         for j in range(cols):
29             for m in range(m_rows):
30                 mm = m_rows-1-m
31                 for n in range(m_cols):
32                     nn = m_cols-1-n
33                     ii = i + (m - m_center_y)
34                     jj = j + (n - m_center_x)
35                     if( ii >= 0 and ii < rows and jj >= 0 and jj<cols):
36                         output[i,j] += input[ii,jj]*mask[mm,nn]
37
38     return output

```

Código 1: Convolución

Luego de esto, utilizando las función 'compute\_gauss\_mask\_2d' se genera una mascara Gaussiana a partir de la ecuación 4. Como se mencionó en el marco teórico (donde se mencionó el funcionamiento del algoritmo), la imagen se suaviza y luego se subsamplea para luego ir agregándose a la pirámide de Gauss. A continuación, en la imagen 5, se muestra un ejemplo del resultado de la pirámide de Gauss con la imagen del Origami, los demás resultados se adjuntan en el anexo.

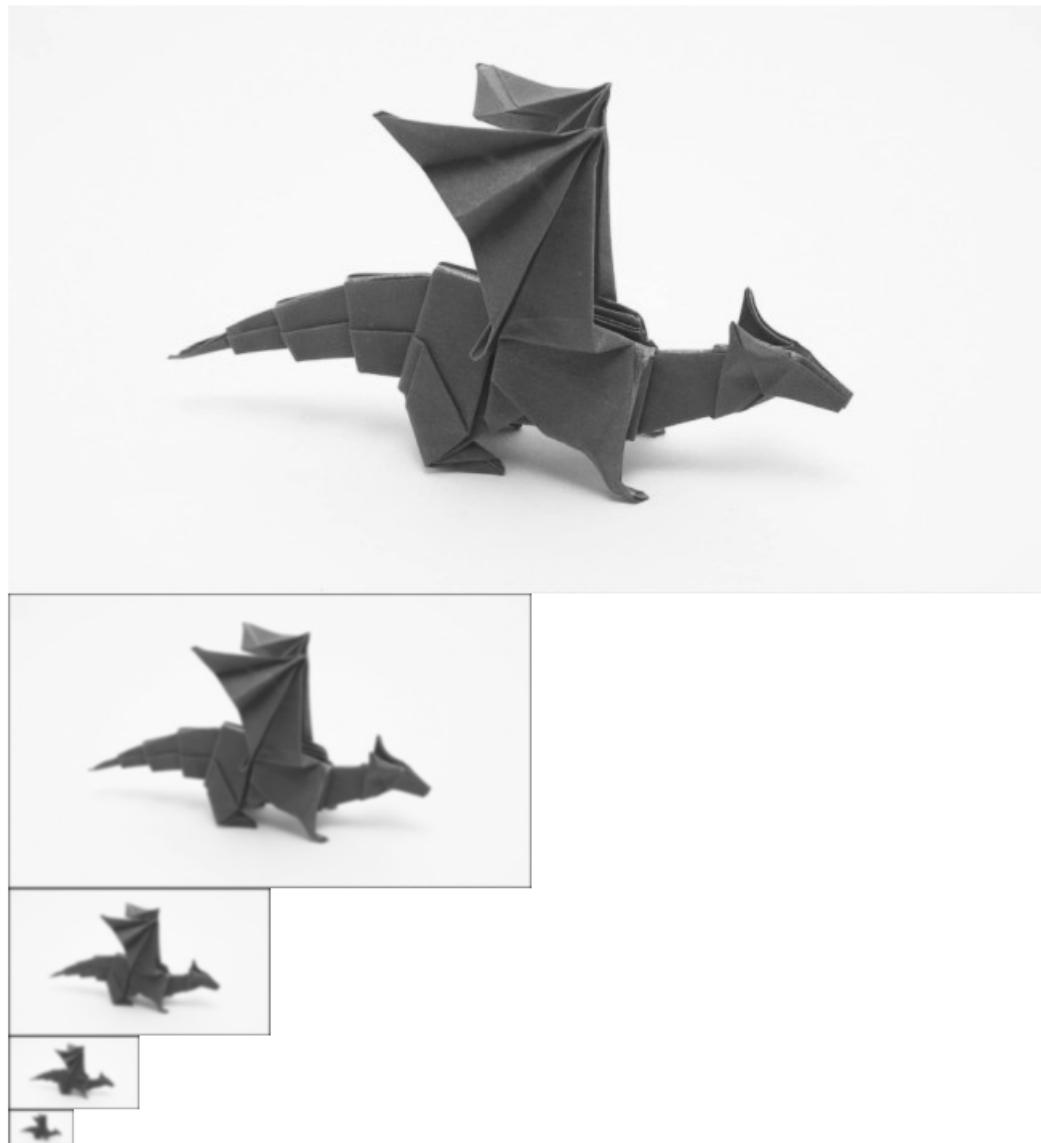


Figura 5: Pirámide de Gauss de la imagen Origami

Al analizar los resultados, tanto de la imagen 5 como de las 15,16 y 17, se puede observar que al disminuir el tamaño de las imágenes y al aplicar el filtro Gaussiano estás se vuelven más borrosas y pierden nitidez, a pesar de esto, se sigue distinguiendo la forma de las imágenes lo cual se debe al filtrado previo. Las imágenes de 15 y 16 son las que más se ven afectadas al aplicar este proceso.

### 3.2. Parte B: Cálculo de pirámides de Laplace

Se siguió el proceso explicado en el marco teórico. A continuación, en la imagen 6, se muestra un ejemplo del resultado de la pirámide de Laplace con la imagen del Origami, los demás resultados se adjuntan en el anexo.

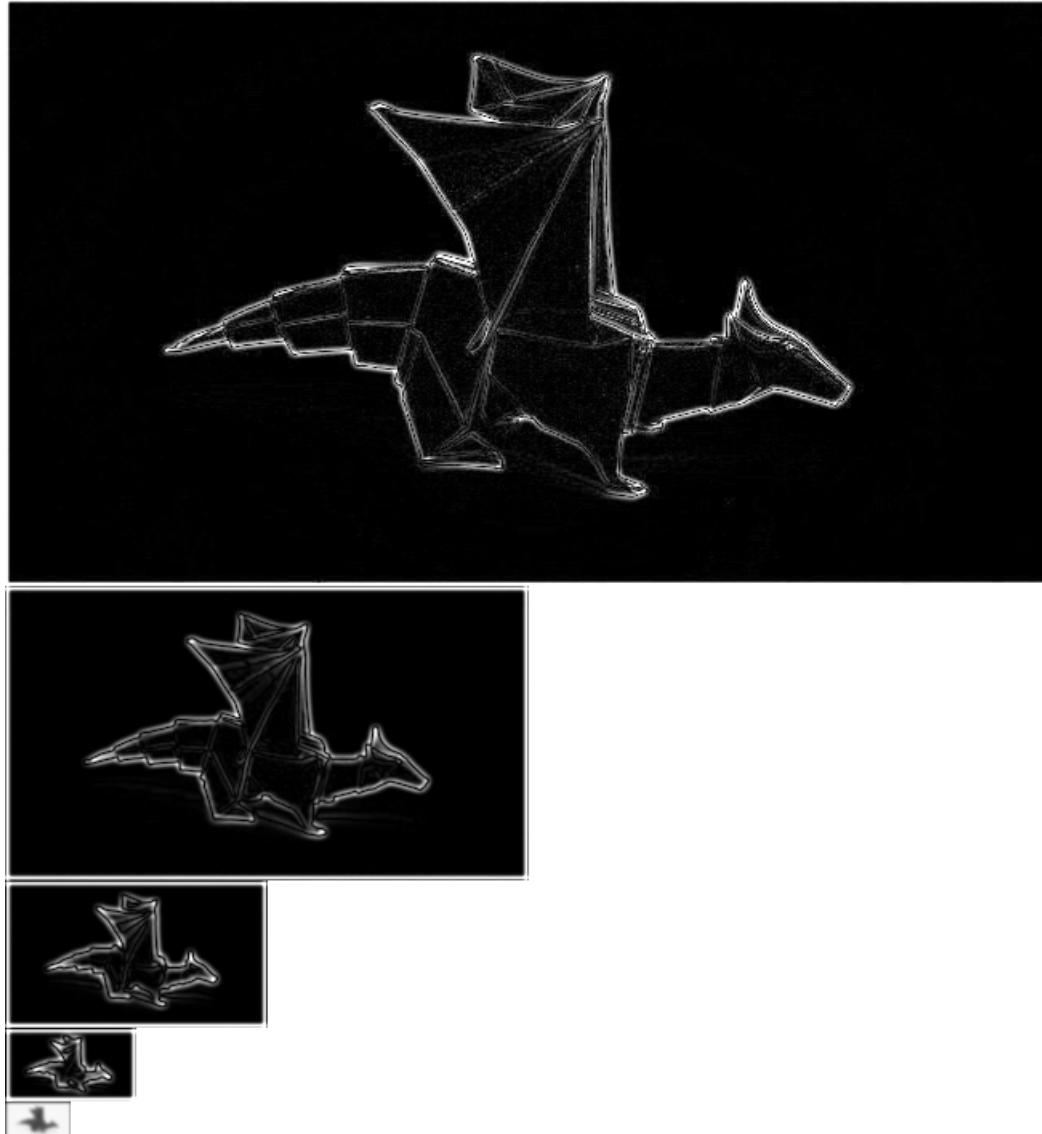


Figura 6: Pirámide de Laplace de la imagen Origami

Al analizar los resultados, tanto de la imagen 6 como de las 18 y 20, se pueden distinguir los bordes de las figuras y objetos de las imágenes de forma clara, lo cual no ocurre en la imagen 19, esto se debe a que la imagen de la corteza se detectan muchos bordes lo cual no deja que se distingan bien los bordes principales. A medida que se bajan en los niveles de la pirámide (suponiendo que la imagen original es el nivel 1), los bordes van perdiendo nitidez y se vuelven borrosos (debido al filtro pasa bajos que se le aplica) pero al menos en los primeros 3 niveles se sigue observando de forma

clara los bordes.

### 3.2.1. Reconstrucción de una imagen a partir de las pirámides

Para reconstruir la imagen fue necesario realizar un ajuste de tamaños de estas, esto se debe a que la imagen up\_sampleada en algunos casos no tenía las mismas dimensiones que la imagen anterior de la pirámide (esto debido a las imágenes de tamaños impares), para esto se realizó un resize (ajuste de tamaño) de la imagen de forma manual, lo cual permitió sumar las imágenes pero produjo que los bordes derecho e inferior se ven oscurecidos. Otra solución para el problema de las dimensiones pudo haber sido modificar la función suma para que sumará las matrices recorriendo la matriz de menor tamaño. A continuación en la figura 7 se muestran los resultados de las imágenes reconstruidas. Además la reconstrucción de la imagen Origami se encuentra en el anexo en la figura 21.

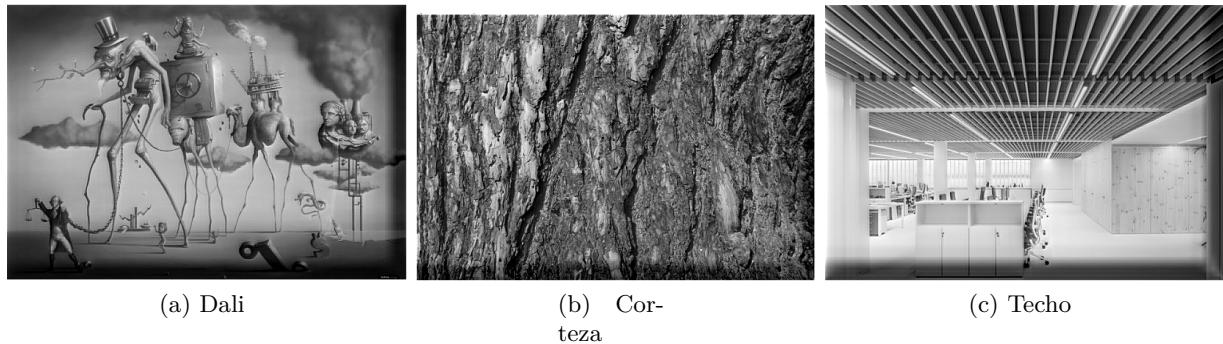


Figura 7: Imágenes reconstruidas.

Al observar el resultado de la imagen reconstruida se observa que a pesar de que los bordes derecho e inferior estén oscurecidos, la imagen se logra distinguir de forma nítida, si bien no se reconstruye como la imagen original, el resultado es bastante similar.

### 3.3. Parte C: Filtrados Pasaaltos

En esta sección se programó la función 'pasa\_alto' la cual recibe como parámetros una imagen en escala de grises, la desviación estándar y el ancho de la mascara y un factor el cual solo se utilizó para mejorar la visualización de los resultados. Se realizaron 2 convoluciones (una en cada eje) y luego se combinaron utilizando el valor absoluto de ambos resultados pixel por pixel.

```

1 def pasa_alto(img,tipo_filtro,sigma,width,factor=5):
2     output = np.zeros((img.shape[0], img.shape[1]), np.float32)
3     if tipo_filtro=="1d":
4         dgmask_x = compute_dgauss_mask_2d(sigma, width,"x")
5         dgmask_y = compute_dgauss_mask_2d(sigma, width,"y")
6     else:
7         dgmask_x = compute_d2gauss_mask_2d(sigma, width,"x")
8         dgmask_y = compute_d2gauss_mask_2d(sigma, width,"y")
9     output_x = convolution_cython(img, dgmask_x)
10    output_y = convolution_cython(img, dgmask_y)
11    for x in range(output.shape[0]):
```

```

12     for y in range(output.shape[1]):
13         #se le agrega un factor solo para mostrar de mejor manera los resultados
14         output[x][y] = np.sqrt(output_x[x][y]**2+output_y[x][y]**2)*factor
15     return output

```

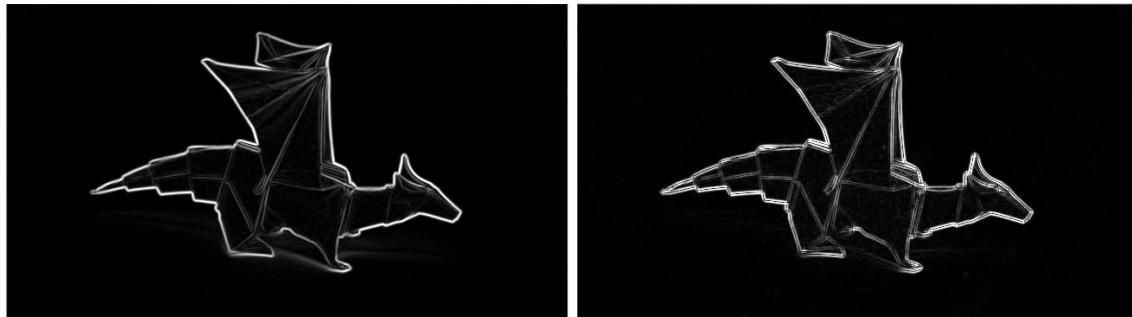
Código 2: Código filtro pasa alto

Además del código principal se generaron las funciones 'compute\_dgauss\_mask\_2d' y 'compute\_d2gauss\_mask\_2d' las cuales calculan la mascara de la derivada de la Gaussiana y de la segunda derivada de la Gaussiana respectivamente (DoG y Log) para cada eje (se debe especificar el eje de la derivada). Esto se realizó utilizando las ecuaciones 5, 6 para DoG y 7, 8 para Log.

### 3.3.1. Filtrado pasa altos imágenes de prueba

Los resultados al aplicar los filtros en las imágenes de prueba se encuentran a continuación, en estas se utilizo un  $\sigma = 1$  y un kernel de tamaño 5 para DoG y se utilizo un  $\sigma = 1$  y un kernel de tamaño 15 para LoG. El uso de una ventana de mayor tamaño para LoG se debe a que cuando se utilizaban menores valores la imagen no sufría mayores cambios, no era posible identificar los bordes. A pesar de esto los resultados fueron bastante similares, puede que esta diferencia se deba a algún error en la programación de los filtros.

Por otro lado los resultados del filtrado fueron mejores con DoG que con LoG, estos últimos estaban más borrosos lo cual se puede deber a una



(a) DoG Imagen Origami

(b) Log Imagen Origami

Figura 8: Filtrado a la imagen Origami

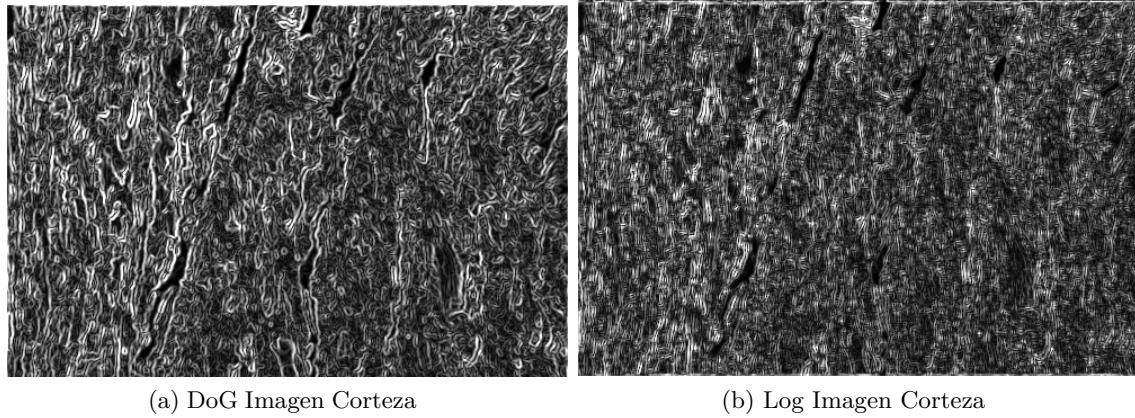


Figura 9: Filtrado a la imagen Cortezá



Figura 10: Filtrado a la imagen Dali

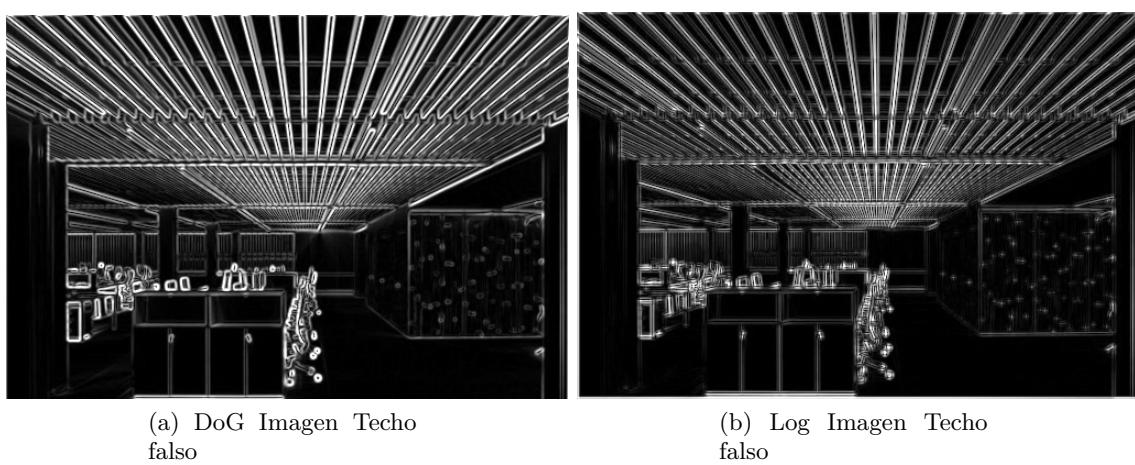


Figura 11: Filtrado a la imagen Techo falso

### 3.3.2. Filtrado pasa altos Pirámides de Gauss

Dependiendo de los pisos de la pirámide de Gauss a los cuales se les aplica el filtrado, los resultados varían. Si se ocupan desviaciones muy pequeñas ( $\sigma < 1$ ) para LoG la imagen filtrada queda blanca, esto se debe a la misma formula del Laplaciano, lo cual luego de la convolución quedan valores muy altos (sobre 256 incluso). Por esto es importante la elección de los sigmas. Con respecto al tamaño de la desviación, se probaron varias combinaciones en las cuales se determino que una desviación de tamaño uno ( $\sigma = 1$ ) produce muy buenos resultados para el DoG.

Para desviaciones muy grandes ( $\sigma > 3$ ) las imágenes resultantes perdían nitidez, incluso resultaban ser completamente oscuras. Perdiendo la información que se buscaba rescatar con el filtrado,

Uno de los problemas de aumentar el tamaño del kernel es el hecho de que la convolución necesita realizar más operaciones por lo que se vuelve más lento el proceso de aplicar el filtrado a cada imagen, por otro lado aumentar el tamaño del kernel no mejora el resultado del filtrado al menos para las imágenes con las que se esta trabajando.



Figura 12: Pirámide filtrada utilizando el filtro pasa alto con DoG. Utilizando un filtro y una desviación grandes ( $\sigma = 5$ ,  $width = 100$  )

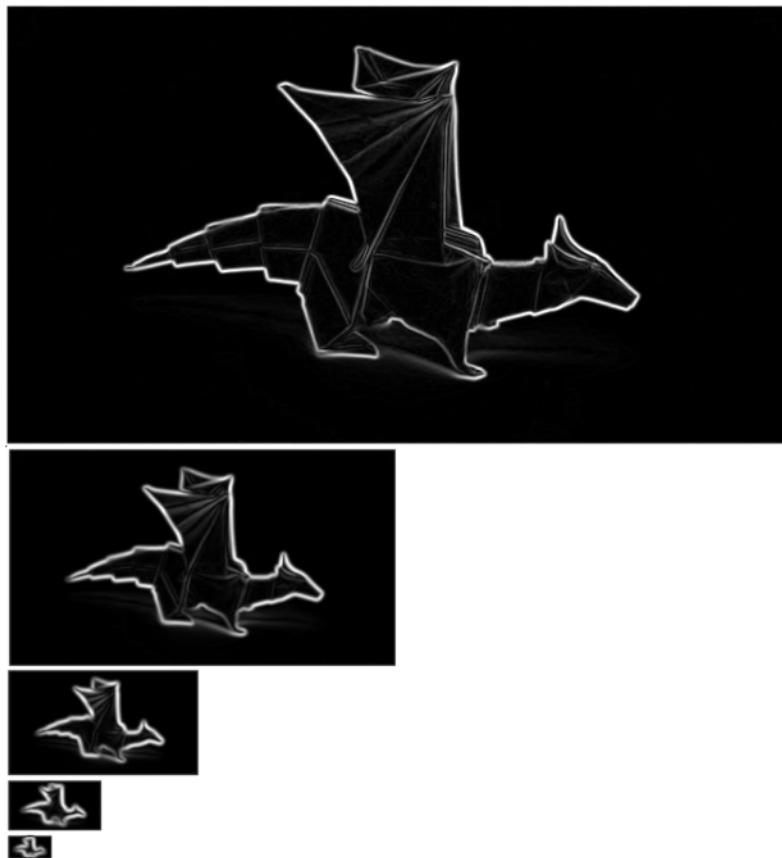


Figura 13: Pirámide filtrada utilizando el filtro pasa alto con DoG. Utilizando los tamaños finales de los filtros y de las desviaciones.

De todas maneras la imagen de Dali y la imagen de corteza, tuvo peores resultados en sus niveles inferiores debido a su gran cantidad de bordes, el resultado se puede observar la figura 14.



Figura 14: Dali filtrada pasa altas.

## 4. Conclusión

Luego de desarrollar esta tarea, se logró comprender la importancia de las Pirámides de Gauss y Laplace, además de los efectos de los filtros y sus parámetros dentro de diferentes imágenes y distintas resoluciones de ellas, se logró realizar un programa el cual cambia la resolución y el tamaño de una imagen logrando mantener la información de esta, detectar los bordes de una imagen y generar máscaras Gaussianas y sus derivadas. Estas herramientas son bastante útiles para nuestra formación como ingenieros. Si bien se obtuvieron buenos resultados en general, algunas de los resultados fueron un poco peor de los esperados, por ejemplo el filtro LoG tuvo resultados que detectaban los bordes pero dejando un poco difusos los resultados, por otro lado la imagen reconstruida pierde un poco información por los límites impares de los tamaños de algunas imágenes, estos problemas se pueden deber a errores en la programación.

Por otro lado, fue posible poner en práctica los conceptos y técnicas vistas en clases, programarlas desde cero en Python e incluso programar con Cython lo cual permitió mejorar los tiempos de ejecución del programa, la utilización de Cython es más importante de lo que uno piensa pues el procesamiento de imágenes puede tomar mucho tiempo en caso de programarse solo con Python (convirtiendo poco eficiente las implementaciones). La mayor dificultad que se encontró al desarrollar la tarea fue la de hacer calzar las dimensiones de las imágenes cuando se realizaba el respectivo `sub_sample` o `up_sample`, en el caso de las imágenes con tamaño impar al tratar de reconstruir la imagen con la pirámide de Laplace algunas veces las dimensiones no calzaban y esto me tomó días en solucionarlo.

## 5. Anexos

### 5.1. Parte A: Cálculo de pirámides de Gauss

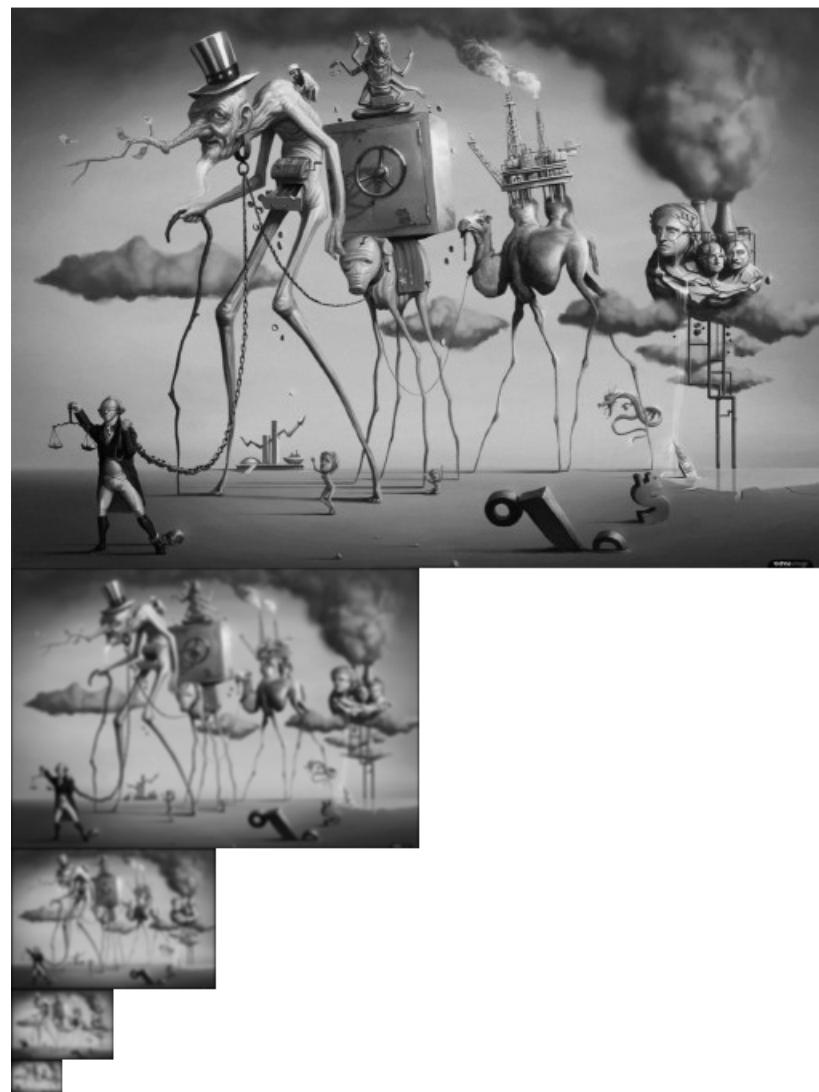


Figura 15: Pirámide de Gauss de la imagen Dali



Figura 16: Pirámide de Gauss de la imagen Corteza



Figura 17: Pirámide de Gauss de la imagen Techo

## 5.2. Parte B: Cálculo de pirámides de Laplace

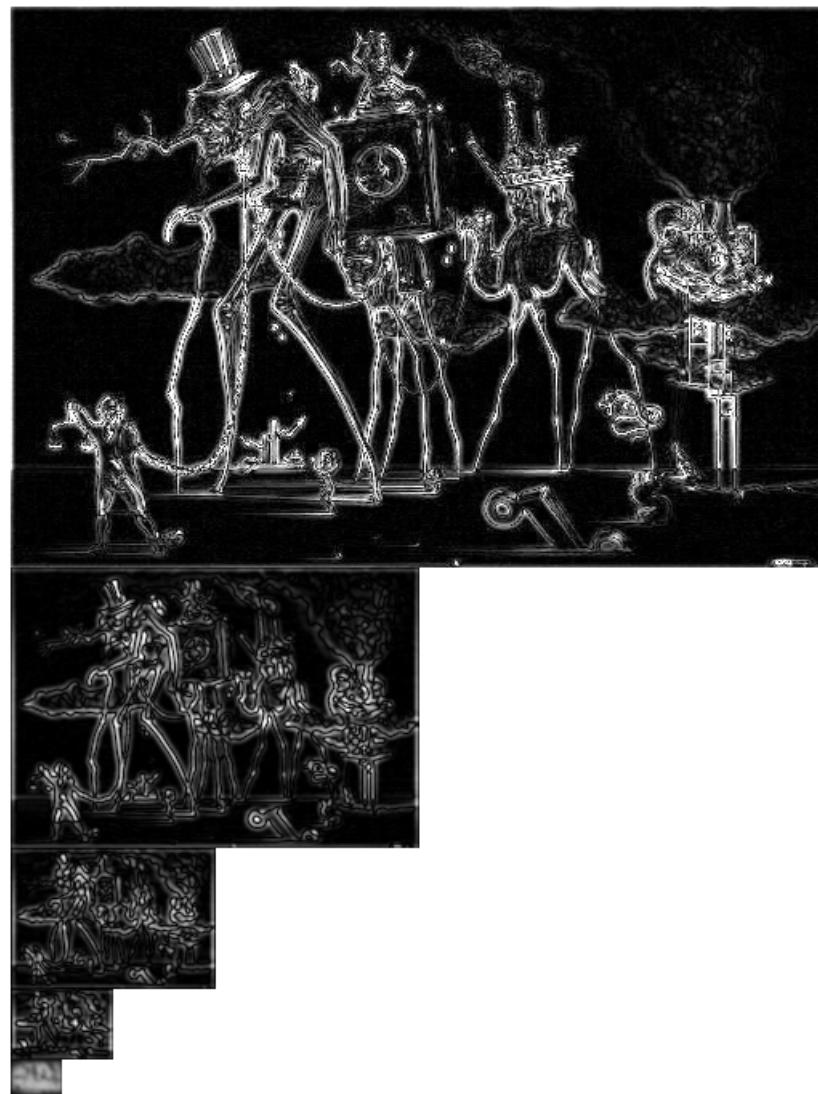


Figura 18: Pirámide de Laplace de la imagen Dali

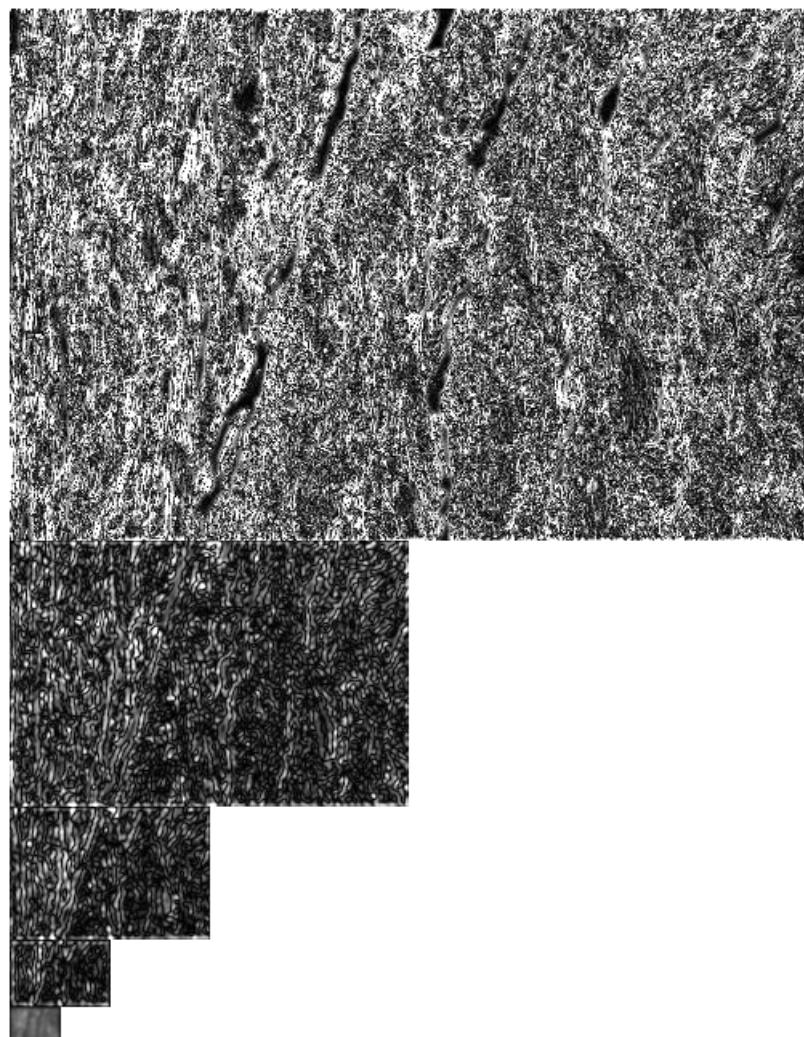


Figura 19: Pirámide de Laplace de la imagen Corteza

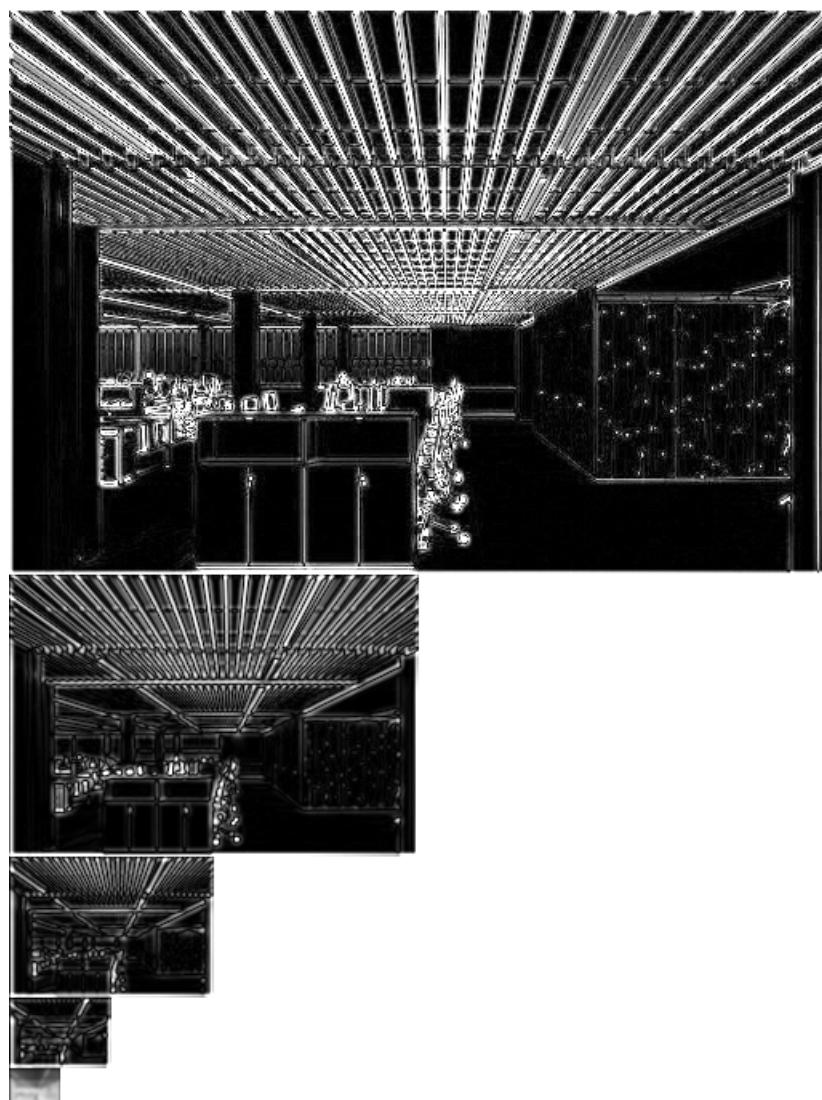


Figura 20: Pirámide de Laplace de la imagen Techo

### 5.3. Reconstrucción de las imágenes

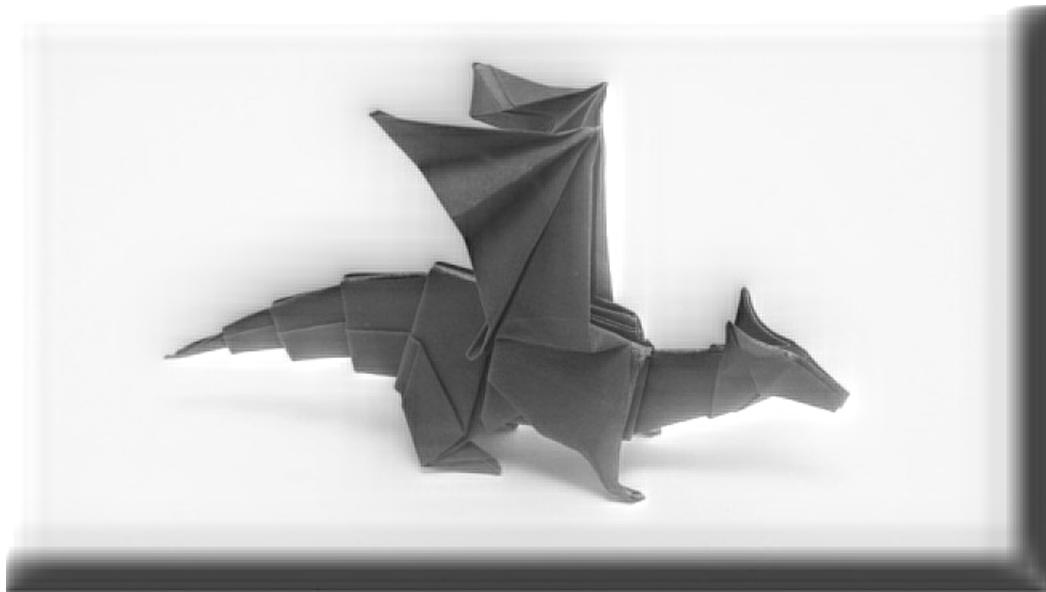


Figura 21: Reconstrucción de la imagen Origami.

## 5.4. Filtros



Figura 22: Pirámide de Gauss filtrada pasa altos.

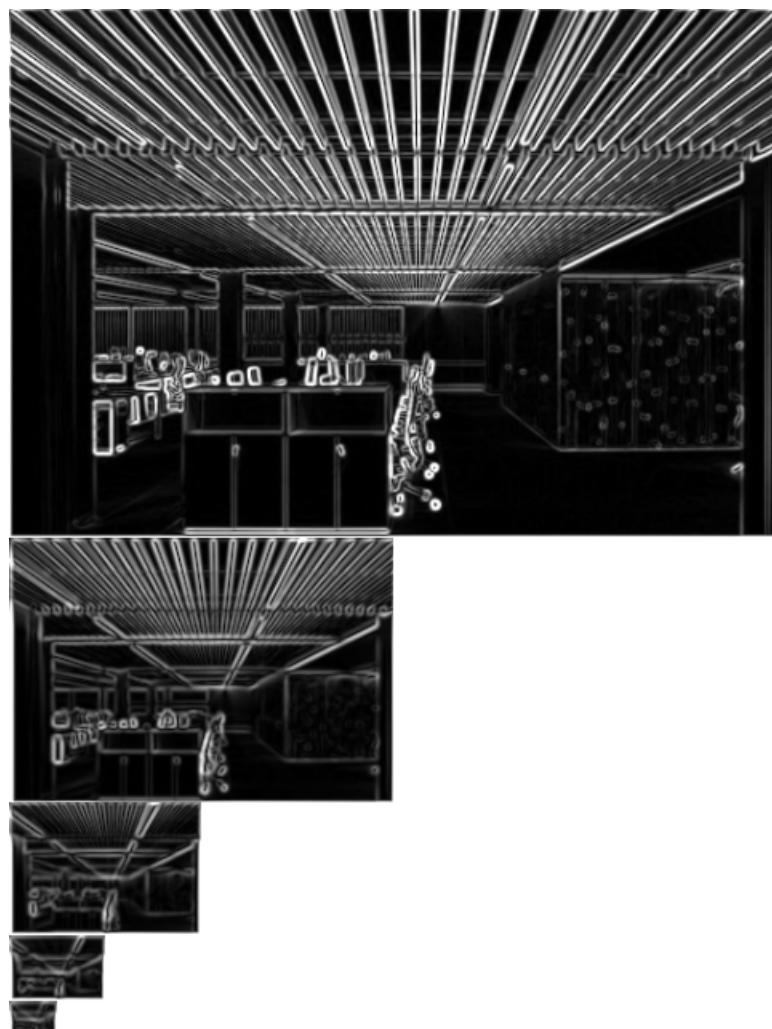


Figura 23: Pirámide de Gauss filtrada pasa altos.

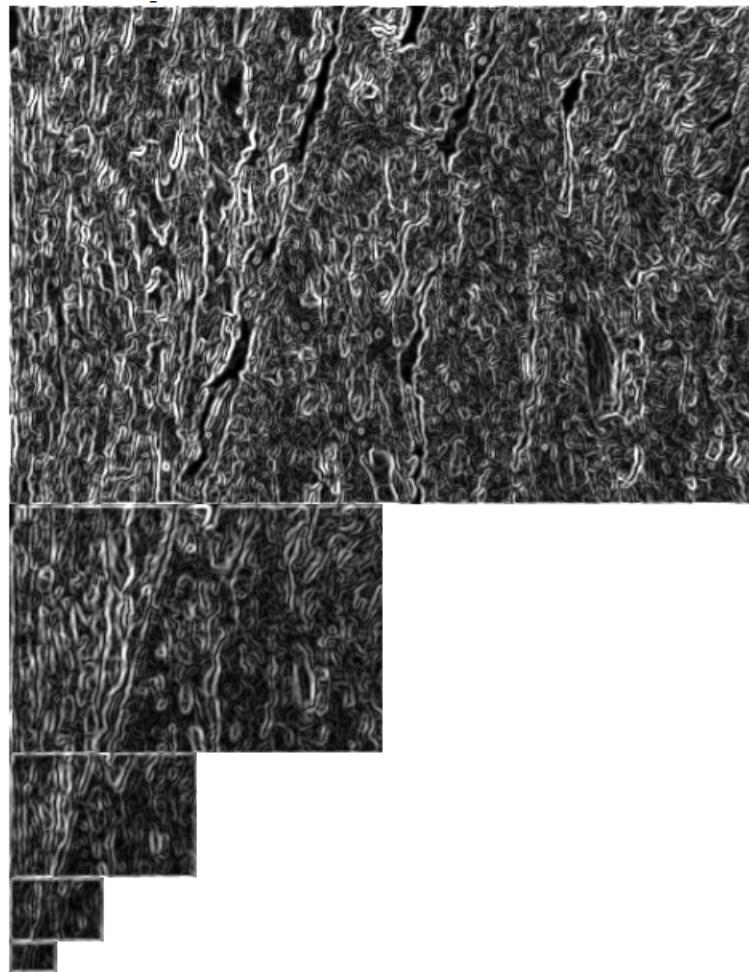


Figura 24: Pirámide de Gauss filtrada pasa altos.

## 5.5. Código

```
1 """tarea1_cython_convolution_JZ.ipynb
2
3 Automatically generated by Colaboratory.
4
5 Original file is located at
6 https://colab.research.google.com/drive/1kmEDIIfIWmYc4l5CxU5sf0f7F2ELTTv6W
7
8 # Pirámides de Gauss y Laplace
9 **Profesor:** Javier Ruiz del Solar \\
10 **Auxiliar:** Patricio Loncomilla \\
11 Estudiante: *Joaquín Zepeda Valero*
12 """
13
14 # El objetivo de esta tarea es:
15 # 1) Implementar convolucion (usando cython)
16 # 2) Implementar piramides de Gauss y Laplace
```

```
17 # 3) Implementar reconstruccion de una imagen a partir de las piramides
18 # 4) Graficar las piramides
19 # 5) Aplicar filtrado pasa alto con laplaciano de gaussiana, y derivadas de gaussiana
20 # 6) Comparar las imagenes filtradas con las de la piramide de laplace
21 #
22 # Nota: los arreglos (matrices) creados deben ser de tipo np.float32
23
24
25 # Mostrar archivos en la carpeta del notebook
26 #!ls
27
28 # Para medir tiempo de ejecucion
29 #!pip install ipython-autotime
30
31 # Commented out IPython magic to ensure Python compatibility.
32 # Extensiones
33 # %load_ext Cython
34 # %load_ext autotime
35
36 # Paquetes a ser usados
37 import numpy as np
38 import cv2
39 import cython
40 import numpy as np
41 import math
42 # Este paquete solo se debe usar si se usa colaboratory
43 from google.colab.patches import cv2_imshow
44
45 # Commented out IPython magic to ensure Python compatibility.
46 # %cython
47 # import cython
48 # import numpy as np
49 # cimport numpy as np
50 #
51 # # La convolucion debe ser implementada usando cython (solo esta funcion en cython)
52 # #@cython.boundscheck(False)
53 # cpdef np.ndarray[np.float32_t, ndim=2] convolution_cython(np.ndarray[np.float32_t, ndim=2]
54 #     ↪ input, np.ndarray[np.float32_t, ndim=2] mask):
55 #     cdef int rows,cols,m_rows,m_cols
56 #     cdef int i,j,ii,jj
57 #     cdef int nn,mm,m_center_x,m_center_y,m,n
58 #     cdef float sum
59 #     cdef np.ndarray[np.float32_t, ndim=2] output = np.zeros([input.shape[0], input.shape[1]], dtype
60 #         ↪ = np.float32)
61 #
62 #     # tamano de la imagen
63 #     rows = input.shape[0]
64 #     cols = input.shape[1]
65 #     #
66 #     # tamano de la mascara
67 #     m_rows = mask.shape[0]
```

```
66 # m_cols = mask.shape[1]
67 #
68 # #centros
69 # m_center_x = m_rows//2
70 # m_center_y = m_cols//2
71 #
72 # # Por hacer: implementar convolucion entre "input" y "mask"
73 # for i in range(rows):
74 #     for j in range(cols): #TENGO QUE REVISAR LOS LIMITES
75 #         for m in range(m_rows):
76 #             mm = m_rows-1-m
77 #             for n in range(m_cols):
78 #                 nn = m_cols-1-n
79 #
80 #                 ii = i + (m - m_center_y)
81 #                 jj = j + (n - m_center_x)
82 #                 if( ii >= 0 and ii < rows and jj >= 0 and jj<cols):
83 #                     output[i,j] += input[ii,jj]*mask[mm,nn]
84 #
85 # return output
86 #
87
88 # Para esta función es importante recordar que la indexación cambia, se busca que el (0,0)
89 # se ubique en el centro. Además para normalizar se divide por la suma total de los valores
90 # de la matriz no normalizada
91
92 # sigma corresponde a la desviación, width el tamaño de la máscara
93 def compute_gauss_mask_2d(sigma, width):
94     gmask = np.zeros((width, width), np.float32)
95     # Por hacer: implementar cálculo de máscara gaussiana 2d pixel a pixel
96     centro = width//2
97     for row in range(width):
98         for col in range(width):
99             gmask[row,col] = np.exp(-((row-centro)**2+(col-centro)**2)/(2*sigma**2))/(2*np.pi*sigma
    **2)
100
101 # Se debe normalizar tras calcularla para que las sumas de los pixeles sea igual a 1
102 return gmask/np.sum(gmask)
103
104 #Prueba
105 from mpl_toolkits.mplot3d import Axes3D
106 from matplotlib import cm
107 import matplotlib.pyplot as plt
108
109 sigma = 1
110 width = 9
111 gmask = compute_gauss_mask_2d(1, width)
112 assert int(np.sum(gmask))==1
113
114 fig = plt.figure()
115 ax = fig.add_subplot(111, projection='3d')
```

```
116 x = y = np.arange(0, width, 0.1)
117 X, Y = np.meshgrid(x, y)
118
119 def gaussian(row,col,centro,sigma):
120     return np.exp(-((row-centro)**2+(col-centro)**2)/(2*sigma**2))/(2*np.pi*sigma**2)
121
122 Z = gaussian(X,Y,width//2,sigma).reshape(X.shape)
123
124 ax.plot_surface(X, Y, Z,cmap=cm.coolwarm)
125 plt.title("Distribución Gaussiana 2D")
126 plt.show()
127
128 #Prueba
129 from mpl_toolkits.mplot3d import Axes3D
130 from matplotlib import cm
131 import matplotlib.pyplot as plt
132
133 sigma = 2
134 width = 7
135 gmask = compute_gauss_mask_2d(1, width)
136 assert int(np.sum(gmask))==1
137
138 fig = plt.figure()
139 ax = fig.add_subplot(111, projection='3d')
140 x = y = np.arange(0, width, 1)
141 X, Y = np.meshgrid(x, y)
142
143 Z = gmask.reshape(X.shape)
144
145 ax.plot_surface(X, Y, Z,cmap=cm.coolwarm)
146 plt.title("Kernel Gaussiano 2D")
147 plt.show()
148
149 def apply_blur(input, sigma, width):
150     # Por hacer:
151     # 1) Calcular mascara gaussiana 2d con parametros sigma y width
152     gmask = compute_gauss_mask_2d(sigma, width)
153
154     # 2) Calcular convolucion entre la imagen de entrada "input" y la mascara 2d
155     output = convolution_cython(input, gmask)
156     return output
157
158 def do_subsample(img):
159     # por hacer: implementar submuestreo pixel a pixel
160     width = img.shape[0]
161     heighth = img.shape[1]
162     subsample = np.zeros((int(np.ceil(width/2)), int(np.ceil(heighth/2))), np.float32)
163     print(int(np.ceil(width/2)), int(np.ceil(heighth/2)))
164     for i in range(int(np.ceil(width/2))):
165         for j in range(int(np.ceil(heighth/2))):
166             subsample[i,j] = img[2*i][2*j]
```

```
167     return subsample
168
169 originalBGR = cv2.imread('origami_2022.jpg') #Leer imagen
170
171 if originalBGR is None:
172     assert False, 'Imagen no encontrada'
173
174 if len(originalBGR.shape) == 3:
175     original = cv2.cvtColor(originalBGR, cv2.COLOR_BGR2GRAY)
176 else:
177     original = originalBGR
178
179 input = np.float32( original )
180 print(input.shape,input)
181
182 """# Pausa - Prueba subsample"""
183
184 import cv2
185 from google.colab.patches import cv2_imshow
186
187 originalBGR = cv2.imread('origami_2022.jpg') #Leer imagen
188
189 if originalBGR is None:
190     assert False, 'Imagen no encontrada'
191
192 if len(originalBGR.shape) == 3:
193     original = cv2.cvtColor(originalBGR, cv2.COLOR_BGR2GRAY)
194 else:
195     original = originalBGR
196
197 input = np.float32( original )
198 print(input.shape)
199 cv2_imshow(input)
200
201 print(input.shape)
202 a_img = do_subsample(do_subsample(do_subsample(do_subsample(input))))
203 cv2_imshow(a_img)
204
205 """# Continuando"""
206
207 def calc_gauss_pyramid(input, levels):
208     gausspyr = []
209     current = np.copy(input)
210     gausspyr.append(current)
211     for i in range(1,levels):
212         # Por hacer:
213
214         # 1) Aplicar apply_blur() a la imagen gausspyr[i-1], con sigma 2.0 y ancho 7
215         filtrada = apply_blur(gausspyr[i-1], 2.0 , 7)
216
```

```
217 # 2) Submuestrear la imagen resultante usando do_subsample y guardando el resultado en
218 #    ↪ current
219 current = do_subsample(filtrada)
220
220 gausspyr.append(current)
221 return gausspyr
222
223 import cv2
224 from google.colab.patches import cv2_imshow
225
226 def show_gauss_pyramid(pyramid):
227     # Por hacer: mostrar las imagenes de la piramide de gauss
228     # Se recomienda usar cv2_imshow( ) para mostrar las imagenes
229
230     for img in pyramid:
231         cv2_imshow(img)
232
233 def subtract(input1, input2):
234     assert input1.shape == input2.shape, print("Problema con tamaños",input1.shape,input2.shape)
235     # Por hacer: calcular la resta entre input1 e input2, pixel a pixel
236     width = input1.shape[0]
237     heighth = input1.shape[1]
238     output = np.zeros((width, heighth), np.float32)
239     for i in range(width):
240         for j in range(heighth):
241             output[i][j] = np.float32(input1[i][j]-input2[i][j])
242     return output
243
244 def add(input1, input2):
245     assert input1.shape == input2.shape, print("Problema con tamaños",input1.shape,input2.shape)
246
247     # Por hacer: calcular la suma(?) entre input1 e input2, pixel a pixel
248     width = input1.shape[0]
249     heighth = input1.shape[1]
250     output = np.zeros((width, heighth), np.float32)
251     for i in range(width):
252         for j in range(heighth):
253             output[i][j] = np.float32(input1[i][j]+input2[i][j])
254     return output
255
256 def calc_laplace_pyramid(input, levels):
257     gausspyr = []
258     laplacepyr = []
259     current = np.copy(input)
260     gausspyr.append(current)
261     for i in range(1, levels):
262         # Por hacer:
263         # 1) Aplicar apply_blur( ) a la imagen gausspyr[i-1], con sigma 2.0 y ancho 7
264         filtrada = apply_blur(gausspyr[i-1],2.0,7)
265         # 2) Guardar en laplacepyr el resultado de restar gausspyr[i-1] y la imagen calculada en (1)
```

```
266 laplacepyr.append(subtract(gausspyr[i-1],filtrada)) # Esta linea se debe reemplazar por lo
   ↪ indicado en (2)
267 # 3) Submuestrear la imagen calculada en (1), guardar el resultado en current
268 current = do_subsample(filtrada)
269 gausspyr.append(current)
270 laplacepyr.append(current) # Se agrega el ultimo piso de la piramide de Laplace
271 return laplacepyr
272
273 def abs_fact(input,factor):
274     # Se debe calcular el valor absoluto de los pixeles, y luego multiplicarlos por un factor
275     width = input.shape[0]
276     heighth = input.shape[1]
277     output = np.zeros((width, heighth), np.float32)
278     for i in range(width):
279         for j in range(heighth):
280             output[i][j] = np.abs(input[i][j])*factor
281     return output
282
283 def show_laplace_pyramid(pyramid):
284     # Por hacer: mostrar las imagenes de la piramide de laplace:
285     # Se debe calcular el valor absoluto de los pixeles, y luego multiplicarlos por un factor
286     # Sin embargo, la ultima imagen del ultimo piso se muestra tal cual
287     # Se recomienda usar cv2_imshow( ) para mostrar las imagenes
288
289 for i in range(len(pyramid)-1):
290     nimg = abs_fact(pyramid[i],factor=5)
291     cv2_imshow(nimg)
292     cv2_imshow(pyramid[-1])
293
294 def do_upsample(img):
295     # Por hacer: implementar duplicacion del tamaño de imagen pixel a pixel
296     # Un pixel de la imagen de salida debe ser el promedio de los 4 pixeles mas cercanos de la imagen
       ↪ de entrada
297     # Se debe tener cuidado de que los indices no salgan fuera del tamano de la imagen
298     width = img.shape[0]
299     heighth = img.shape[1]
300     upsample = np.zeros((width*2, heighth*2), np.float32)
301     for i in range(width-1):
302         for j in range(heighth-1):
303             val = np.float32((img[i][j]+img[i+1][j]+img[i][j+1]+img[i+1][j+1])/4)
304
305             upsample[2*i,2*j] = val
306             upsample[2*i+1,2*j] = val
307             upsample[2*i,2*j+1] = val
308             upsample[2*i+1,2*j+1] = val
309
310     return upsample
311
312 print(input.shape)
313 print(do_upsample(input).shape)
314 #cv2_imshow(do_upsample(input))
```

```
315
316 def resize(input,x,y):
317     if x==input.shape[0] and y==input.shape[1]:
318         return input
319     output = np.ones((x,y),np.float32)
320     #print(input.shape,x,y)
321     for i in range(x):
322         for j in range(y):
323             output[i][j] = input[i][j]
324     return output
325
326 def do_reconstruct(laplacepyr):
327     output = np.copy( laplacepyr[len(laplacepyr)-1] )
328     for i in range(1, len(laplacepyr)):
329         level = int(len(laplacepyr)) - i - 1
330         # Por hacer: repetir estos dos pasos:
331         # (1) Duplicar tamano output usando do_upsample( )
332         upsample = do_upsample(output)
333         #tengo problemas con las dimensiones por lo que voy a agregar un reshape
334         #print(upsample.shape)
335         upsample = resize(upsample,laplacepyr[level].shape[0],laplacepyr[level].shape[1])
336         #print(upsample.shape)
337         # (2) Sumar resultado de (1) y laplacepyr[level] usando add( ), almacenar en output
338         output = add(upsample, laplacepyr[level])
339     return output
340
341 ls
342
343 names = ["origami","corteza","dali","techo_falso"]
344
345 for nombre in names:
346     print(nombre+'_2022.jpg')
347     originalBGR = cv2.imread(nombre+'_2022.jpg') #Leer imagen
348
349 if originalBGR is None:
350     assert False, 'Imagen no encontrada'
351
352 if len(originalBGR.shape) == 3:
353     original = cv2.cvtColor(originalBGR, cv2.COLOR_BGR2GRAY)
354 else:
355     original = originalBGR
356
357 input = np.float32( original )
358
359 print('Piramide de gauss:')
360 gausspyramid = calc_gauss_pyramid(input, 5)
361 show_gauss_pyramid(gausspyramid)
362
363 print('Piramide de laplace:')
364 laplacepyramid = calc_laplace_pyramid(input, 5)
365 show_laplace_pyramid(laplacepyramid)
```

```
366
367     print('reconstruida:')
368     reconstr = do_reconstruct(laplacepyramid)
369     cv2_imshow(reconstr)
370
371 #Prueba
372 from mpl_toolkits.mplot3d import Axes3D
373 from matplotlib import cm
374 import matplotlib.pyplot as plt
375
376 sigma = 1
377 width = 9
378
379 fig = plt.figure()
380 ax = fig.add_subplot(111, projection='3d')
381 x = y = np.arange(0, width, 0.1)
382 X, Y = np.meshgrid(x, y)
383
384 def dgaussian(row,col,centro,sigma):
385     return -((row-centro)+(col-centro))*np.exp(-((row-centro)**2+(col-centro)**2)/(2*sigma**2))/
386             np.sqrt(2*np.pi)*sigma**3
387
388 Z = dgaussian(X,Y,width//2,sigma).reshape(X.shape)
389
390 ax.plot_surface(X, Y, Z,cmap=cm.coolwarm)
391 plt.title("Derivative of Gaussian 2D")
392 plt.show()
393
394 #Prueba
395 from mpl_toolkits.mplot3d import Axes3D
396 from matplotlib import cm
397 import matplotlib.pyplot as plt
398
399 sigma = 1
400 width = 9
401
402 fig = plt.figure()
403 ax = fig.add_subplot(111, projection='3d')
404 x = y = np.arange(0, width, 0.1)
405 X, Y = np.meshgrid(x, y)
406
407 def d2gaussian(row,col,centro,sigma):
408     return (-sigma**2+((row-centro)**2+(col-centro)**2))*np.exp(-((row-centro)**2+(col-centro)
409             **2)/(2*sigma**2))/(np.sqrt(2*np.pi)*sigma**5)
410
411 Z = d2gaussian(X,Y,width//2,sigma).reshape(X.shape)
412
413 ax.plot_surface(X, Y, Z,cmap=cm.coolwarm)
414 plt.title("Laplacian of Gaussian 2D")
415 plt.show()
```

```

415 def compute_dgauss_mask_2d(sigma, width,edge):
416     gmask = np.zeros((width, width), np.float32)
417     # Por hacer: implementar calculo de mascara gaussiana 2d pixel a pixel
418     centro = width//2
419     for row in range(width):
420         for col in range(width):
421             if edge=="x":
422                 val = row
423             else:
424                 val = col
425             gmask[row][col] = (((val-centro))/(2*np.pi*sigma**4))*np.exp(-((row-centro)**2+(col-centro
426             ↵ )**2)/(2*sigma**2) )
427     return gmask
428
429 def compute_d2gauss_mask_2d(sigma, width,edge):
430     gmask = np.zeros((width, width), np.float32)
431     # Por hacer: implementar calculo de mascara gaussiana 2d pixel a pixel
432     centro = width//2
433     for row in range(width):
434         for col in range(width):
435             if edge=="x":
436                 val = row
437             else:
438                 val = col
439             gmask[row][col] = (-1 + ((val-centro)**2/sigma**2)*np.exp(-((row-centro)**2+(col-centro)
440             ↵ **2)/(2*sigma**2) )/(2*np.pi*sigma**4)
441     return gmask
442
443 def pasa_alto(img, tipo_filtro, sigma, width, factor=5):
444     output = np.zeros((img.shape[0], img.shape[1]), np.float32)
445     if tipo_filtro=="1d":
446         dgmask_x = compute_dgauss_mask_2d(sigma, width, "x")
447         dgmask_y = compute_dgauss_mask_2d(sigma, width, "y")
448     else:
449         dgmask_x = compute_d2gauss_mask_2d(sigma, width, "x")
450         dgmask_y = compute_d2gauss_mask_2d(sigma, width, "y")
451     output_x = convolution_cython(input, dgmask_x)
452     output_y = convolution_cython(input, dgmask_y)
453     for x in range(output.shape[0]):
454         for y in range(output.shape[1]):
455             #se le agrega un factor solo para mostrar de mejor manera los resultados
456             output[x][y] = np.sqrt(output_x[x][y]**2+output_y[x][y]**2)*factor
457     return output
458
459 for nombre in names:
460     print(nombre+'_2022.jpg')
461     originalBGR = cv2.imread(nombre+'_2022.jpg') #Leer imagen
462
463     if originalBGR is None:
464         assert False, 'Imagen no encontrada'

```

```
464     if len(originalBGR.shape) == 3:
465         original = cv2.cvtColor(originalBGR, cv2.COLOR_BGR2GRAY)
466     else:
467         original = originalBGR
468
469     input = np.float32( original )
470     cv2_imshow(pasa_alto(input,"1d",1,15))
471     cv2_imshow(pasa_alto(input,"2d",1,15,10))
472
473 for nombre in names:
474     print(nombre+'_2022.jpg')
475     originalBGR = cv2.imread(nombre+'_2022.jpg') #Leer imagen
476
477 if originalBGR is None:
478     assert False, 'Imagen no encontrada'
479
480 if len(originalBGR.shape) == 3:
481     original = cv2.cvtColor(originalBGR, cv2.COLOR_BGR2GRAY)
482 else:
483     original = originalBGR
484
485 input = np.float32( original )
486
487 print('Piramide de gauss:')
488 gausspyramid = calc_gauss_pyramid(input, 5)
489 S = [1,1,1,1,1]
490 W = [15,12,9,7,5,3]
491 for img,width,s in zip(gausspyramid,W,S):
492     cv2_imshow(pasa_alto(img,"1d",s,width))
493 for img,width in zip(gausspyramid,W):
494     cv2_imshow(pasa_alto(img,"2d",s,width+5,7))
```

Código 3: Código completo

## Referencias

- [1] Tim Hauke Heibel. 1D and 2D Gaussian Derivatives. Technische Universität Münchenr. Disponible en: <https://campar.in.tum.de/Chair/HaukeHeibelGaussianDerivatives>