

VISIÓN POR COMPUTADOR
GRADO EN INGENIERÍA INFORMÁTICA

PRÁCTICA 1: CONVOLUCIÓN Y DERIVADAS

Realizado por:
Alba Casillas Rodríguez

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE
TELECOMUNICACIÓN

CURSO 2021-2022



Índice

1. Ejercicio 1	2
1.1. Apartado A	2
1.2. Apartado B	8
1.3. Apartado C	10
1.4. Apartado D	18
2. Ejercicio 2	23
2.1. Apartado A	23
2.2. Apartado B	26
2.3. Apartado C	28
3. BONUS - Imágenes Híbridas	30
3.1. Apartado 1	30
3.2. Apartado 2	38
3.3. Apartado 3	40
Referencias	43

1. Ejercicio 1

1.1. Apartado A

Considere la función Gaussiana 1D de media 0 y desviación típica . Calcular las máscaras discretas 1D de la función Gaussiana, la derivada de la Gaussiana y la segunda derivada de la Gaussiana. La función implementada debe considerar que tanto el valor de σ como el tamaño de la máscara son posibles entradas alternativas a la función.

Para la realización de este ejercicio, debemos considerar como valores de entrada un tamaño de máscara T , o un valor de σ . Es por ello que:

- Dado un σ , se deberá calcular el tamaño de máscara, T .
- Dado un tamaño de máscara T , se deberá ajustar el σ

los cuales podremos despejar atendiendo a la fórmula: $\mathbf{T} = 2\mathbf{K} + 1$, donde $\mathbf{K} = 3\sigma$, valor establecido por la literatura, ya que se sabe que casi todos los valores de la discretización de la Gaussiana se encuentran dentro de tres desviaciones estándar de la media. Una vez hayados los valores necesarios, utilizamos el parámetro k para calcular el rango en el que discretizaremos la máscara.

Se debe realizar el cálculo de las máscaras discretas 1D de:

- La función Gaussiana:

$$G(x) = c \exp\left(\frac{-x^2}{2\sigma^2}\right)$$

donde, en esta práctica, se ignora la constante c .

- La primera derivada de la Gaussiana:

$$\frac{\partial G}{\partial x} = -\frac{x}{\sigma^2} G(x) = -\frac{x \exp\left(\frac{-x^2}{2\sigma^2}\right)}{\sigma^2}$$

- La segunda derivada de la Gaussiana:

$$\frac{\partial^2 G(x)}{\partial x^2} = -\left[\frac{1}{\sigma^2} - \frac{x^2}{\sigma^4}\right] G(x) = -\left[\frac{e^{\frac{-x^2}{2\sigma^2}}}{\sigma^2} - \frac{x^2 e^{\frac{-x^2}{2\sigma^2}}}{\sigma^4}\right] = -\frac{(\sigma^2 - x^2) e^{\frac{-x^2}{2\sigma^2}}}{\sigma^4}$$

con la consideración de que para la función Gaussiana, deberemos dividir la máscara por la suma de sus valores; mientras que, en el caso de sus derivadas, no se realizará este paso ya que la suma de los valores de las máscaras se aproximan al cero; y los valores del kernel aumentarían considerablemente.

Represente en ejes cartesianos las máscaras obtenidas como funciones 1D y compare sus formas con las máscaras dadas por la función de OpenCV getDerivKernels para los mismos tamaños de máscara. Use los tamaños 5, 7 y 9.

Para comprobar que las máscaras se calculan correctamente, se usarán los tamaños de máscara **5, 7 y 9**, aunque en ejercicios posteriores se corroborará el funcionamiento mediante el paso de un valor sigma. Nótese que dichos valores son *impares*, ya que el uso de una máscara par dificulta centrar la máscara en un punto concreto de la imagen, mientras que las máscaras de tamaño impar son simétricas alrededor del origen.

Las máscaras obtenidas son:

```
1 TAMANIO DE MASCARA: 5
2
3 Mascara obtenida de la funcion gaussiana:
4 [0.00664603 0.19422555 0.59825683 0.19422555 0.00664603]
5
6 Mascara obtenida de la funcion PRIMERA DERIVADA gaussiana:
7 [0.04999048442209038, 0.7304680515562869, -0.0, -0.7304680515562869,
8 -0.04999048442209038]
9
10 Mascara obtenida de la funcion SEGUNDA DERIVADA gaussiana:
11 [0.1999619376883615, 0.9130850644453586, -2.25, 0.9130850644453586,
     0.1999619376883615]
```

```
1 TAMANIO DE MASCARA: 7
2
3 Mascara obtenida de la funcion gaussiana:
4 [0.00443305 0.05400558 0.24203623 0.39905028 0.24203623 0.05400558
5 0.00443305]
6
7 Mascara obtenida de la funcion PRIMERA DERIVADA gaussiana:
8 [0.033326989614726917, 0.2706705664732254, 0.6065306597126334, -0.0,
9 -0.6065306597126334, -0.2706705664732254, -0.033326989614726917]
10
11 Mascara obtenida de la funcion SEGUNDA DERIVADA gaussiana:
12 [0.08887197230593845, 0.4060058497098381, -0.0, -1.0, -0.0,
     0.4060058497098381, 0.08887197230593845]
```

```

1 TAMANIO DE MASCARA: 9
2
3 Mascara obtenida de la funcion gaussiana:
4 [0.00332573 0.02381792 0.09719199 0.22597815 0.29937241 0.22597815
5   0.09719199 0.02381792 0.00332573]
6
7 Mascara obtenida de la funcion PRIMERA DERIVADA gaussiana:
8 [0.02499524221104519, 0.13425667096200922, 0.36523402577814346,
9   0.42459727611881665, -0.0, -0.42459727611881665,
10  -0.36523402577814346, -0.13425667096200922, -0.02499524221104519]
11
12 Mascara obtenida de la funcion SEGUNDA DERIVADA gaussiana:
13 [0.04999048442209038, 0.1818059085943875, 0.22827126611133966,
14   -0.18576130830198226, -0.5625, -0.18576130830198226,
15   0.22827126611133966, 0.1818059085943875, 0.04999048442209038]

```

Del mismo modo, para dichos tamaños de máscara se utilizará la función de **OpenCV**, *getDerivKernels*:

cv2.getDerivKernels(dx,dy,ksize)

donde *dx* es el orden de la derivada respecto de *x*, *dy* el orden de la derivada respecto de *y* y *ksize* es el tamaño del kernel. Para apreciar mejor el resultado de la función a la hora de mostrarlas, se ha activado el parámetro *normalize*, práctica recomendada por la documentación oficial [1] a la hora de trabajar con imágenes de valores flotantes.

Nos quedamos con el primer parámetro que devuelve la función, la máscara de la función (Gaussiana o sus derivadas) con respecto a *x*:

```

1 TAMANIO DE MASCARA: 5
2
3 Mascara obtenida de la funcion gaussiana por OpenCV:
4 [[1.]
5  [4.]
6  [6.]
7  [4.]
8  [1.]]
9
10 Mascara obtenida de la funcion PRIMERA DERIVADA gaussiana por OpenCV:
11 [[-1.]
12  [-2.]
13  [ 0.]
14  [ 2.]
15  [ 1.]]
16
17
18

```

```
19| Mascara obtenida de la funcion SEGUNDA DERIVADA gaussiana por OpenCV:  
20| [[ 1.]  
21| [ 0.]  
22| [-2.]  
23| [ 0.]  
24| [ 1.]]
```

```
1| TAMANIO DE MASCARA: 7  
2|  
3| Mascara obtenida de la funcion gaussiana por OpenCV:  
4| [[ 1.]  
5| [ 6.]  
6| [15.]  
7| [20.]  
8| [15.]  
9| [ 6.]  
10| [ 1.]]  
11|  
12| Mascara obtenida de la funcion PRIMERA DERIVADA gaussiana por OpenCV:  
13| [[ -1.]  
14| [ -4.]  
15| [ -5.]  
16| [ 0.]  
17| [ 5.]  
18| [ 4.]  
19| [ 1.]]  
20|  
21| Mascara obtenida de la funcion SEGUNDA DERIVADA gaussiana por OpenCV:  
22| [[ 1.]  
23| [ 2.]  
24| [-1.]  
25| [-4.]  
26| [-1.]  
27| [ 2.]  
28| [ 1.]]
```

```
1| TAMANIO DE MASCARA: 9  
2|  
3| Mascara obtenida de la funcion gaussiana por OpenCV:  
4| [[ 1.]  
5| [ 8.]  
6| [28.]  
7| [56.]  
8| [70.]  
9| [56.]  
10| [28.]  
11| [ 8.]  
12| [ 1.]]  
13|
```

```

14| Mascara obtenida de la funcion PRIMERA DERIVADA gaussiana por OpenCV:
15| [[ -1.]
16| [ -6.]
17| [ -14.]
18| [ -14.]
19| [  0.]
20| [ 14.]
21| [ 14.]
22| [  6.]
23| [  1.]]
24|
25| Mascara obtenida de la funcion SEGUNDA DERIVADA gaussiana por OpenCV:
26| [[  1.]
27| [  4.]
28| [  4.]
29| [ -4.]
30| [ -10.]
31| [ -4.]
32| [  4.]
33| [  4.]
34| [  1.]]

```

Graficamos los resultados:

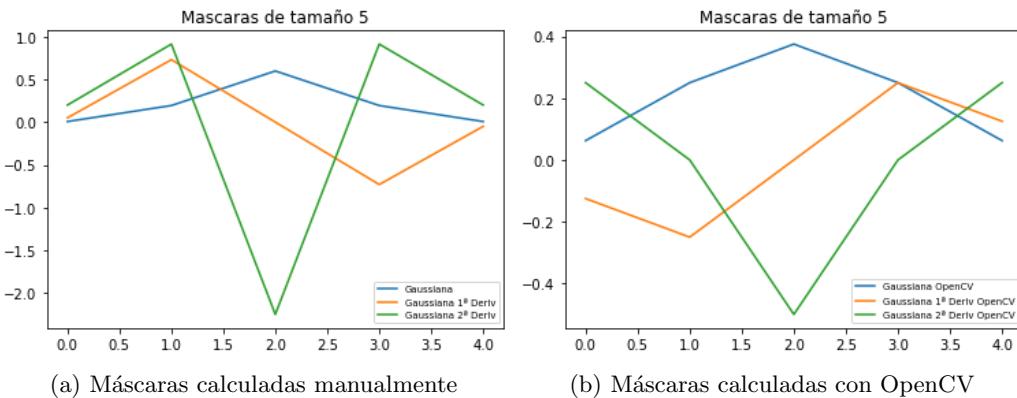


Figura 1: Comparación de máscaras de tamaño 5

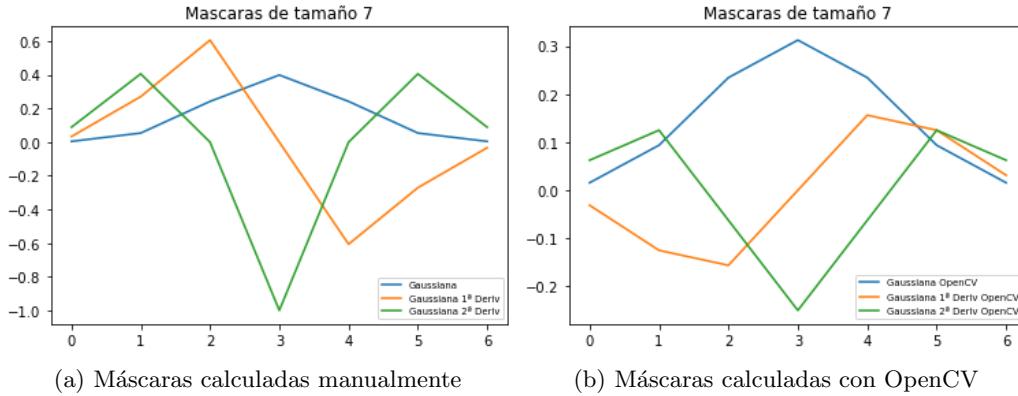


Figura 2: Comparación de máscaras de tamaño 7

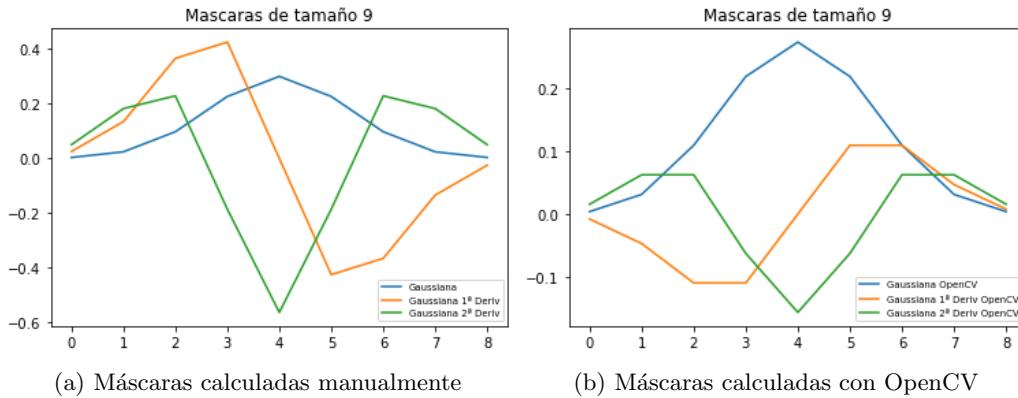


Figura 3: Comparación de máscaras de tamaño 9

Tras observar las gráficas, podemos encontrar similitudes entre ambos métodos: en esencia, las máscaras de la función Gaussiana y su derivada segunda parecen ser las mismas, excepto por el hecho de que se encuentran en distintas escalas. Esto se debe a que *getDerivKernels* realiza un cálculo aproximado de las funciones mediante los operadores de *Sobel* y *Scharr* [1], mientras que nosotros manualmente realizamos un cálculo exacto.

Por otro lado, capta la atención la máscara de la primera derivada, la cual se encuentra invertida respecto al eje X, aunque no sea una diferencia relevante puesto que se trata de una máscara simétrica.

1.2. Apartado B

Calcule las máscaras discretas 1D de longitud 5 y 7 tanto de alisamiento como de derivada de primer orden generadas a partir de la aproximación binomial de la Gaussiana y la máscara de derivada de longitud 3

La cuestión de este ejercicio es saber que, a partir del **Kernel binomial de alisamiento de tamaño 3 (1,2,1)** y el **Kernel binomial de primera derivada de tamaño 3 (-1,0,1)**, vistos en teoría, podemos obtener los kernels de alisamiento y primera derivada de los tamaños de máscara siguientes.

Por ello, para obtener, por ejemplo, el kernel binomial de alisamiento de tamaño 5, deberemos convolucionar el kernel binomial de alisamiento de tamaño 3 consigo mismo; para obtener el kernel binomial de la primera derivada de tamaño 5, se deberá convolucionar el kernel binomial de alisamiento de tamaño 3, con el kernel binomial de primera derivada de tamaño 3.

En esencia, al convolucionar un kernel binomial con el kernel binomial de alisamiento o derivada de tamaño 3, dará como resultado el kernel de siguiente tamaño (impar), es decir, usando la máscara 5 se obtendrá el de máscara 7, usando la máscara 7 el de máscara 9, etc.

Para las convoluciones se hará uso de la función *np.convolve*, ya que en este punto de la práctica aún se permite su uso.

```
1 Calculamos los tamanios de mascara de alisamiento :  
2  
3 Convolucionamos un Kernel binomial de alisamiento de tam 3 y consigo  
    mismo  
4 [1 4 6 4 1]  
5  
6 Convolucionamos un Kernel binomial de alisamiento de tam 5 y el kernel  
    binomial de tam 3  
7 [ 1 6 15 20 15 6 1]  
8  
9 Convolucionamos un Kernel binomial de alisamiento de tam 7 y el kernel  
    binomial de tam 3  
10 [ 1 8 28 56 70 56 28 8 1]
```

```

1 Calculamos los tamanios de mascara de primera derivada:
2
3 Calculamos las mascaras de un Kernel binomial de derivada de tam 3 y
   la derivada de tam 3
4 [-1 -2  0  2  1]
5
6 Calculamos las mascaras de un Kernel binomial de derivada de tam 5 y
   la derivada de tam 3
7 [-1 -4 -5  0  5  4  1]
8
9 Calculamos las mascaras de un Kernel binomial de derivada de tam 7 y
   la derivada de tam 3
10 [-1 -6 -14 -14  0  14  14   6   1]

```

Ejecute la función `cv2.getDerivKernels(0,1,9)`. Observe los vectores de salida y compárelos con los previamente calculados, ¿Qué relación hay? ¿Qué conclusiones extrae sobre la aproximación de OpenCV al cálculo de las máscaras de derivadas de primer orden?

El resultado de ejecutar el método es el siguiente:

```

1
2 Ejecutamos getDerivKernels (0 ,1 ,9)
3 [[ [ 1. -1.]
4   [ 8. -6.]
5   [ 28. -14.]
6   [ 56. -14.]
7   [ 70.  0.]
8   [ 56.  14.]
9   [ 28.  14.]
10  [ 8.  6.]
11  [ 1.  1.] ]]

```

Como se puede observar, con el método `getDerivKernels(0,1,9)` se calcula la primera derivada con respecto al eje Y para un tamaño de máscara 9. Vemos que como resultado, se obtiene en la primera columna el kernel binomial de alisamiento de longitud 9 y en la segunda columna el kernel binomial de primera derivada de tamaño 9; coincidiendo con los valores obtenidos al realizar la convolución del kernel binomial de alisamiento de tamaño 7 y el kernel binomial de primera derivada de longitud 3 (para obtener el kernel binomial de alisamiento de longitud 9) y la convolución del kernel binomial de primera derivada de tamaño 7 y el kernel binomial de primera derivada de tamaño 3 (para obtener el kernel binomial de primera derivada de tamaño 9).

De esta manera se puede afirmar que la función `getDerivKernels` realiza sus cálculos mediante el uso de kernels binomiales.

1.3. Apartado C

Implementar la convolución de una imagen con una máscara 2D de dimensiones inferior a la imagen suponiendo la propiedad de separabilidad de la máscara e imponiendo bordes reflejados. La entrada a la función serán las máscaras 1D descomposición de la máscara 2D.

Para la realización de todos los apartados de este ejercicio, se ha decidido probar con valores de sigma de **1, 3 y 5** con el objetivo de visualizar qué diferencias se obtienen al aumentar el sigma sobre una misma imagen.

A partir de sigma, se han reutilizado las funciones para el cálculo de las máscaras del ejercicio 1A; de manera que a la hora de realizar la convolución 2D sobre una imagen, utilizaremos un kernel para aplicar de forma horizontal y otro para aplicar de forma vertical; los cuales podrán ser diferentes, aunque para este apartado se ha decidido hacer uso de la misma máscara.

El hecho de que se pueda realizar una convolución 2D a partir de las máscaras 1D es debido a que se trabaja con *Kernels separables*, de manera que aplicar primero el Kernel en el eje X y sobre su resultado, el Kernel en el eje Y resulta más eficiente al tener un menor número de operaciones en comparación a usar directamente un Kernel 2D sobre la imagen.

Dado los argumentos de entrada, desarrollamos la función:

convolucion2D(imagen, hmask, vmask)

que tendrá el siguiente funcionamiento:

1. Aplicamos un ***padding*** a la imagen para que pueda aplicarse la máscara en los píxeles de la imagen cercanos a los bordes. Dicho padding dependerá de la longitud de la máscara que a utilizar, y se añadirá en el exterior de la imagen. El tamaño del borde, por tanto, será: $\text{int}((\text{len}(hmask) - 1)/2)$ y lo calcularemos mediante el uso del método *copyMakeBorder*, visto en la práctica anterior.

Los bordes utilizados serán *bordes reflejados*, por lo que los píxeles de fuera de los bordes son un reflejo de los de dentro.

2. Crearemos una matriz auxiliar rellena de ceros en la que almacenaremos los resultados de la primera convolución.
3. Se recorre una vez las filas con el ancho de la imagen original para realizar la ***primera convolución***. El resultado será guardado en la matriz creada en el paso anterior, teniendo en cuenta se deberá "saltar" el padding establecido a la hora de guardar los resultados de la operación. Para la multiplicación

del kernel con la matriz, se ha hecho uso de la función *matmul* [2], la cual devuelve la matriz producto de los parámetros de entrada:

```
matrix_tmp[inicio+padding] = np.matmul(hmask,  
    imagen_padding[inicio:(inicio+len(hmask)),:])
```

4. Para aportar simplicidad y optimización al algoritmo, calcularemos la transpuesta de la matriz resultante de la primera convolución, de manera que se podrá realizar la *segunda convolución* recorriendo de nuevo las filas una única vez.
5. Realizamos el mismo proceso, recorriendo esta vez la altura de la matriz resultante de la primera convolución e ignorando su padding. Esto se debe a que la matriz de imagen de entrada no tiene por qué ser necesariamente cuadrada, por lo que usar la altura de la imagen original para recorrer las filas sería incorrecto.
6. Una vez realizadas ambas convoluciones, eliminamos el padding para mantener las dimensiones originales de la imagen. Para ello, simplemente se debe extraer la submatriz que no contiene bordes.
7. Normalizamos la imagen convolucionada haciendo uso de la función implementada en la práctica anterior. De esta manera no se perderá información y además podremos visualizarla correctamente.

Una vez explicada la lógica del procedimiento, visualizamos los resultados obtenidos:

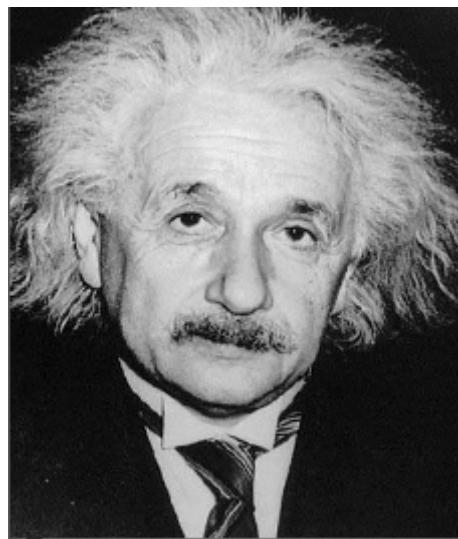


Figura 4: Imagen original utilizada

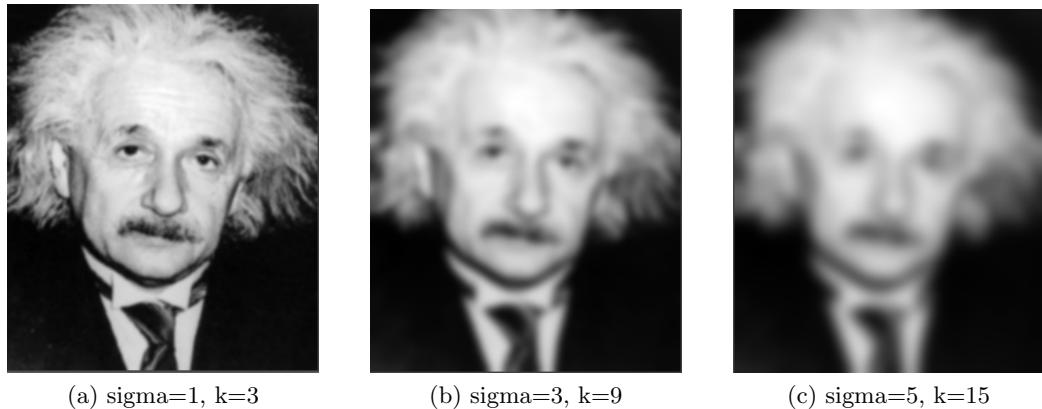
(a) $\sigma=1, k=3$ (b) $\sigma=3, k=9$ (c) $\sigma=5, k=15$

Figura 5: Comparación de convoluciones 2D mediante sigma

Como se puede apreciar en la *Figura 5* que, como era de esperar, se produce un efecto de emborramiento o desenfoque de la imagen.

El valor de **sigma** expresa la desviación típica de la función gaussiana con la cual se construye la máscara. Gracias a sigma y a las funciones anteriormente definidas podemos despejar el valor de **k**, el cual se puede observar que crece a medida que se aumenta el valor de sigma.

Este natural **k** expresa el número de píxeles que tendría cada lado de la máscara. Se destaca que debe ser impar para que la máscara tenga un centro que corresponda al píxel de la imagen original cuyo valor se va a modificar.

A medida que el sigma proporcionado aumenta, el efecto de emborronamiento se hace más notable. El aumento de **k** (es decir, una máscara más grande) hace que se tenga en cuenta un mayor número de píxeles al convolucionar; mientras que el aumento de sigma permite una mayor varianza alrededor del centro y, por tanto, un aumento de los valores más laterales, de manera que se utilizará *información cada vez menos local* con respecto al píxel actual.

Teniendo en cuenta la explicación anterior, un valor de sigma igual a 1 es recomendable, ya que es suficiente para apreciar el desenfoque producto de la convolución al eliminar frecuencias altas, sin obligar a usar un kernel más grande que añada tiempo computacional a la función.

Comparar su funcionamiento con:

La salida de cv2.GaussianBlur para una máscara de entrada Gaussiana con iguales parámetros en ambos casos. Mostrar ambos resultados.

Utilizamos la función *GaussianBlur*:

cv2.GaussianBlur(src, ksize, sigmaX, sigmaY)

donde:

- **src** es la imagen a convolucionar.
 - **ksize** es el tamaño del kernel, el cual se escribirá en forma de tupla ya que puede diferir en ancho y alto, aunque nosotros utilizaremos el mismo valor para ambas dimensiones. Se debe tener en cuenta que debe ser *positivo* e *impar*.
 - **sigmaX** es el valor de sigma en la dirección X, donde usaremos los mismos valores que en el apartado anterior.
 - **sigmaY** es el valor de sigma en la dirección Y. En nuestro caso, valdrá -1 para que el propio método estime el valor más adecuado.

Mostramos los resultados:

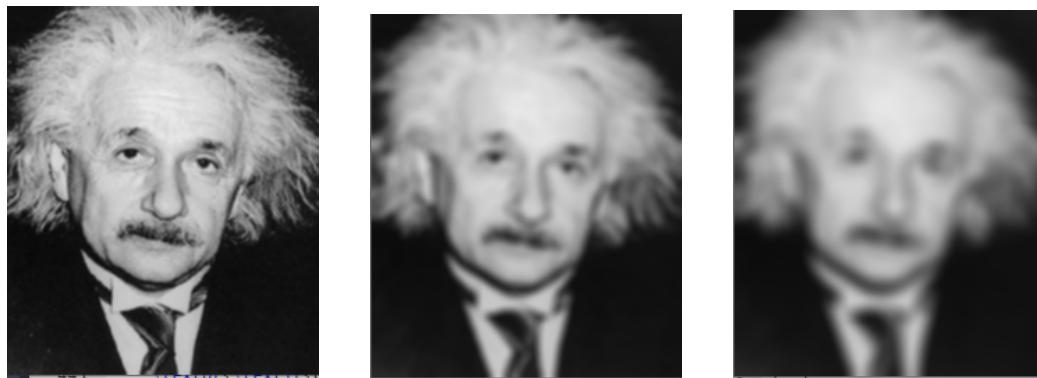
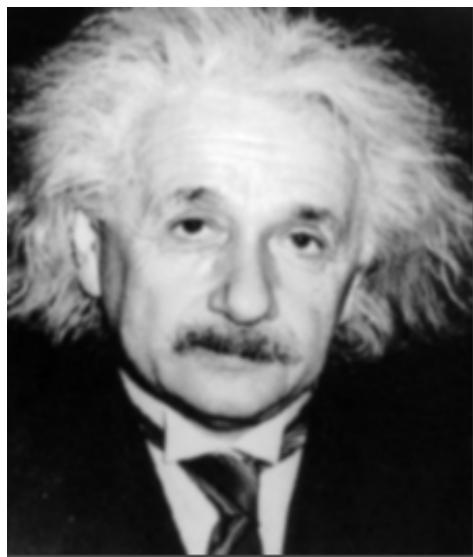
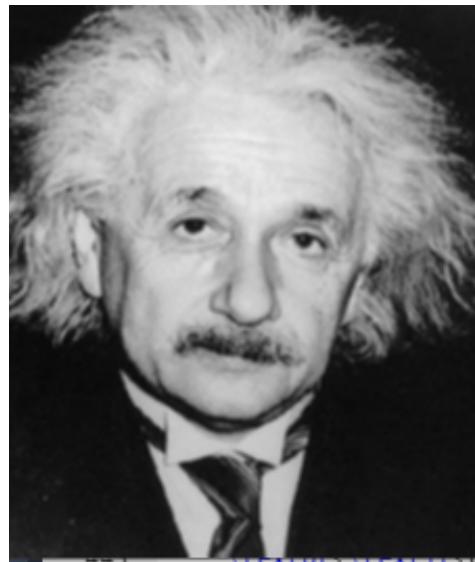


Figura 6: Comparación de GaussianBlur mediante sigma

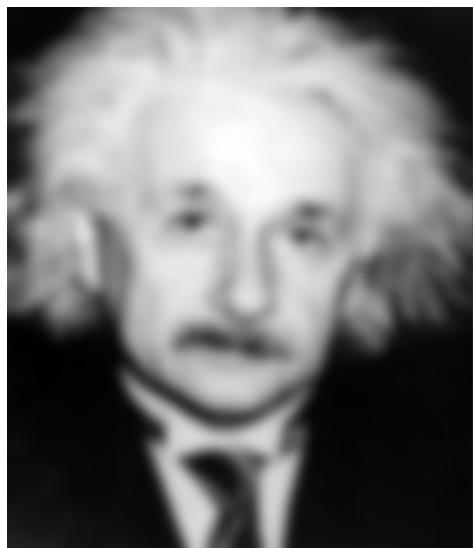


(a) Convolución manual

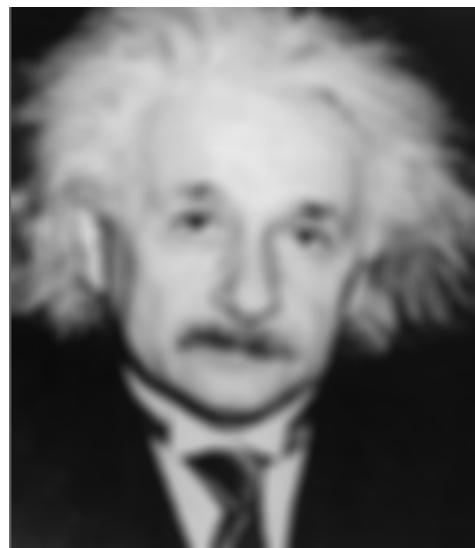


(b) Gaussian Blur

Figura 7: Comparación de Convolución Manual con GaussianBlur para sigma=1



(a) Convolución manual



(b) Gaussian Blur

Figura 8: Comparación de Convolución Manual con GaussianBlur para sigma=3

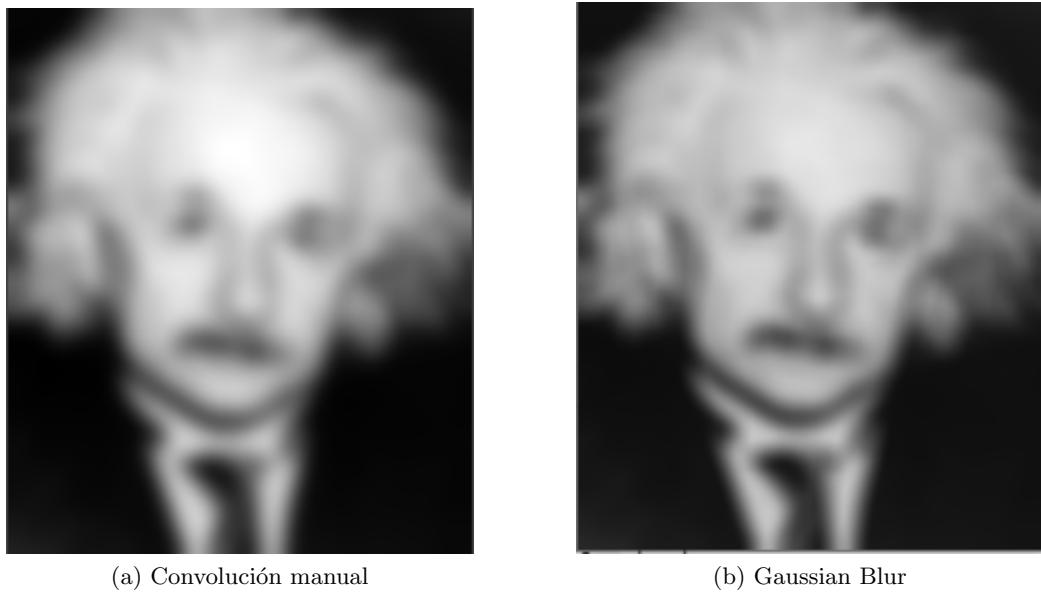


Figura 9: Comparación de Convolución Manual con GaussianBlur para sigma=5

Tras comparar ambos métodos, podemos ver que aunque obtenemos resultados muy similares, no son exactamente iguales, pudiéndose apreciar las diferencias más fácilmente al aumentar sigma. No es de extrañar, puesto que **OpenCV** prima *eficiencia ante precisión*, por lo que nuestro método proporciona un desenfoque más correcto a costa de tomar más tiempo de computación.

Usar las máscaras del punto A para calcular la imágenes derivadas respecto de x e y de una imagen dada. Mostrar los resultados.

La esencia de este apartado reside en que, tras calcular las máscaras *hmask* y *vmask* para la función Gaussiana y la primera derivada de la Gaussiana respectivamente, debemos aplicar las siguientes fórmulas:

$$g_x = I * G_y * G'_x$$

$$g_y = I * G_x * G'_y$$

que se traducen en:

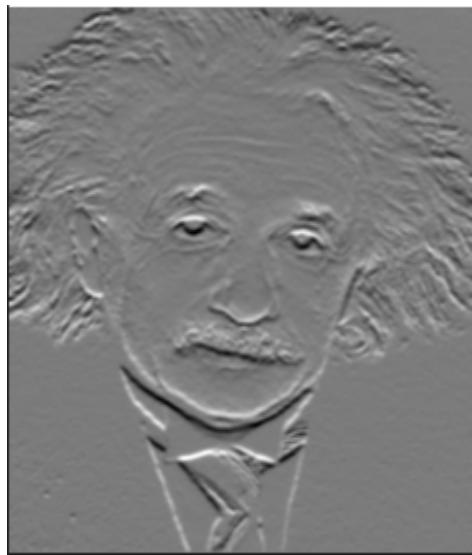
Derivada respecto de X:

convolucion2D(imagen, vmask, hmask)

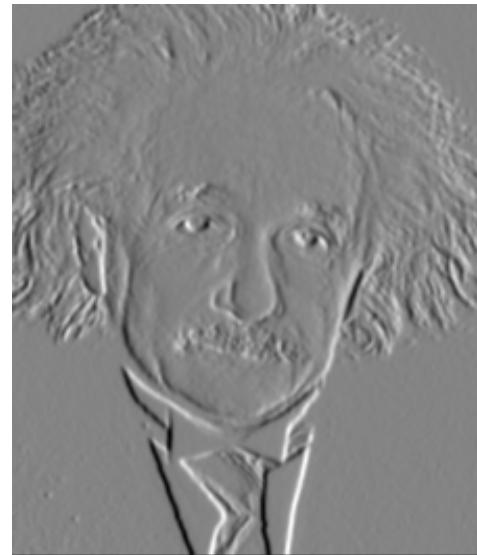
Derivada respecto de Y:

convolucion2D(imagen, hmask, vmask)

Los resultados son los siguientes:



(a) Derivada respecto de X

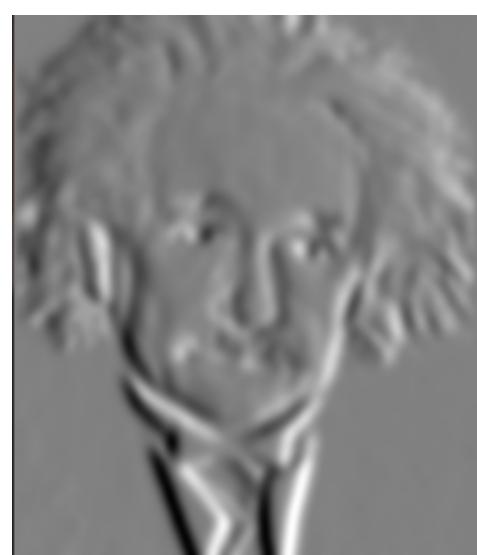


(b) Derivada respecto de Y

Figura 10: Derivadas con $\sigma=1$



(a) Derivada respecto de X



(b) Derivada respecto de Y

Figura 11: Derivadas con $\sigma=3$



(a) Derivada respecto de X



(b) Derivada respecto de Y

Figura 12: Derivadas con sigma=5

Observamos que la derivada en X muestra las transiciones en el eje horizontal, mientras que la derivada en Y muestra las transiciones en el eje vertical. Comprobamos que este tipo de máscaras resaltan los píxeles que transitan entre distintos niveles de grises, es por ello que algunas zonas se resaltan en negro (transición de un valor más alto a uno más bajo) y otras en blanco (transición de un valor más bajo a uno más alto).

1.4. Apartado D

Usar las implementaciones del punto A para calcular máscaras normalizadas de la Laplaciana de una Gaussiana. Mostrar ejemplos de funcionamiento usando bordes reflejados y dos valores de sigma (1 y 3) con alguna imagen. Compare los resultados con la salida dada por OpenCV con la función Laplaciana. Discuta pros y contras.

La **Laplaciana de una Gaussiana (LoG)** [4] se trata de un detector de líneas el cual se puede utilizar para construir un detector de bordes. Se define como el operador de Laplaciana aplicado a un Kernel Gaussiano. El operador de Laplaciana es la suma de las derivadas de segundo orden.

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

Dado que las derivadas de una imagen son difíciles de calcular, a menudo se utiliza un filtro Gaussiano al calcular estas derivadas, de esta manera, además, se aplica un suavizado que nos permitirá no detectar demasiadas variaciones en la imagen. Por tanto, cuando se aplica al operador de Laplaciana, dicho filtro Gaussiano conduce a la Laplaciana de la Gaussiana.

$$\Delta(I * G) = I * \Delta G = I \otimes \frac{\partial^2}{\partial x^2} G + I \otimes \frac{\partial^2}{\partial y^2} G$$

Teniendo en cuenta que la primera derivada de la Gaussiana es separable, partiendo de la Gaussiana podemos separar sus componentes en X e Y, de manera que al final se obtiene un filtro Gaussiano en *horizontal* y otro en *vertical*; obteniendo la derivada en X por el filtro Gaussiano en Y, de manera que se aplica un *suavizado en una dirección* y un *filtro de diferencias* en el otro. Si extendemos esta idea a la segunda derivada, se obtiene una convolución en horizontal con la segunda derivada de la Gaussiana y luego, en vertical, el suavizado de la Gaussiana y un suavizado en horizontal de la Gaussiana con la convolución en vertical de la segunda derivada de la Gaussiana:

$$\frac{\partial^2}{\partial x^2} G + \frac{\partial^2}{\partial y^2} G = G''_h(x) \cdot G_v(y) + G_h(x) \cdot G''_v(y)$$

Nuestro Kernel de la LoG ha pasado de ser una operación de 4 convoluciones 1D. Como último paso, el resultado se deberá multiplicar por σ^2 para aplicar una **normalización de escala**, ya que de no hacerlo, el aumento de sigma generaría una respuesta cada vez más '*llana*' al aplicar la segunda derivada y llegaría un momento en el que no se detectarían cambios.

Una vez explicada la idea fundamental del proceso, pasamos a mostrar los resultados:

Para un sigma = 1:

Calculamos los Kernels 1D utilizados en las convoluciones 1D:

```

1 Mascara Gaussiana
2 [0.00443305  0.05400558  0.24203623  0.39905028  0.24203623  0.05400558
   0.00443305]
3
4 Mascara Segunda Derivada
5 [0.08887197230593845,  0.4060058497098381,  -0.0,  -1.0,  -0.0,
   0.4060058497098381,  0.08887197230593845]
```

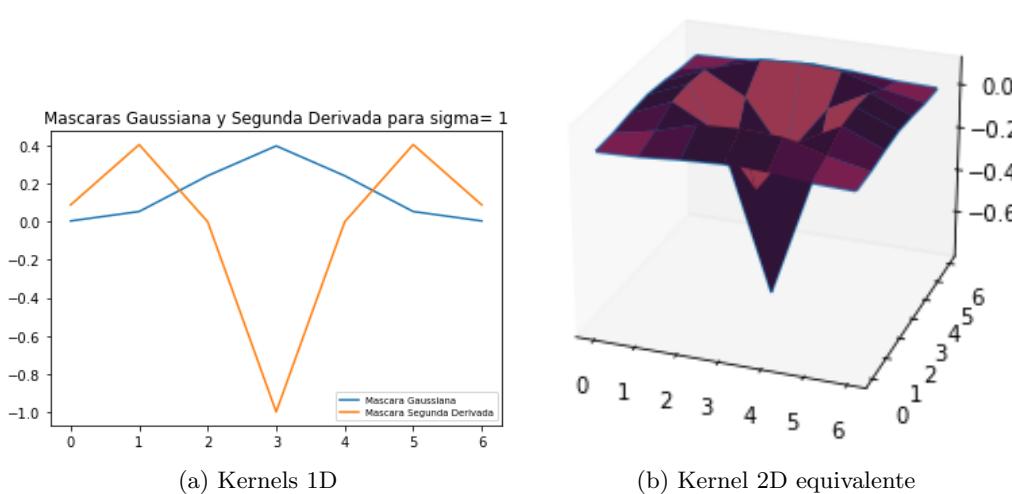


Figura 13: Kernels 1D y Kernel 2D equivalente para sigma = 1

El Kernel 2D equivalente se ha obtenido mediante el producto de las máscaras mediante la función `outer` de la librería **numpy**. Como se era de esperar, se obtiene la famosa forma de '*sombrero mexicano*' invertido.

Para un sigma = 3:

Calculamos los Kernels 1D utilizados en las convoluciones 1D:

```

1 Mascara Gaussiana
2 [0.00147945  0.00380424  0.00875346  0.01802341  0.03320773  0.05475029
3  0.08077532  0.106639    0.12597909  0.133176    0.12597909  0.106639
4  0.08077532  0.05475029  0.03320773  0.01802341  0.00875346  0.00380424
5  0.00147945]
6
7 Mascara Segunda Derivada
8 [0.009874663589548716, 0.01939632769321322, 0.032458532650138504,
   0.045111761078870896, 0.04925475728934246, 0.035528222636423606,
   -0.0, -0.04942823474795111, -0.09342809569449535,
   -0.1111111111111111, -0.09342809569449535, -0.04942823474795111,
   -0.0, 0.035528222636423606, 0.04925475728934246,
   0.045111761078870896, 0.032458532650138504, 0.01939632769321322,
   0.009874663589548716]
```

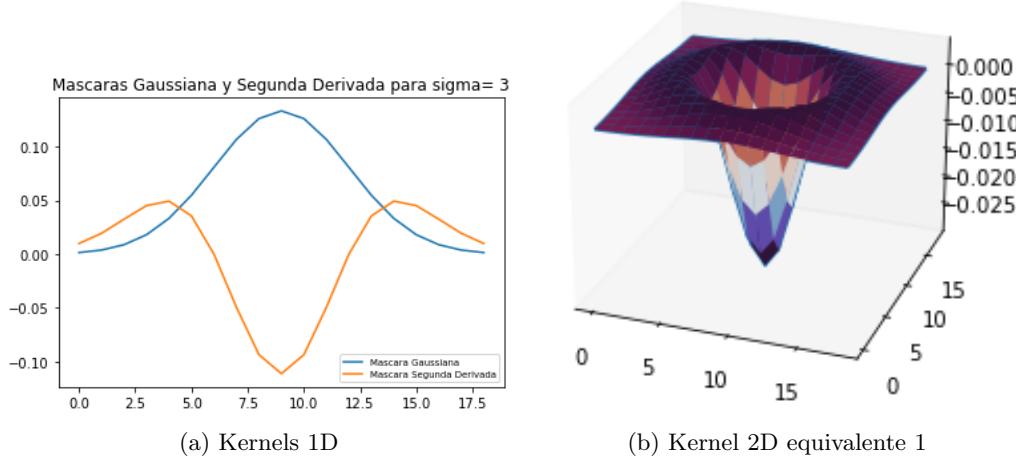
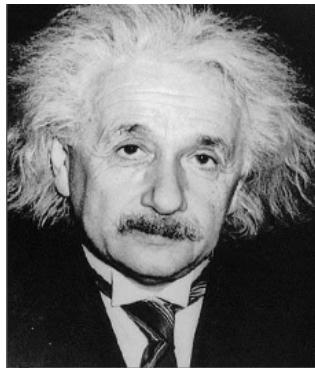


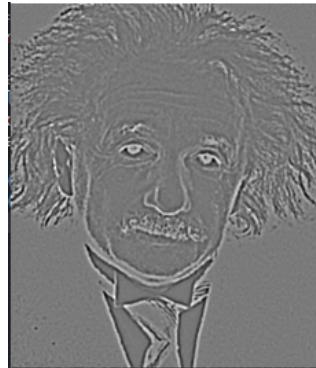
Figura 14: Kernels 1D y Kernel 2D equivalente para sigma = 3

Con un sigma de valor 3 también obtenemos un sombrero mexicano invertido, el cuál se puede apreciar mejor que para un sigma de 1, debido al mayor número de valores para la máscara.

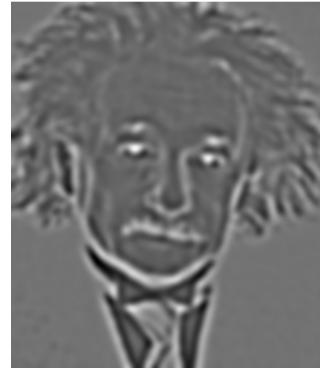
A continuación, se muestran los resultados:



(a) Imagen original



(b) Laplaciana de la Gaussiana, sigma = 1



(c) Laplaciana de la Gaussiana, sigma = 3

Figura 15: Imágenes obtenidas aplicando la Laplaciana de la Gaussiana

Los resultados de la Laplaciana de la Gaussiana son los esperados: un comportamiento diferencial y suavizante. No es difícil apreciar que un aumento de sigma provoca una mayor pérdida de definición, debido a la aplicación del filtro gaussiano, apreciando también un aumento del grosor de los bordes de la imagen.

Por último, se procede a comparar los resultados obtenidos con la función dada por **OpenCV**:

(cv2.Laplacian(src, ddepth, ksize, borderType))

donde:

- **src** es la imagen usada para obtener la Laplaciana de la Gaussiana.
- **ddepth** es la profundidad deseada de la imagen destino. Se ha utilizado el valor *cv2.CV_64F* porque durante la práctica se trabaja con las imágenes en valores flotantes (*astype(np.float64)*).
- **ksize** es el tamaño de la apertura con la que se calcula los filtros de segunda derivada. Calculamos el valor de K mediante los valores de sigma utilizados en este ejercicio y las funciones del ejercicio 1A.
- **borderType** es el tipo de borde utilizado, en nuestro caso, bordes reflejados, como en el resto de la práctica.

Los resultados obtenidos son:

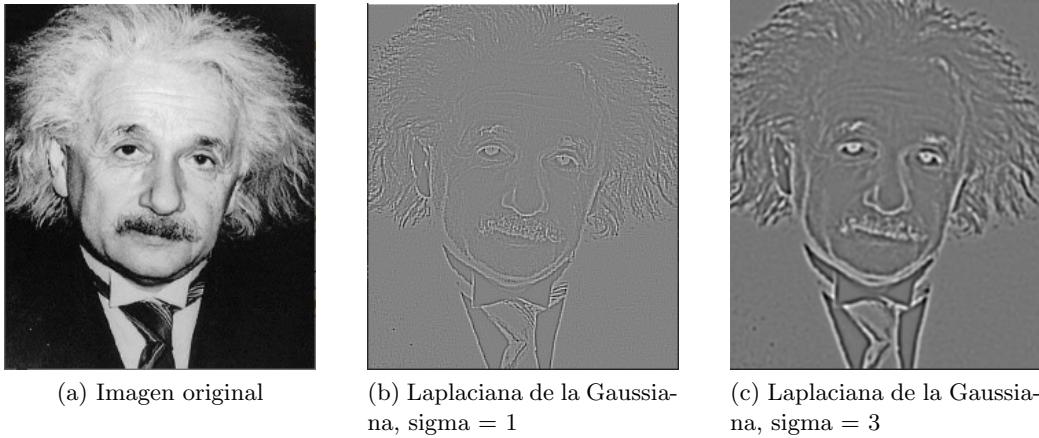


Figura 16: Imágenes obtenidas aplicando la Laplaciana de la Gaussiana de OpenCV

Se puede observar que, aunque los resultados son bastante similares, no son idénticos. El cálculo realizado manualmente produce un resultado más suavizado y con los bordes más marcados que con la función de OpenCV.

Esto se debe a que, de igual manera que en el apartado anterior también se podía visualizar una diferencia entre nuestros cálculos y el resultado de la función *GaussianBlur* ya que se priorizaba eficiencia ante precisión, extender dicho filtro de suavizado al cálculo de la Laplaciana conlleva también que el resultado siga siendo una aproximación menos definida pero más eficiente.

2. Ejercicio 2

2.1. Apartado A

Generar una función que genere una representación en pirámide Gaussiana de 4 niveles de una imagen. Mostrar ejemplos de funcionamiento usando bordes replicados o reflejados, y justificar la elección de los parámetros. Mostrar todos los niveles de la pirámide en una única imagen. Comparar los resultados con la pirámide obtenida usando las funciones OpenCV.

Para calcular la pirámide Gaussiana partiremos de la imagen original como base de la pirámide. Tras esto, por cada nivel que queramos en la pirámide, suavizamos la imagen con una máscara gaussiana para asegurar que no aparezca información espuria de alta frecuencia en la imagen de muestreo reducido (*aliasing*). El suavizado promedia los píxeles antes del submuestreo, de manera que la imagen reducida será más fiel a la original.

Una vez realizado dicho alisado, se debe aplicar un *downsampling*, es decir, quedarnos con la mitad de las filas y de las columnas (en nuestro caso, las pares), para quedarnos con una imagen más pequeña en cada nivel.

Para calcular la pirámide, se ha utilizado bordes reflejados, al igual que en ejercicios anteriores, 4 niveles y se ha probado con valores de sigma de (1, 2 y 3). Además, para mostrar todos los niveles en una única imagen, se ha reutilizado el método de concatenación de imágenes de la práctica anterior. Mostramos los resultados:

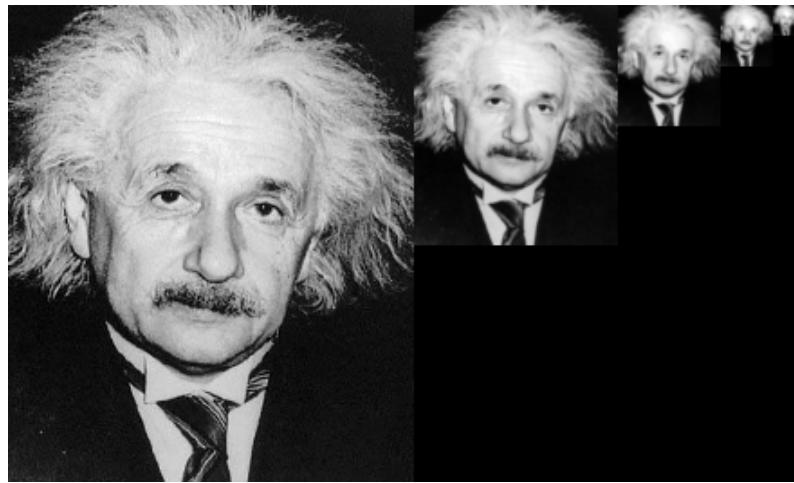


Figura 17: Pirámide Gaussiana con 4 niveles y sigma 1

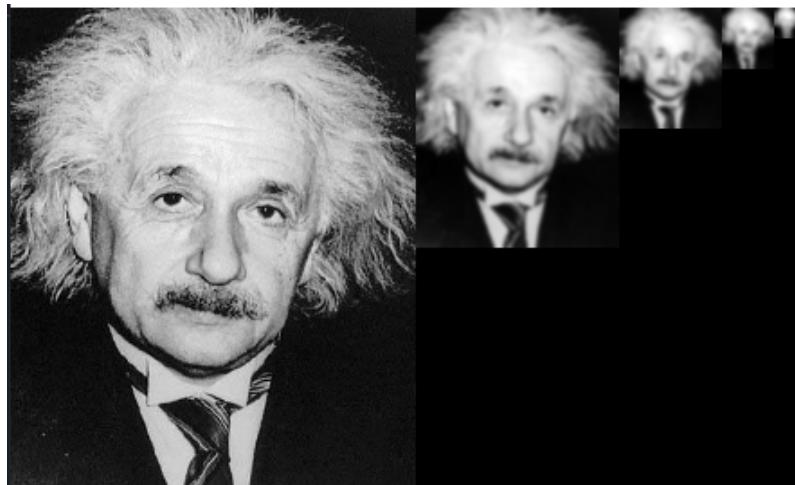


Figura 18: Pirámide Gaussiana con 4 niveles y sigma 2

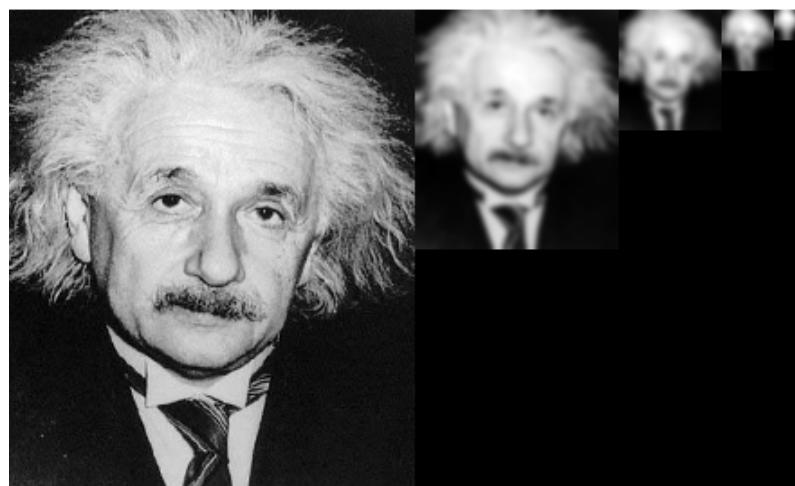


Figura 19: Pirámide Gaussiana con 4 niveles y sigma 3

Un valor de sigma de 1 sería recomendable, ya que es capaz de suavizar la imagen sin que pierda demasiada definición, al contrario de lo que ocurre al aumentar este valor.

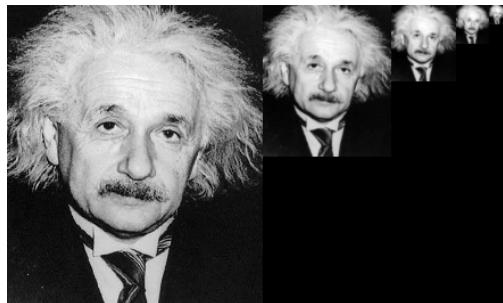
A continuación, se compara el resultado obtenido con la pirámide originada mediante las funciones de **OpenCV**:

La función utilizada para dicho cálculo es *pyrDown* [6]

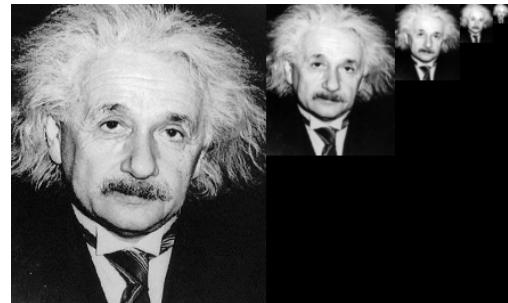
(cv2.pyrDown(src, borderType))

donde:

- **src** es la imagen usada en cada nivel.
- **borderType** es el tipo de borde utilizado.



(a) Pirámide Gaussiana de sigma 1



(b) Pirámide Gaussiana de OpenCV

Figura 20: Comparación de Pirámides Gaussianas

Como se puede observar, el resultado es prácticamente idéntico al de nuestra pirámide utilizando un valor de sigma 1.

2.2. Apartado B

Generar una función que genere una representación en pirámide Laplaciana de 4 niveles de una imagen. Mostrar ejemplos de funcionamiento usando bordes. Mostrar todos los niveles de la pirámide en una única imagen. Comparar los resultados con la pirámide obtenida usando las funciones OpenCV.

La pirámide Laplaciana constituye una forma de codificar eficientemente la información de una imagen, ya que permite la reconstrucción de una imagen original a partir de una reducida y suavizada sucesivas veces.

Se parte de la pirámide Gaussiana de una imagen, la cual en nuestro caso, usará bordes reflejados, 4 niveles y un sigma de 1 (el que dio mejores resultados en el apartado anterior) y, por cada uno de los niveles, se calculará el nivel i de la pirámide restando dicho nivel de la pirámide Gaussiana con el nivel $i+1$ de la pirámide Gaussiana reescalada con interpolación bilineal. De esta manera, en cada nivel nos quedaremos con una **diferencia de gaussianas**, dando un resultado aproximado al de la Laplaciana.

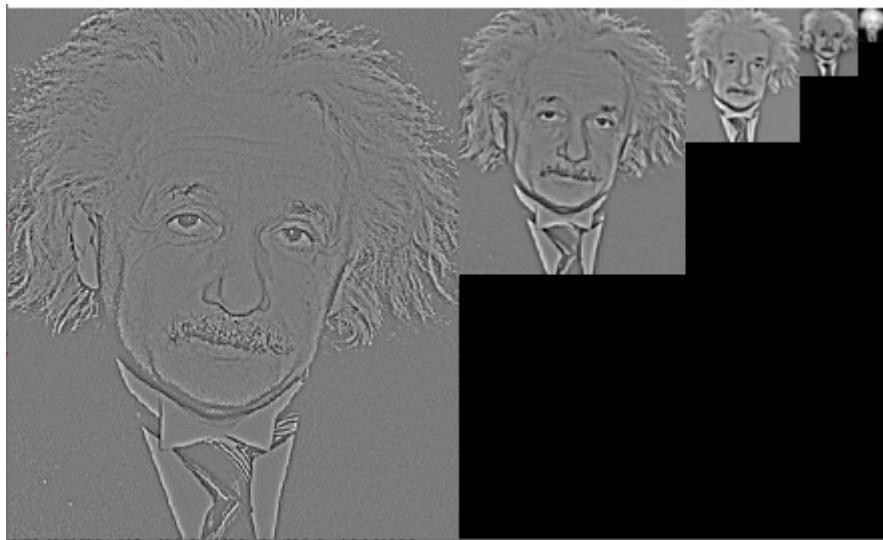


Figura 21: Pirámide Laplaciana con 4 niveles y sigma 1

Se puede observar que el nivel más alto (el último) es el que contiene las frecuencias bajas de la imagen, guardando los elementos que ocupan una mayor superficie y descartando aquellos píxeles que sean redundantes, mientras que los niveles más altos son los que registran las frecuencias altas a distintas escalas, resaltando los cambios bruscos de intensidad (mayoritariamente en los bordes de los objetos).

A continuación, se compara el resultado obtenido con la pirámide originada mediante las funciones de **OpenCV**:

Utilizaremos la función *pyrUp* [6] para reescalar al nivel anterior.

`(cv2.pyrUp(src, dstsize))`

donde:

- **src** es la imagen usada en cada nivel.
 - **dstsize** es el tamaño de la imagen de salida, el cual lo estableceremos al nivel anterior (el nivel $i+1$ en el tamaño i).

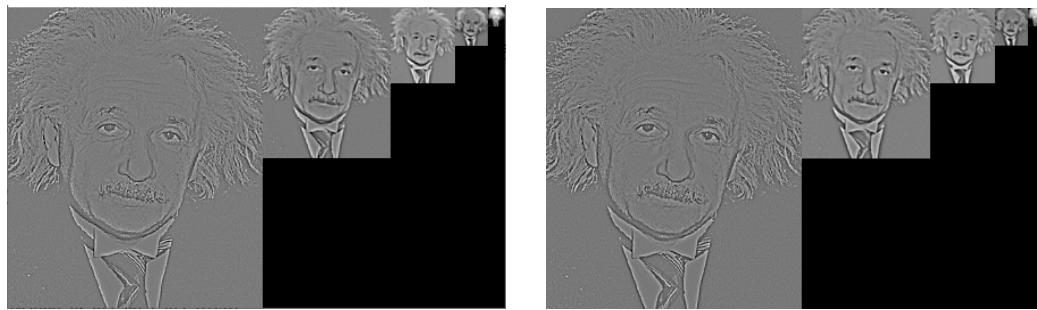


Figura 22: Comparación de Pirámides Laplacianas

Como se puede observar, el resultado del método desarrollado es equivalente a utilizar la función de OpenCV, pues proporciona resultados prácticamente idénticos.

2.3. Apartado C

Verificar el correcto funcionamiento de la pirámide Laplaciana por medio de mostrar su capacidad para recuperar la imagen original. Se debe mostrar el error obtenido en la aproximación, en términos de distancia Euclídea entre los niveles de gris de la imagen original y la imagen reconstruida.

De acuerdo a la explicación del ejercicio anterior, gracias a que la pirámide Laplaciana almacena las altas y bajas frecuencias de una imagen, es posible a partir de ella realizar una reconstrucción de la imagen conservando casi todos los detalles, cosa que sería imposible hacer con la pirámide Gaussiana, puesto que solo conserva bajas frecuencias.



Figura 23: Proceso de reconstrucción

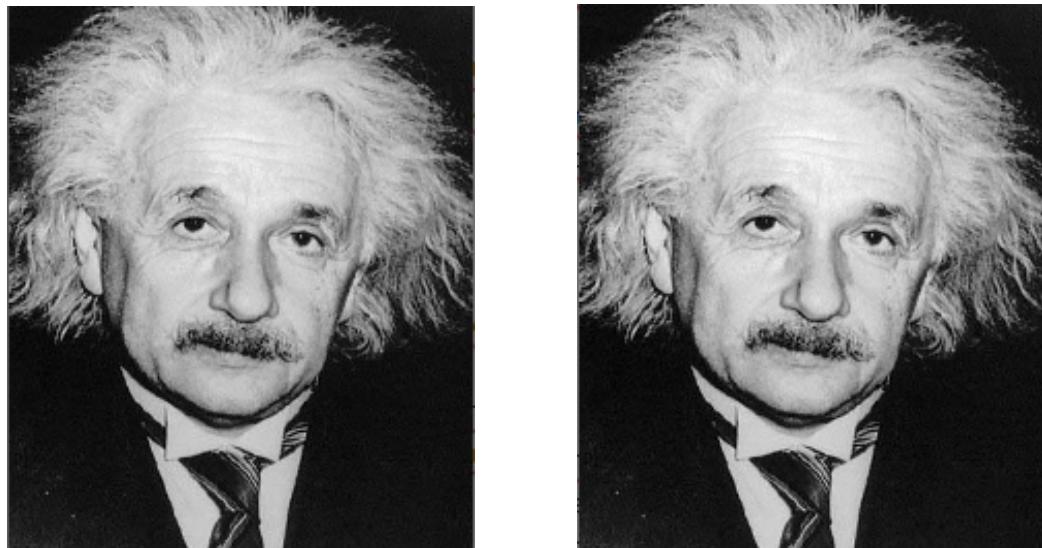


Figura 24: Reconstrucción de la imagen original a partir de la pirámide Laplaciana

A continuación, se calcula el error obtenido en la aproximación entre los niveles de gris de la imagen original y la imagen reconstruida. Para ello aplicamos la **distancia Euclídea**:

$$D(img1, img2) = \sqrt{\sum_{i=1}^n \sum_{j=1}^n (img1_{ij} - img2_{ij})^2}$$

1 El error entre las dos imágenes es: 3.5913945091812934e-13

Como era de esperar, el error obtenido es un valor muy cercano a 0, lo que demuestra que nuestra reconstrucción es fiel a la imagen original.

3. BONUS - Imágenes Híbridas

3.1. Apartado 1

Implementar una función que genere las imágenes de baja y alta frecuencia a partir de las parejas de imágenes (solo en la versión de imágenes de gris) .La imagen a usar para las bajas frecuencias y las altas respectivamente es importante para la calidad final. El valor de sigma (frecuencia de corte) más adecuado para cada imagen en cada pareja hay que encontrarlo por experimentación de prueba y error.

Escribir una función que muestre las tres imágenes (alta, baja e híbrida) en una misma ventana. Recordar que las imágenes después de una convolución contienen número flotantes que pueden ser positivos y negativos.

Como es explicado en el artículo *Hybrid images* [10], una imagen híbrida consiste en la suma de las frecuencias bajas de una imagen (suavizado) con las frecuencias altas de otra (detalles).

Se elige la *imagen de bajas frecuencias* aquella que tiene un aspecto más suavizado, y se obtendrán dichas frecuencias mediante un fuerte alisado para que solo quede casi una mancha sin detalles (de manera que más se tendrá que alejar el observador o más se deberá reducir su tamaño para poder distinguirla por encima de la otra imagen); mientras que la imagen de *altas frecuencias* será aquella que presente los bordes más acentuados ; a la cual primero aplicaremos un suavizado y después restaremos esta imagen resultante a la original para que queden extraídos sus detalles.

Para cada una de las dos imágenes de una pareja, se eligen *distintos* valores de sigma:

- **A mayor sigma para la imagen de altas frecuencias, más fuertes serán sus detalles.** Al restar a la imagen original otra imagen que ha perdido mucho detalle, la diferencia entre ambas será mayor y, por ende, se intensificarán más los detalles.
- **A mayor sigma para la imagen de bajas frecuencias, más suavizada quedará la imagen.** Esta idea ha ido quedando clara durante toda la práctica, un aumento de sigma al aplicar el filtro gaussiano difumina más la imagen.

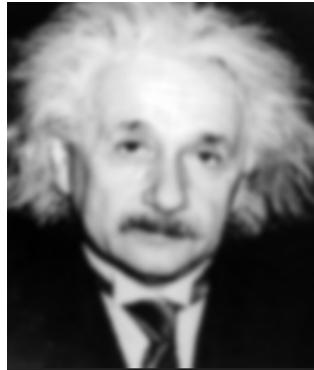
Para encontrar unos parámetros adecuados, se ha probado mediante experimentación de prueba y error, aunque se ha tomado la decisión de que el sigma

para bajas frecuencias siempre sea mayor que el de las altas frecuencias, de manera que el suavizado sea fuerte y cause el efecto deseado (al acercarse y alejarse de la imagen).

Realizar la composición con, al menos, 3 de las parejas de imágenes y construir pirámides Gaussianas de, al menos, 4 niveles con las imágenes resultado. Explicar el efecto que se observa.

IMÁGENES DE EINSTEIN Y MARILYN

-Sigma frecuencias bajas = 2 y Sigma frecuencias altas = 1



(a) Imagen de bajas frecuencias, sigma = 2



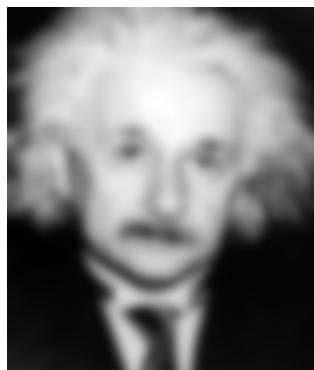
(b) Imagen de altas frecuencias, sigma = 1



(c) Imagen híbrida

Figura 25: Creación de imagen híbrida.

-Sigma frecuencias bajas = 4 y Sigma frecuencias altas = 2



(a) Imagen de bajas frecuencias, sigma = 4



(b) Imagen de altas frecuencias, sigma = 2



(c) Imagen híbrida

Figura 26: Creación de imagen híbrida.

-Sigma frecuencias bajas = 5 y Sigma frecuencias altas = 3



(a) Imagen de bajas frecuencias, sigma = 5

(b) Imagen de altas frecuencias, sigma = 3

(c) Imagen híbrida

Figura 27: Creación de imagen híbrida.

-Sigma frecuencias bajas = 7 y Sigma frecuencias altas = 5



(a) Imagen de bajas frecuencias, sigma = 7

(b) Imagen de altas frecuencias, sigma = 5

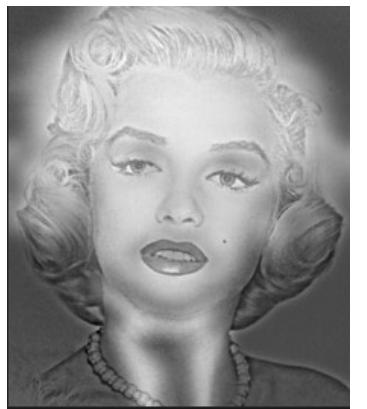
(c) Imagen híbrida

Figura 28: Creación de imagen híbrida.

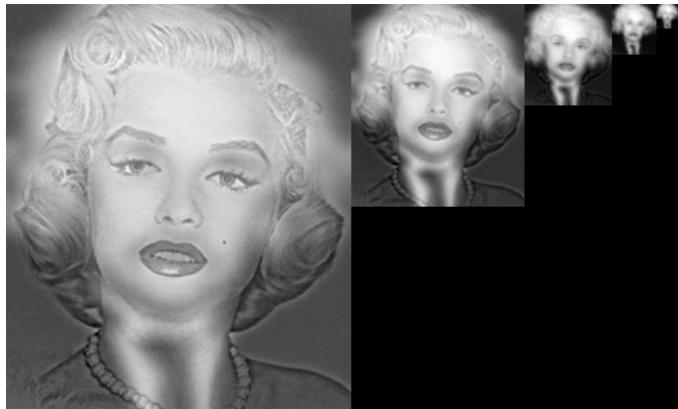
Atendiendo a los resultados para esta pareja de imágenes, observamos que para unos sigmas(bajo,alto) de (2,1) y unos sigmas de (4,2) los detalles de Marilyn no están lo suficientemente marcados como para notarse al superponerse con Einstein; mientras que, para unos sigmas de (7,5), apenas se puede observar a Einstein, incluso aunque nos alejemos notablemente de la imagen. Es por ello que los valores elegidos finalmente para mostrar la pirámide Gaussiana serán de ***sigma = 5 para frecuencias bajas y sigma = 3 para frecuencias altas***.

Para mostrar la pirámide, simplemente se hará uso de la construcción realizada en esta misma práctica.

Mostramos la pirámide Gaussiana:



(a) Imagen híbrida



(b) Pirámide Gaussiana

Figura 29: Imagen híbrida y su pirámide Gaussiana.

IMÁGENES DEL PÁJARO Y EL AVIÓN

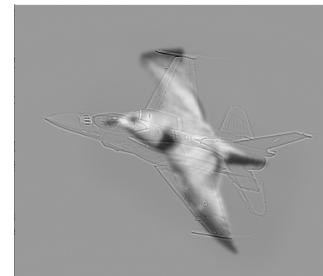
-Sigma frecuencias bajas = 2 y Sigma frecuencias altas = 1



(a) Imagen de bajas frecuencias, sigma = 2



(b) Imagen de altas frecuencias, sigma = 1



(c) Imagen híbrida

Figura 30: Creación de imagen híbrida.

-Sigma frecuencias bajas = 5 y Sigma frecuencias altas = 3



(a) Imagen de bajas frecuencias, sigma = 5



(b) Imagen de altas frecuencias, sigma = 3



(c) Imagen híbrida

Figura 31: Creación de imagen híbrida.

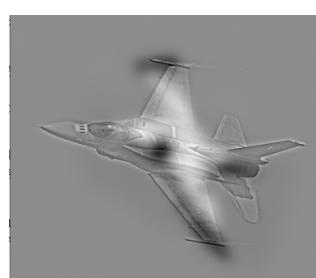
-Sigma frecuencias bajas = 7 y Sigma frecuencias altas = 3



(a) Imagen de bajas frecuencias, sigma = 7



(b) Imagen de altas frecuencias, sigma = 3



(c) Imagen híbrida

Figura 32: Creación de imagen híbrida.

-Sigma frecuencias bajas = 10 y Sigma frecuencias altas = 3



(a) Imagen de bajas frecuencias, sigma = 10



(b) Imagen de altas frecuencias, sigma = 3



(c) Imagen híbrida

Figura 33: Creación de imagen híbrida.

Atendiendo a los resultados para esta pareja de imágenes, observamos que para unos sigmas(bajo,alto) de (2,1) los detalles del avión son prácticamente indistinguibles. Para unos sigmas de (5,3) empezamos a tener un mejor resultado, los detalles del avión se aprecian bastante bien. Sin embargo, se ha decidido aumentar el sigma de las frecuencias bajas para comrpobar si la visualización del pájaro puede mejorar. Tras un aumento a los valores 7 y 10, observamos como la hibridación se aprecia mejor. Aunque los dos últimos de resultados de la experimentación no sean muy diferentes entre ellos y se aprecie bien ambas figuras, elegiré un ***sigma = 7 para frecuencias bajas y sigma = 3 para frecuencias altas***, ya que al pasar de sigma 7 a 10, aunque se aprecie bien, quizás el pájaro esté ya demasiado borroso.

Mostramos la pirámide Gaussiana:

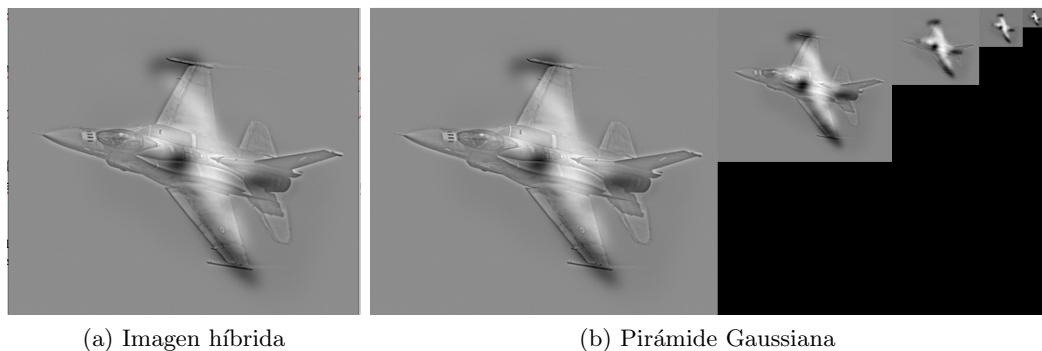


Figura 34: Imagen híbrida y su pirámide Gaussiana

IMÁGENES DEL PEZ Y EL SUBMARINO

-Sigma frecuencias bajas = 2 y Sigma frecuencias altas = 1

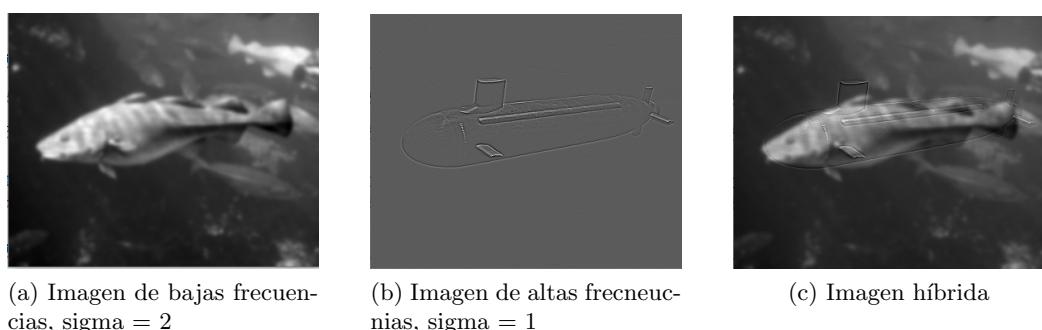
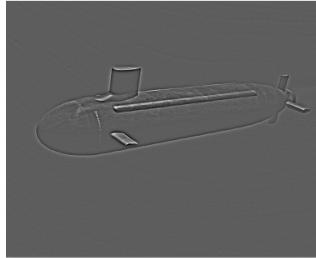


Figura 35: Creación de imagen híbrida.

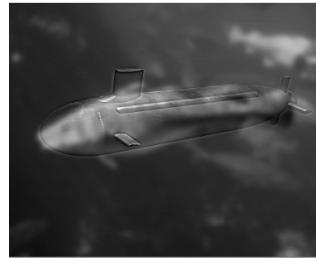
-Sigma frecuencias bajas = 3 y Sigma frecuencias altas = 2



(a) Imagen de bajas frecuencias, sigma = 3



(b) Imagen de altas frecuencias, sigma = 2



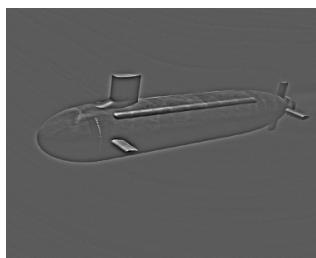
(c) Imagen híbrida

Figura 36: Creación de imagen híbrida.

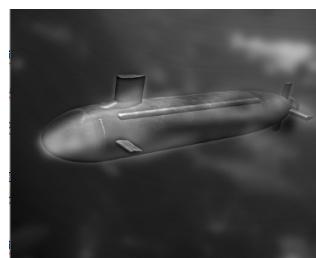
-Sigma frecuencias bajas = 5 y Sigma frecuencias altas = 3



(a) Imagen de bajas frecuencias, sigma = 5



(b) Imagen de altas frecuencias, sigma = 3



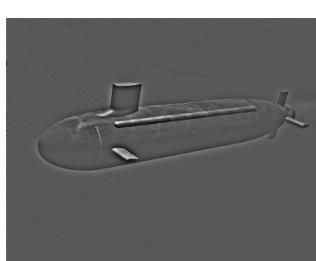
(c) Imagen híbrida

Figura 37: Creación de imagen híbrida.

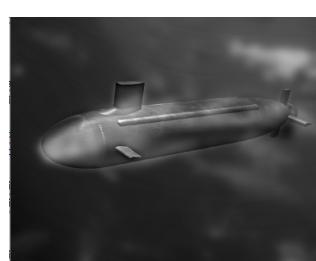
-Sigma frecuencias bajas = 5 y Sigma frecuencias altas = 4



(a) Imagen de bajas frecuencias, sigma = 5



(b) Imagen de altas frecuencias, sigma = 4



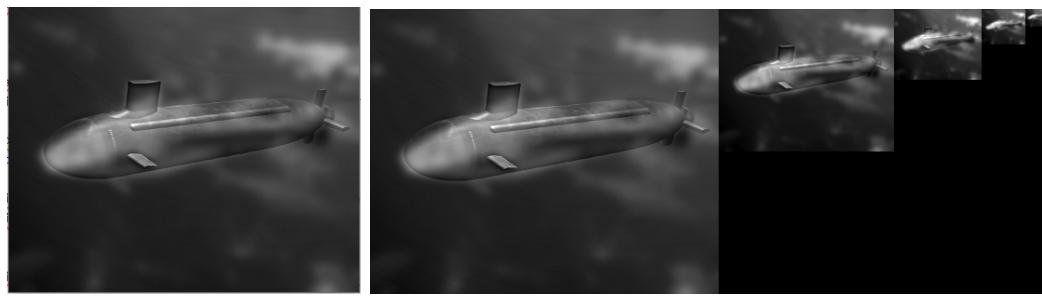
(c) Imagen híbrida

Figura 38: Creación de imagen híbrida.

Atendiendo a los resultados para esta pareja de imágenes, observamos que para

unos sigmas(bajo,alto) de (2,1) y (3,2) apenas se notan los detalles del submarino y el pez tiene un suavizado no demasiado pronunciado. Para unos sigmas de (5,3), el pescado se encuentra lo suficientemente suavizado como para crear un buen efecto, pero se considera que un sigma de valor 4 para las altas frecuencias consigue una mejor hibridación al resaltar un poco más el submarino. Aumentar el valor de sigma para las altas frecuencias más de este valor haría que el submarino tomase demasiado protagonismo y sería más costoso distinguir la figura del pez. Por tanto, se elige finalmente un *sigma = 5 para frecuencias bajas y sigma = 4 para frecuencias altas.*

Mostramos la pirámide Gaussiana:



(a) Imagen híbrida

(b) Pirámide Gaussiana

Figura 39: Imagen híbrida y su pirámide Gaussiana

Tras haber mostrado las pirámides Gaussianas para las tres parejas de imágenes, se puede asegurar que el efecto causado es similar al que ocurre cuando nos acercamos/alejamos de la pantalla. En la imagen de mayor tamaño se distingue sin problema la imagen de altas frecuencias y al disminuir el tamaño de la imagen en los sucesivos niveles, comienza a verse cada vez mejor la imagen de bajas frecuencias.

3.2. Apartado 2

Realizar todas las parejas de imágenes híbridas en su formato a color.

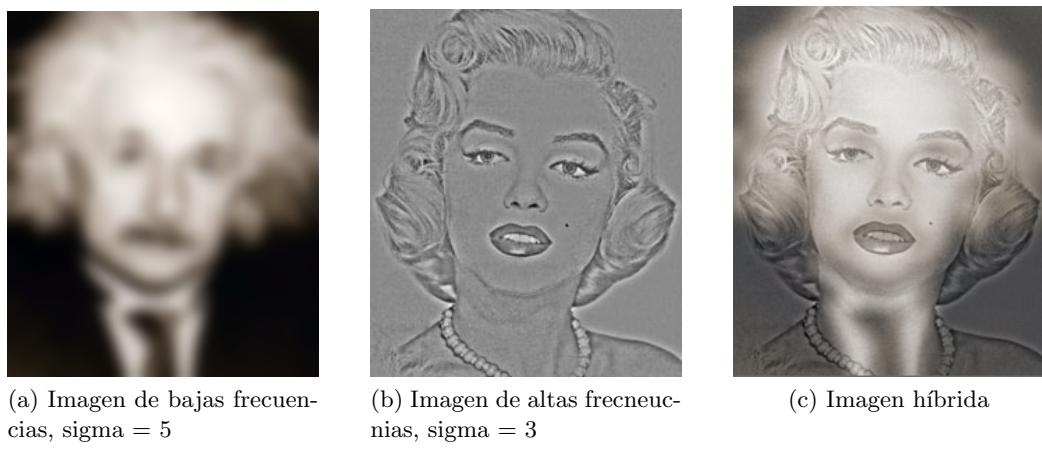
Para la resolución de este ejercicio, se ha visto necesario adaptar la convolución 2D realizada en la práctica para que tenga en cuenta las imágenes tribanda. En esencia, el único cambio que se ha tenido que hacer es considerar el eje z al guardar las dimensiones (con la función *shape*) para evitar un error.

Realmente, el verdadero cambio se encuentra en el método que calcula las imágenes de altas frecuencias, bajas frecuencias, ya que en vez de convolucionar una única vez con la imagen (monobanda), con la imagen tribanda se calculan **tres convoluciones**, modificando el *eje z* de la imagen con sus valores 0,1 y 2, es decir, **B-G-R**.

Una vez se tienen las tres imágenes en los tres diferentes canales, se deben concatenar sus matrices (2D) en un único array 3D [11]. Para conseguir esto, utilizamos dos funciones de **NumPy**: *np.newaxis* [12], que aumenta en uno la dimensión de la matriz proporcionada y *np.concatenate* [13], que realiza la propia concatenación de las imágenes en el eje especificado (en nuestro caso, el *eje z* o 2).

Se muestran las parejas de imágenes híbridas para los sigmas elegidos en el apartado anterior:

IMÁGENES DE EINSTEIN Y MARILYN



(a) Imagen de bajas frecuencias, sigma = 5

(b) Imagen de altas frecuencias, sigma = 3

(c) Imagen híbrida

Figura 40: Creación de imagen híbrida.

IMÁGENES DEL PÁJARO Y EL AVIÓN



(a) Imagen de bajas frecuencias, sigma = 7



(b) Imagen de altas frecuencias, sigma = 3



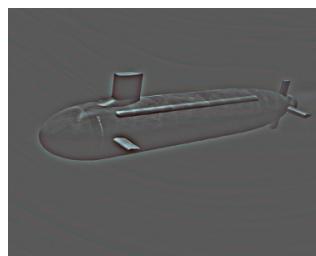
(c) Imagen híbrida

Figura 41: Creación de imagen híbrida.

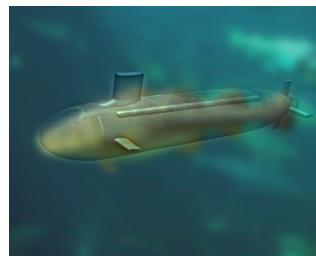
IMÁGENES DEL PEZ Y EL SUBMARINO



(a) Imagen de bajas frecuencias, sigma = 5



(b) Imagen de altas frecuencias, sigma = 4

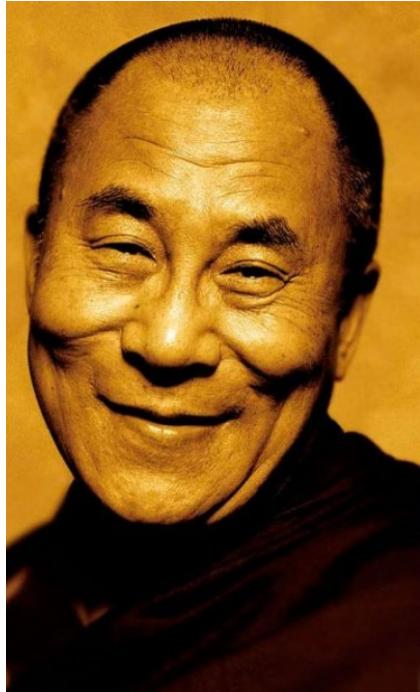


(c) Imagen híbrida

Figura 42: Creación de imagen híbrida.

3.3. Apartado 3

Realizar una imagen híbrida con, al menos, una pareja de imágenes de su elección que hayan sido extraídas de imágenes más grandes. Justifique la elección y todos los pasos que realiza.



(a) Foto de Dalai Lama



(b) Foto de una llama feliz

Figura 43: Imágenes base para la producción de una imagen híbrida

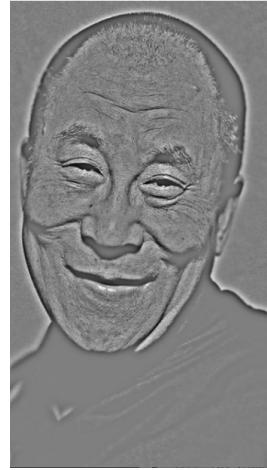
Para la realización de este apartado he elegido una foto de Dalai Lama y una de una llama feliz. He tomado esta decisión ya que ambas fotografías utilizan una gama de colores similares, ambos sujetos aparecen en una posición similar y poseen un alto parecido entre ellos.

Primero he tenido que igualar las dimensiones de ambas imágenes. Para ello, he utilizado la función de **OpenCV**: *resize*, y he interpolado ambas imágenes a las dimensiones mínimas de entre las dos. Una vez hecho esto, he seleccionado la imagen de Dalai como la imagen de altas frecuencias debido ya que posee sombras y colores más oscuros, los cuales serán mejor detectados como bordes; y por tanto, la llama feliz será la imagen de bajas frecuencias.

Los sigmas elegidos ha sido mediante experimentación de los parámetros, decidiéndome al final por un ***sigma = 7 para frecuencias bajas y sigma = 5 para frecuencias altas***. El resultado final para las imágenes en escala de grises y a color es:



(a) Imagen de bajas frecuencias, sigma = 7



(b) Imagen de altas frecuencias, sigma = 5



(c) Imagen híbrida

Figura 44: Creación de imagen híbrida.



(a) Imagen de bajas frecuencias, sigma = 7

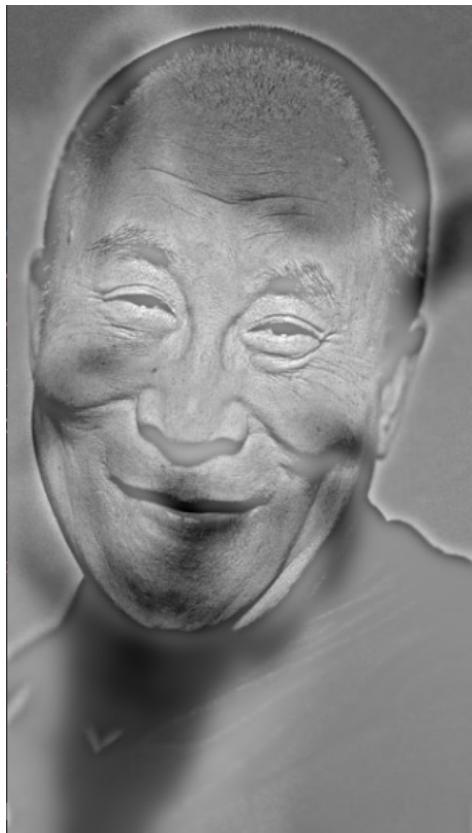


(b) Imagen de altas frecuencias, sigma = 5



(c) Imagen híbrida

Figura 45: Creación de imagen híbrida.



(a) Foto de Dalai Lama



(b) Foto de una llama feliz

Figura 46: Imágenes híbridas finales

Referencias

- [1] getDerivKernels. *getDerivKernels method specification*
https://docs.opencv.org/4.5.3/d4/d86/group__imgproc__filter.html#ga6d6c23f7bd3f5836c31cf994fc4aea
- [2] matmul. *matmul method specification*
<https://numpy.org/doc/stable/reference/generated/numpy.matmul.html>
- [3] GaussianBlur. *GaussianBlur method specification*
https://docs.opencv.org/4.5.3/d4/d86/group__imgproc__filter.html#gaabe8c836e97159a9193fb0b11ac52cf1
- [4] Laplacian. *The Laplacian of Gaussian filter*
<https://www.crisluengo.net/archives/1099/>
- [5] Laplacian Method. *Laplacian() method, openCV documentation*
https://docs.opencv.org/3.4.15/d4/d86/group__imgproc__filter.html#gad78703e4c8fe703d479c1860d76429e6
- [6] Image Pyramid. *Use the OpenCV functions pyrUp() and pyrDown() to down-sample or upsample a given image.*
https://docs.opencv.org/3.4.15/d4/d1f/tutorial_pyramids.html
- [7] Laplacian Pyramid *Laplacian Image Pyramids Algorithm Overview*
<http://cs.brown.edu/courses/csci1430/2011/results/proj1/georgem/>
- [8] High Frequency Filter *High Frequency Filter Overview*
http://www.dimages.es/Tutorial%20A.I/enhancement/alta_freq.htm
- [9] Low Frequency Filter *Low Frequency Filter Overview*
[http://www.dimages.es/Tutorial%20A.I/enhancement/filtropb.htm/](http://www.dimages.es/Tutorial%20A.I/enhancement/filtropb.htm)
- [10] Hybrid images *Hybrid images technique*
https://stanford.edu/class/ee367/reading/OlivaTorralb_Hybrid_Siggraph06.pdf
- [11] Color Hybrid images *Concatenate three images (rgb) into one*
<https://wjngkoh.wordpress.com/2015/01/29/numpy-concatenate-three-images-rgb-into-one/>
- [12] newaxis *how to use newaxis method*
<https://qastack.mx/programming/29241056/>
- [13] concatenate *Concatenate methos specification*
<https://numpy.org/doc/stable/reference/generated/numpy.concatenate.html>