



UNIVERSIDAD
DE GRANADA



Práctica 1: Convolución y Derivadas

Visión por Computador

Autor:

Lugli, Valentino Glauco · YB0819879

Índice

1 Notas sobre la implementación	2
2 Implementación de Funciones Básicas	3
2.1 Generación de máscara discreta 1D gaussiana y derivadas	3
2.2 Generación de máscara discreta por aproximación binomial	8
2.3 Implementación de convolución 2D separable	9
2.4 Cálculo de Laplaciana de la Gaussiana	14
3 Implementación de Funciones Avanzadas	17
3.1 Generación de Pirámide Gaussiana	17
3.2 Generación de Pirámide Laplaciana	19
3.3 Recuperación de imagen por medio de Pirámide Laplaciana	20
4 Bonus	21
4.1 Generación de imágenes híbridas en escala de grises	21
4.2 Generación de imágenes híbridas a color	24
4.3 Imágenes híbridas a elección libre	27

1. Notas sobre la implementación

La práctica fue realizada completamente en Python haciendo uso del IDE “Spyder”, este IDE permite tener “celdas” de código que se pueden ejecutar independientemente del código que tienen antes o después, es decir, es como una celda de código de un Colab Notebook: no es necesario ejecutar todo el programa entero, se puede ir paso a paso.

Al igual que un Colab Notebook, se han de ejecutar las celdas en orden la primera vez que se carga el fichero para tener las funciones y variables de celdas anteriores en memoria. Una vez realizado esto se pueden ejecutar las celdas en cualquier orden.

Las celdas se delimitan con un comentario de la forma “`# %%`” y pueden ejecutarse haciendo



Ctrl+Enter en una celda resaltada o bien haciendo clic en el ícono que está a la derecha del ícono de “Play” que ejecuta el código entero secuencialmente.

2. Implementación de Funciones Básicas

2.1. Generación de máscara discreta 1D gaussiana y derivadas

Dada la función gaussiana (1) y su primera (2) y segunda derivada (3), descritas como

$$f(x) = c \cdot e^{\frac{-x^2}{2\sigma^2}} \quad (1)$$

$$f'(x) = -\frac{x \cdot f(x)}{\sigma^2} \quad (2)$$

$$f''(x) = \frac{f(x) \cdot (x^2 - \sigma^2)}{\sigma^4} \quad (3)$$

Se implementa la función `gaussianMask(dx, sigma, maskSize)` la cual permite por el parámetro `dx` obtener una máscara de alisamiento gaussiana (`dx = 0`), una máscara de la primera derivada gaussiana (`dx = 1`) o de la segunda (`dx = 2`).

Se definen 3 funciones para cada cálculo, `gaussian(x, sigma)`, `gaussianFirstD(x, sigma)` y `gaussianSecondD(x, sigma)` que calculan respectivamente la gaussiana, su primera y segunda derivada tal y como están definidas matemáticamente esas expresiones en (1), (2) y (3), con la excepción que la constante c se iguala a 1.

Para obtener la máscara discreta se puede pasar por parámetro el valor de la constante σ (`sigma`) la cual indica que tan ancha o angosta será la campana gaussiana o la longitud que se desea tenga la máscara discreta (`maskSize`).

Esta máscara contiene el valor de la función gaussiana para ciertos puntos, si bien podrían ser múltiples finitos puntos decimales, se decidió por facilidad que la función gaussiana sea calculada sobre valores enteros. Estos valores han de asemejarse lo más posible a la función original y por lo tanto, al saber que el 99 % de los valores de una función gaussiana se encuentran a 3 desviaciones estándar de la media, si se pasa por parámetro el valor de σ , el tamaño de la máscara T sería $T = 2 \cdot [3 \cdot \sigma] + 1$, dos veces puesto que se está tomando la parte positiva y negativa de la campana, y se suma 1 por que también se incluye el cero.

Si se pasa por parámetro el tamaño de la máscara, entonces se calcula un σ que sea válido para ese tamaño, es decir, $\sigma = \frac{T-1}{6}$, notar que se espera que el tamaño T se asume es impar, pues se están trabajando con máscaras simétricas.

Cabe notar que si se pasan por parámetro tanto el tamaño de la máscara como el σ , el valor del σ toma prioridad, ya que puede que el tamaño no se corresponda.

Una vez obtenido el valor del σ y del tamaño de la máscara, se crea un vector que contendrá la máscara utilizando comprensión de listas de Python el cual se rellena comenzando por los valores negativos hacia los positivos, se procede llenando el vector de la siguiente manera:

$$\text{Máscara : } [f(-k), f(-k+1), \dots, f(0), f(k+1), \dots, f(k)] \quad (4)$$

Siendo una variable $k = \frac{T-1}{2}$, y $f(x)$ puede ser alguna de las tres funciones anteriormente mencionadas.

El resultado de esta máscara se normaliza para que sume 1 en el caso de que se pida la máscara gaussiana, puesto que como es un filtro de alisamiento, no debe afectar la imagen más allá de desenfocarla.

El vector resultante, sea de cualquier función, se convierte en un vector de Numpy antes de ser retornado de la función.

La función `getDerivKernel(dx, dy, ksize)` permite obtener también máscaras de alisamiento y derivadas, los valores `dx`, `dy` indican qué derivada se desea obtener en función de `x` e `y`, mientras que `ksize` indica el tamaño que se quiere para esas máscaras.

Se muestra la comparación lado a lado máscaras de tamaño 5, 7 y 9 de las máscaras gaussianas y sus derivadas obtenidas por la función implementada y las máscaras que provee OpenCV para ese tamaño y orden de derivada en las Figuras 1, 2, 3 respectivamente.

Se puede observar que, si bien no son idénticas las funciones puesto que la implementación de OpenCV realmente está produciendo las máscaras de los filtros de Scharr y Sobel, la forma que poseen es muy similar a la de una función gaussiana y sus derivadas.

Esto se debe principalmente porque los filtros de Scharr y Sobel son utilizados en la detección de bordes, lo que es una aproximación a la derivada de una imagen: algo para lo que también se utiliza principalmente la función Gaussiana y sus derivadas, por lo tanto es natural que todas las funciones posean una forma similar puesto que están en búsqueda de obtener el mismo objetivo.

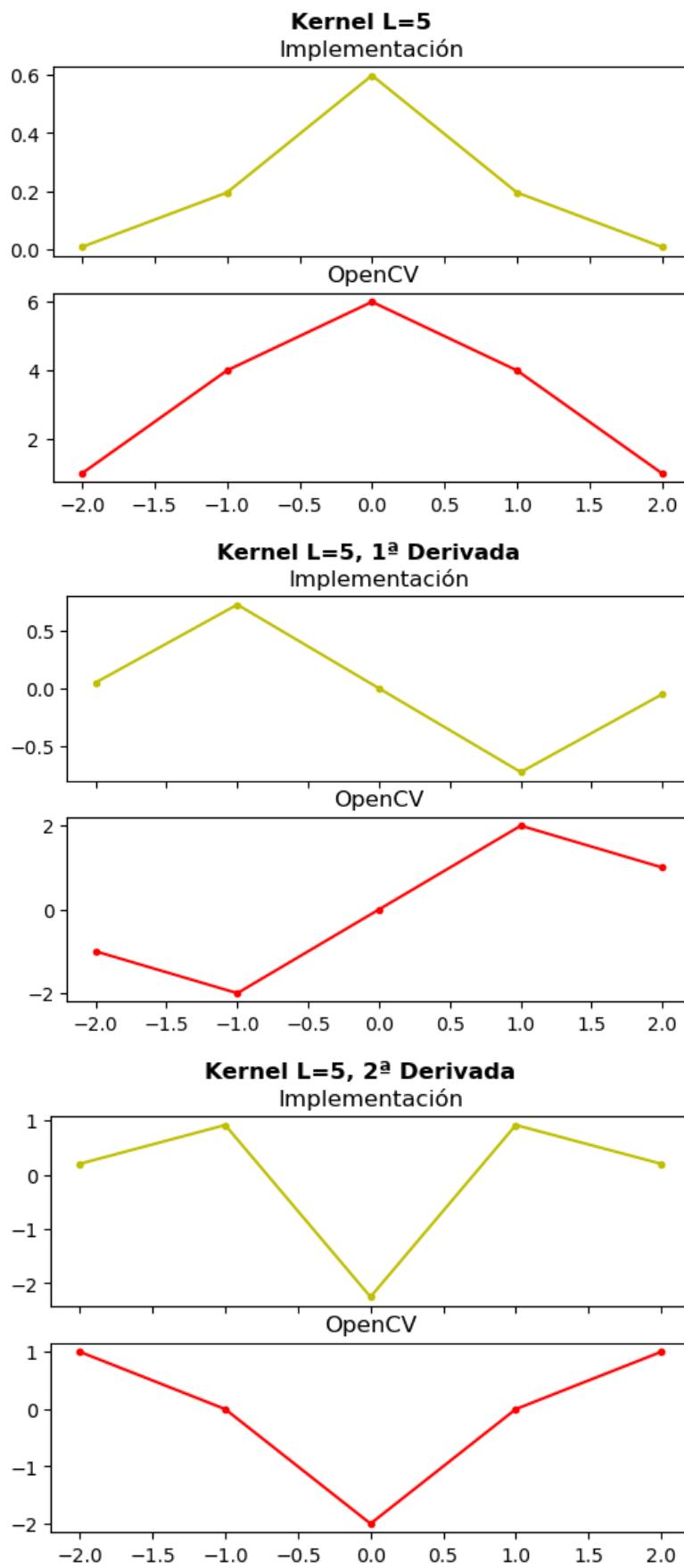


Figura 1: Máscaras de Tamaño 5

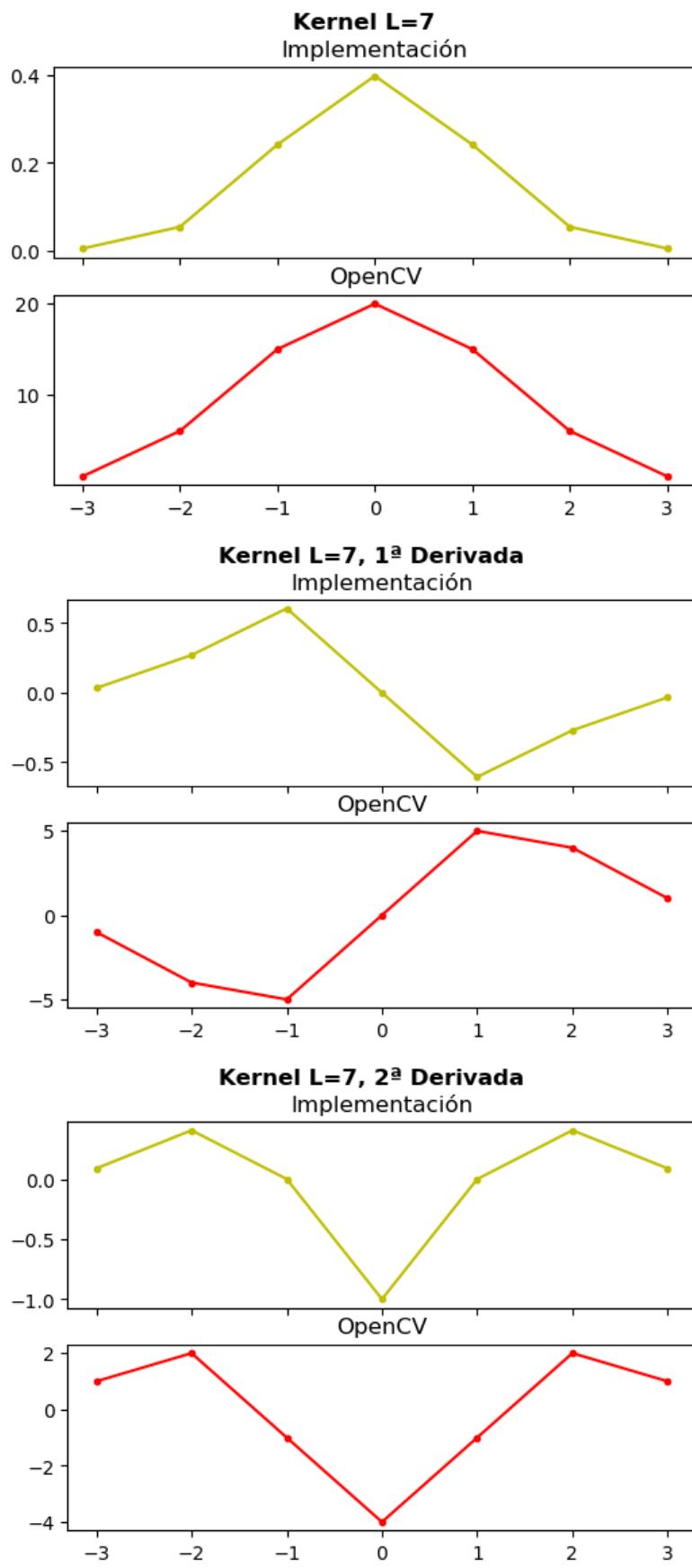


Figura 2: Máscaras de Tamaño 7

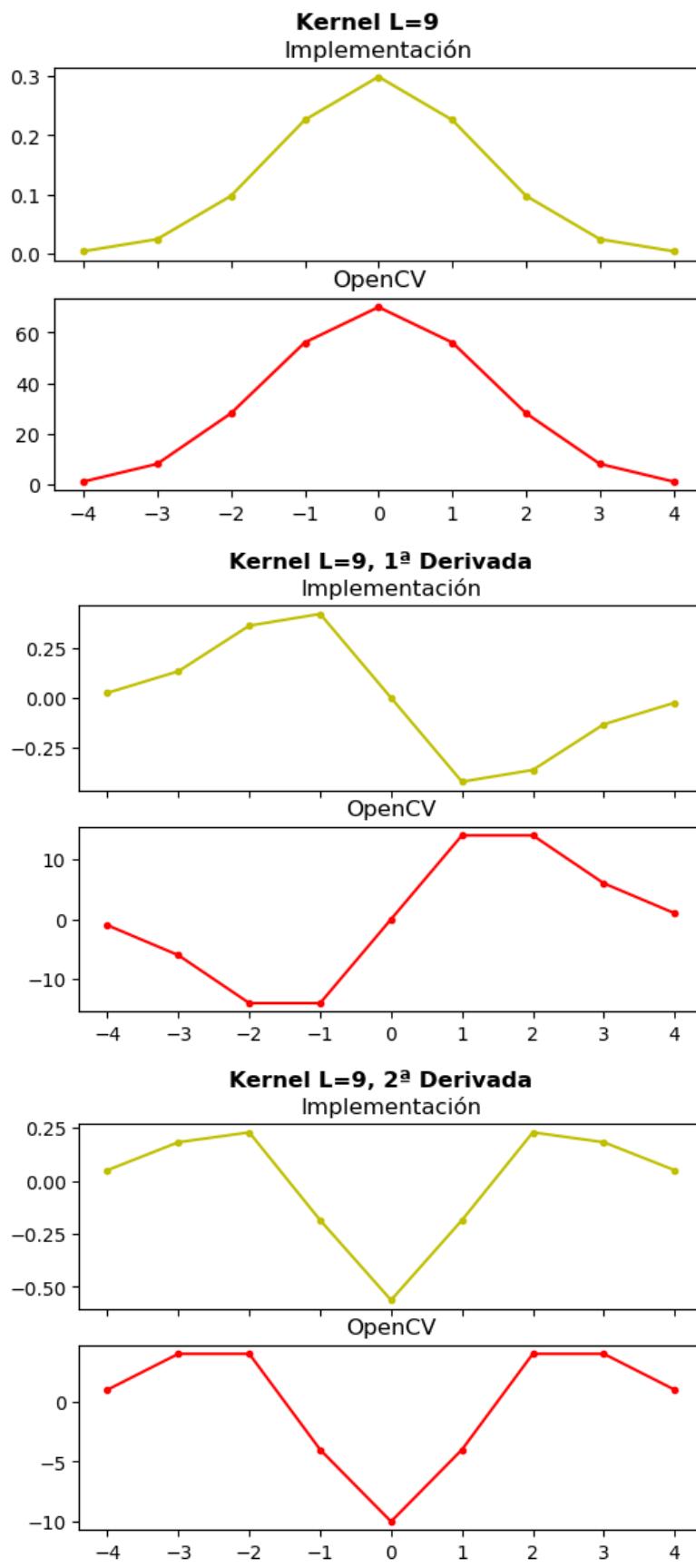


Figura 3: Máscaras de Tamaño 9

2.2. Generación de máscara discreta por aproximación binomial

Es posible obtener también máscaras de alisamiento y sus derivadas utilizando el teorema del binomio, partiendo de una máscara binomial $[1, 2, 1]$ y una máscara de derivada $[-1, 0, 1]$.

La clave para poder obtener, por ejemplo, las máscara de alisamiento de tamaño 5 es convolucionando con sí misma la máscara binomial (5), y para la de tamaño 7 es convolucionar la máscara de tamaño 5 con la máscara binomial, (6).

De la misma manera, se puede obtener las máscaras de derivadas de primer orden por medio de la convolución de la máscara binomial con una máscara de derivada, por ejemplo, para tamaño 5 (7) se realiza la convolución de la máscara de derivada de tamaño 3 con la binomial, y para el tamaño 7 (8) se repite el proceso pero ahora es la convolución con la máscara de derivada de tamaño 5.

$$[1, 2, 1] * [1, 2, 1] = [1, 4, 6, 4, 1] \quad (5)$$

$$[1, 4, 6, 4, 1] * [1, 2, 1] = [1, 6, 15, 20, 15, 6, 1] \quad (6)$$

$$[1, 2, 1] * [-1, 0, 1] = [-1, -2, 0, 2, 1] \quad (7)$$

$$[1, 2, 1] * [-1, -2, 0, 2, 1] = [-1, -4, -5, 0, 5, 4, 1] \quad (8)$$

Esto se puede realizar en Python utilizando la función de Numpy llamada `np.convolve(a, v)`, la cual toma dos vectores 1D `a` y `v` y realiza la convolución entre las mismas, se realizó de esta manera puesto que se indicó al autor que para este ejercicio en particular se podían utilizar las funciones ya implementadas en Numpy.

La función previamente utilizada de Numpy con los parámetros `getDerivKernels(0, 1, 9)` obtiene como resultado las siguientes máscaras, en X e Y respectivamente.

$$[1, 8, 28, 56, 70, 56, 28, 8, 1] \quad (9)$$

$$[-1, -6, -14, -14, 0, 14, 14, 6, 1] \quad (10)$$

Es un patrón muy similar al que se ha visto en el cálculo de las máscaras de tamaño 5 y 7, y en efecto, si se realiza la convolución de la máscara de alisamiento tamaño 7 con la binomial (11) se obtiene (9); de misma forma una convolución de la máscara de derivadas de tamaño 7 con la binomial (12) obtiene (10).

$$[1, 2, 1] * [1, 6, 15, 20, 15, 6, 1] = [1, 8, 28, 56, 70, 56, 28, 8, 1] \quad (11)$$

$$[1, 2, 1] * [-1, -4, -5, 0, 5, 4, 1] = [-1, -6, -14, -14, 0, 14, 14, 6, 1] \quad (12)$$

De hecho, se puede comprobar que `getDerivKernels` produce los mismos valores para tamaños 5 y 7 tanto en máscaras de alisamiento como de derivada, esto se debe a que la definición del filtro de Sobel 3×3 son dos vectores $[1, 2, 1]$ y $[-1, 0, 1]$, los cuales son la aproximación binomial de la gaussiana y un filtro de derivadas, por lo tanto, OpenCV para generar máscaras de mayor tamaño, realiza un proceso similar al descrito anteriormente, convolucionando la máscara binomial consigo misma y luego con la máscara de derivadas para obtener filtros de Sobel de mayor tamaño, que además soporta lo comentado en el ejercicio anterior del porqué son similares las máscaras: son

diferentes aproximaciones a la gaussiana, utilizando métodos diferentes, además confirma la idea teórica de que la convolución de una máscara gaussiana o approximando a una gaussiana con otra máscara gaussiana da como resultado una máscara también gaussiana de mayor tamaño, en cambio si se realiza con una de derivada, se obtiene una máscara de derivada ampliada.

Como último punto, el filtro de Scharr es calculado cuando se especifica a la función el parámetro `ksize=cv.FILTER_SCHARR` y produce máscaras de tamaño 3 que no se asemejan a la aproximación binomial de la gaussiana.

2.3. Implementación de convolución 2D separable

Para este ejercicio se desarrolló la función `convolveImage(img, xMask, yMask, borderType)` donde `img` es la imagen que se desea convolucionar, `xMask` es la máscara horizontal, `yMask` es la máscara vertical con las que se convolucionará la imagen y `borderType` es el tipo del borde para el padding.

Aunque no se ha especificado de ello, se implementó la función para que pudiera soportar máscaras de distintos tamaños, en parte por un posible uso futuro y también por comodidad del autor al momento de implementar la función.

Al entrar en la función, lo primero que se realiza es obtener el tamaño de la imagen en general, las longitudes de las máscaras y el punto medio –que es equivalente a la longitud de un lado de la máscara–.

Se piensa que el padding y la convolución se encuentran muy relacionados, por lo tanto tiene sentido que la propia función lo añada, de hecho, las funciones de OpenCV tienen siempre un padding aplicado por defecto aunque no se indique explícitamente: esta función añade el padding en dos pasos, antes de cada convolución se añade el padding por el lado de la imagen que se convolucionará con la función de OpenCV `copyMakeBorder()`.

Las máscaras, que inicialmente se proveen como un vector fila, de la manera en que se optimiza el cálculo, se convierte en un vector columna y se repiten los valores por tantas filas o columnas tenga la imagen original por medio de las operaciones de Numpy `np.reshape()` y `np.repeat()`, esto claro dependiendo si se trata de la máscara horizontal o vertical.

Por ejemplo, si se tiene una máscara, sea horizontal o vertical, de la forma $\begin{bmatrix} 1 & 2 & 1 \end{bmatrix}$ se transforma a $\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}$ y luego se repite n veces siendo n la longitud de un lado de la imagen con el cual se va a convolucionar la máscara:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 2 & 2 & 2 & 2 & \dots & 2 \\ 1 & 1 & 1 & 1 & \dots & 1 \end{bmatrix}$$

Dicho esto, luego se construye de una vez la imagen intermedia, nombrada `tempImg` en el código que almacenará el resultado de la primera convolución, es en esencia al principio una copia de la imagen original la cual tendrá sus valores de la imagen original sobreescritos, se realizó así por conveniencia.

Como se mencionó anteriormente el padding se añade ahora y como primero se convoluciona verticalmente, se añade el padding arriba y abajo de la imagen original.

La convolución se compone de un único bucle Python el cual avanza por un lado de la imagen, como ya se ha mencionado se realiza primero verticalmente, es decir, fila a fila de la imagen original de arriba a abajo.

Dentro de este bucle, la máscara encaja sobre la imagen centrada en una cierta fila, allí se realiza la multiplicación la imagen con la máscara de modo que en cada celda se obtiene la multiplicación de esa celda de la imagen con la celda de la máscara que le corresponde, y luego se suman los valores por columnas –es esencialmente como se haría una convolución más teórica solo que ampliada para hacer toda una fila en una sola operación– y se obtiene una dicha fila convolucionada de la imagen con la máscara la cual se almacena en la imagen temporal, el proceso continúa hasta recorrer la imagen entera. Naturalmente en las primeras y ultimas filas de la imagen original, la máscara tendrá valores fuera de rango y allí es donde entra el padding añadido anteriormente.

Una vez ha terminado la primera convolución la imagen se transpone para repetir el mismo proceso pero ahora “horizontalmente” y se añade el padding del lado restante: lo bueno de realizarlo así es que el padding tendrá la versión convolucionada de la imagen, por lo que se evitan artefactos en la imagen final.

Finalmente se repite el proceso con otro bucle que funciona exactamente igual aunque realice la convolución horizontalmente, solo que esta vez da como resultado la imagen convolucionada en ambas direcciones.

Se realizó la convolución 2D como dos convoluciones de máscaras 1D con la imagen pues esto es mucho más eficiente a nivel teórico y práctico a niveles de complejidad, se sabe por teoría que una convolución de una imagen por un kernel 2D tiene complejidad $O(m^2n^2)$ con n el tamaño del lado una imagen y m el tamaño del kernel, mientras que las convoluciones con un kernel 1D se reduce a $O(n^2m)$.

Se ha implementado una versión que funciona para imágenes a color, que se utiliza en la sección Bonus de la práctica, es exactamente el mismo procedimiento pero teniendo en cuenta que se tiene una tercera dimensión para los colores.

Una vez se realiza esto, la imagen se retorna de la función.

Esto que se está realizando es la versión discreta de la convolución tal y como se ha visto en teoría permite que, dadas dos señales o funciones, se obtenga una tercera señal la cual expresa cómo la forma de una señal es modificada por la otra, en este caso que las señales son imágenes es lo que sería aplicarle un filtro a una imagen: la imagen se ve modificada por la forma del filtro.

El equivalente en OpenCV de convolucionar una imagen con dos máscaras simétricas de alisamiento es la función `GaussianBlur(src, ksize, sigmaX)` la cual toma una imagen `src`, se indica un tamaño `ksize` de kernel, y `sigmaX` para indicarle el sigma de X y como no se especifica el sigma de Y, por defecto son iguales.

Imagen original



Implementación



OpenCV



Figura 4: Desenfoque Gaussiano ($\sigma = 2$), Comparativa

En la figura 4¹ se muestra un ejemplo de la imagen de una motocicleta, original, procesada por la función implementada y por la función de OpenCV utilizando el mismo sigma de 2, mismo tamaño de kernel, es decir, (13,13) y los bordes reflejados (`cv.BORDER_REFLECT`).

Visualmente se ven prácticamente idénticas las imágenes; para obtener una medida más objetiva se implementó la función auxiliar `getDiff()` que toma dos imágenes y le computa la diferencia como el promedio de la distancia euclídea entre cada píxel de una imagen y de otra.

$$diff(img_1, img_2) = \frac{1}{n} \cdot \sum_{i=0}^n \sqrt{img_1(i)^2 - img_2(i)^2} \quad (13)$$

Para este par de imágenes se obtiene una diferencia de $2,9413 \times 10^{-14}$, un resultado que indica que las dos convoluciones son prácticamente iguales, salvo por errores de redondeo.

Si por otro lado, se deja que OpenCV calcule el tamaño de la máscara dado el sigma, se obtiene una diferencia niveles de gris de 0,0227 entre las imágenes, indicando que OpenCV prefiere acelerar el cálculo de la convolución si se le permite pues ahora si existe una diferencia –que si bien también es imperceptible al ojo– a nivel de gris pero que la implementación de las máscaras gaussianas y la convolución son extremadamente similares a lo que se ha realizado en esta práctica.

Se han generado también imágenes derivadas utilizando las máscaras del ejercicio A para generarlas: para generar la derivada de X, se convolucionó la máscara de derivadas horizontalmente y de alisamiento verticalmente (`convolveImage(img, derivada, alisamiento)`), para la derivada de Y lo opuesto.

Se puede observar que en efecto se están derivando en la dirección X o Y por la dirección de las líneas: por ejemplo, en la derivada en X se puede ver la rueda de la moto pasa horizontalmente de un tono a otro de gris, y eso mismo sucede verticalmente en la derivada en Y.

Para obtener la magnitud se realizó $|\Delta I| = \sqrt{dx^2 + dy^2}$ con la funciones de Numpy, ver Figuras 5, 6, 7.

¹En la Figura 4 se muestra solamente la versión de OpenCV de mayor semejanza puesto que la otra versión es aunque objetivamente difiere por mucho, es también idéntica visualmente, pero en el código la imagen se genera y es comparada.

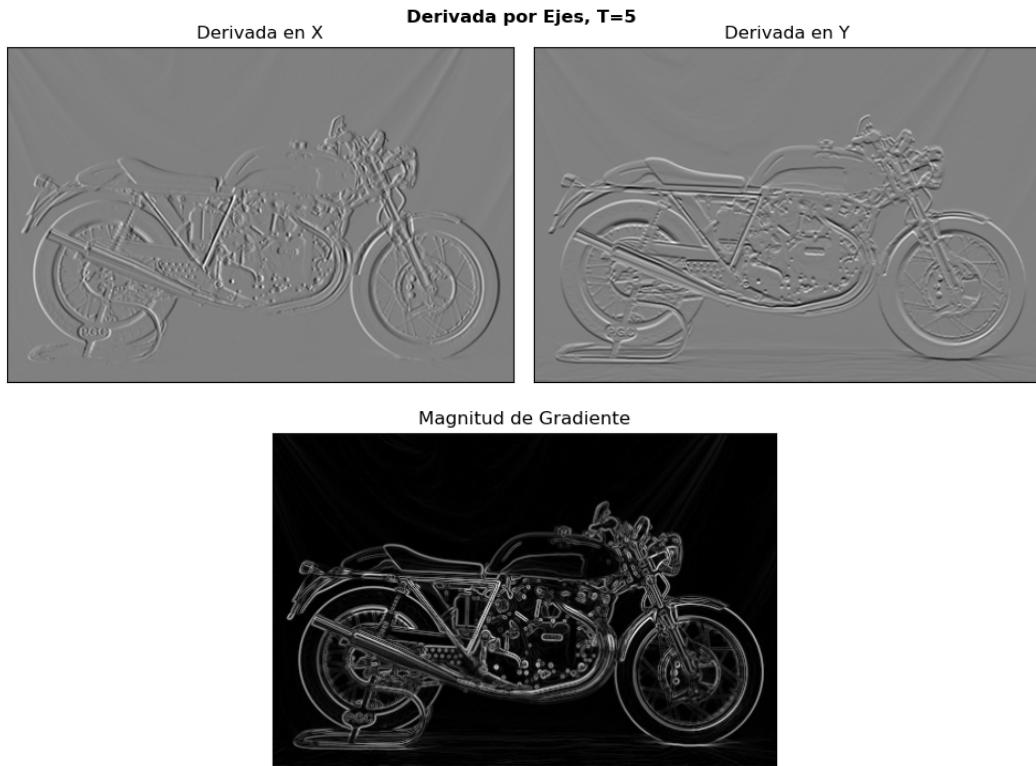


Figura 5: Derivadas de imagen con máscara de tamaño 5

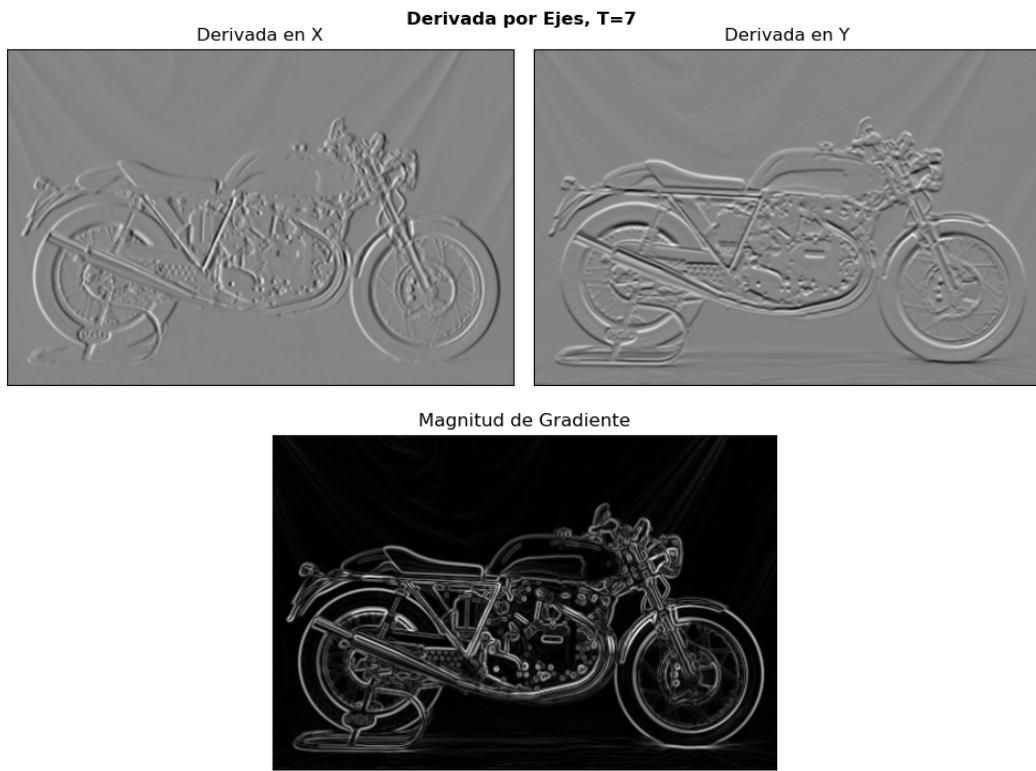


Figura 6: Derivadas de imagen con máscara de tamaño 7

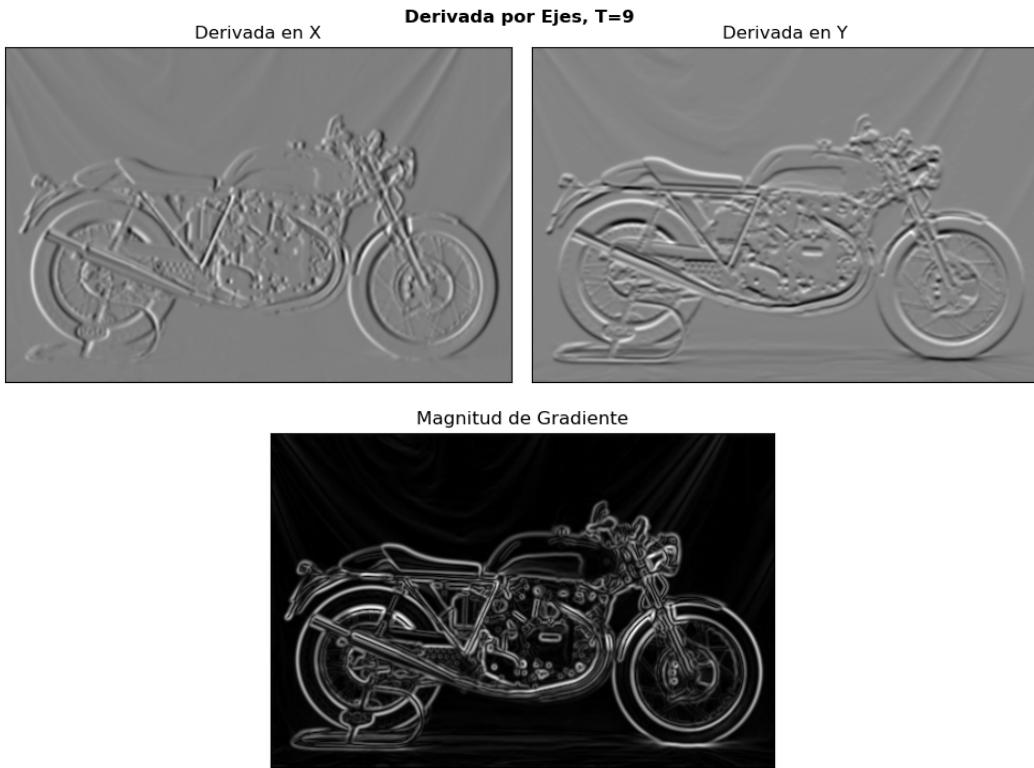


Figura 7: Derivadas de imagen con máscara de tamaño 9

2.4. Cálculo de Laplaciana de la Gaussiana

Para este ejercicio se implementaron dos funciones, `laplacianMask(sigma, maskSize)` y `laplacian(img, sigma)`.

La función `laplacianMask`, la cual calcula una máscara Laplaciana de Gaussiana y toma como parámetros el `sigma` que se desea utilizar para las máscaras gaussianas o bien la longitud de dicha máscara y en concreto se implementa la siguiente ecuación:

$$L = \sigma^2(G_{xx}(x, y, \sigma) + G_{yy}(x, y, \sigma)) \quad (14)$$

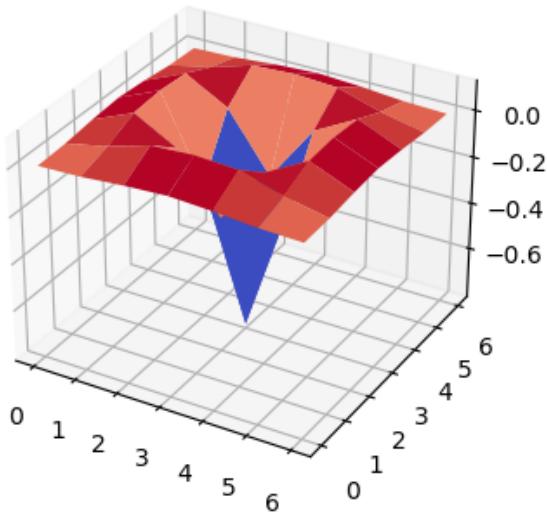
Se está aplicando el operador de Laplace, el cual se define como la suma de las segundas derivadas parciales, en este caso, se está aplicando sobre la función gaussiana.

Una máscara Laplaciana no es separable puesto que sus filas y columnas no son linealmente dependientes, aún así, se puede obtener por medio de la suma de convoluciones 2D que sí son separables puesto que la gaussiana y sus derivadas lo son.

Ya más concretamente, y como se pide en el ejercicio, se obtienen primero dos máscaras: una máscara gaussiana `gauss` y una de segunda derivada gaussiana `gdxx` por medio de la función `gaussianMask()` implementada anteriormente.

Para obtener las máscaras 2D que se utilizarán para obtener luego la Laplaciana se realiza el producto matricial entre las máscaras para obtener una máscara de las derivadas en dos direcciones, esto se realiza con la función de Numpy `np.outer(gauss, dxx)` para la matriz derivada por X y `np.outer(dxx, gauss)` para la Y.

Máscara Laplaciana ($\sigma=1$)



Máscara Laplaciana ($\sigma=3$)

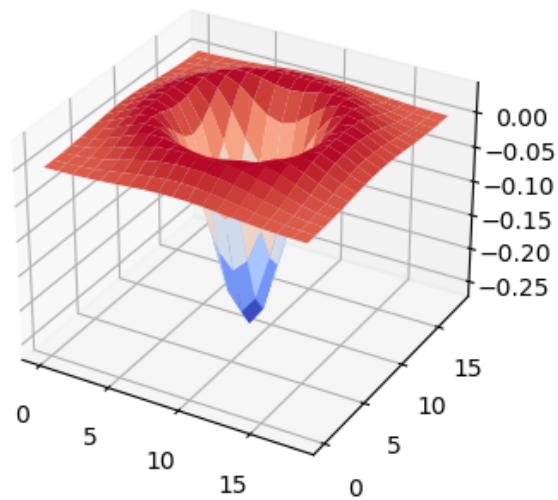


Figura 8: Representación de Máscaras 2D de LoG como superficies

Una vez se tienen estas dos máscaras, se suman y el resultado se multiplica por σ^2 –esto a nivel teórico se sabe que es para mantener los “zero-crossings” en los bordes detectados– para obtener la Laplaciana de la Gaussiana.

Estas máscaras se pueden observar en la Figura 8, que efectivamente generan el “sombrero mexicano invertido” reconocible de las derivadas segundas de la Gaussiana.

Ahora bien, utilizar directamente esta máscara sobre una imagen no es útil: como se sabe, no es separable y por lo tanto, la convolución sería mucho menos eficiente si se realiza de esta manera. Por suerte, también como se sabe, esta máscara fue generada por máscaras 1D gaussianas, por lo tanto, la manera para proceder es realizar las operaciones anteriormente descritas sobre una imagen.

Para ello se utiliza la función `laplacian(img, sigma, maskSize)`; en ella se generan las máscaras dependiendo del `sigma` o del tamaño de máscara al igual que la función anterior, pero ahora, a diferencia en vez de realizar el producto matricial de máscaras se realiza es la convolución para obtener la imagen derivada por filas y columnas con la función `convolveImage()`.

Como la convolución puede dejar valores positivos y negativos, se normalizan las imágenes antes de ser sumadas y multiplicadas por σ^2 .

El resultado de esta operación en una se puede observar en la Figuras 9, 10. Se compara con el resultado que se obtiene de la función de OpenCV `cv.Laplacian(src, ddepth, ksize)` para imagen `src`, con una profundidad de bits `ddepth = cv.CV_64F` para ser consistente de que se están trabajando con imágenes `float64` y un tamaño de máscara `ksize` equivalente al `sigma` que se está utilizando.

Existe una diferencia visual ligera entre ambas imágenes, aunque en general tanto para $\sigma = 1$ o $\sigma = 3$, el efecto en general del realce de bordes es el mismo pero si se puede notar que la

implementación de OpenCV obtiene unas imágenes con más diferencias entre grises y por lo tanto son un tanto más oscuras para el mismo sigma, e incluso no difuminan tanto la imagen para sigmas más altos.

En parte, la diferencia es que el operador Laplaciano de OpenCV se implementa con máscaras de Sobel, a diferencia de la implementación de esta práctica que realiza la derivada sobre una máscara Gaussiana y como se vió en los ejercicios anteriores, si bien estas máscaras son muy parecidas, no son exactamente la misma y se sabe que la implementación de OpenCV intenta siempre dar resultados más rápidos que de mejor calidad, por lo tanto es razonable que el resultado no sea el mismo.

¿Eso es bueno o es malo? Depende, si se desea tener más precisión, se quiere que la Laplaciana sea lo más precisa posible pues es mejor entonces utilizar una implementación que realmente realice el operador Laplaciano sobre una Gaussiana aunque esto pueda ser más costoso computacionalmente; si se desea más velocidad y esto puede ser muy deseable en aplicaciones en tiempo real entonces es mejor utilizar la implementación de OpenCV puesto que está optimizada sobre todo para ser más veloz que precisa.

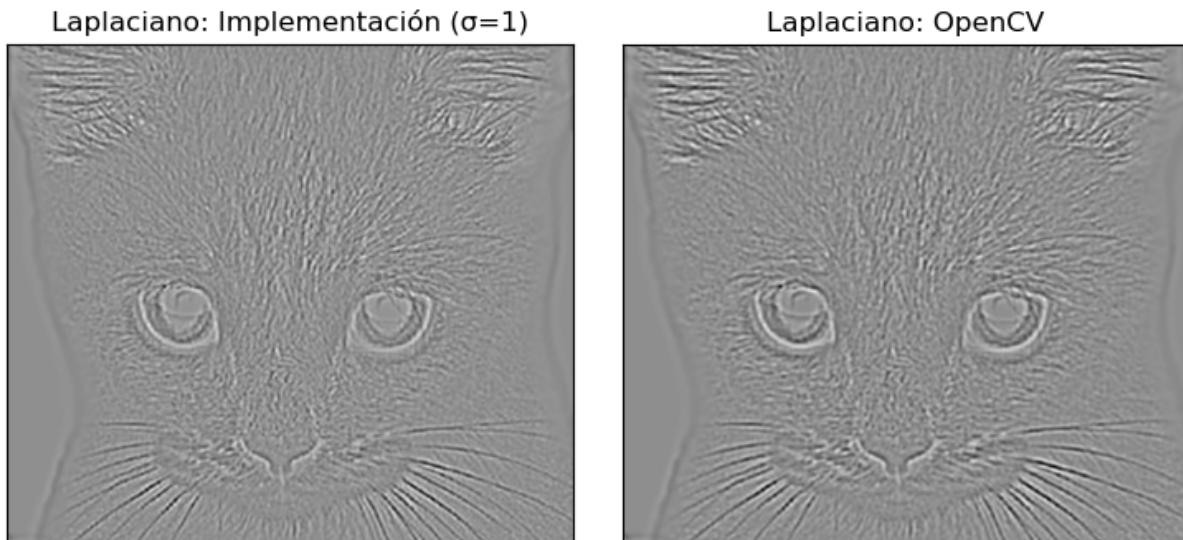


Figura 9: Operador Laplaciano, Comparativa para $\sigma = 1$

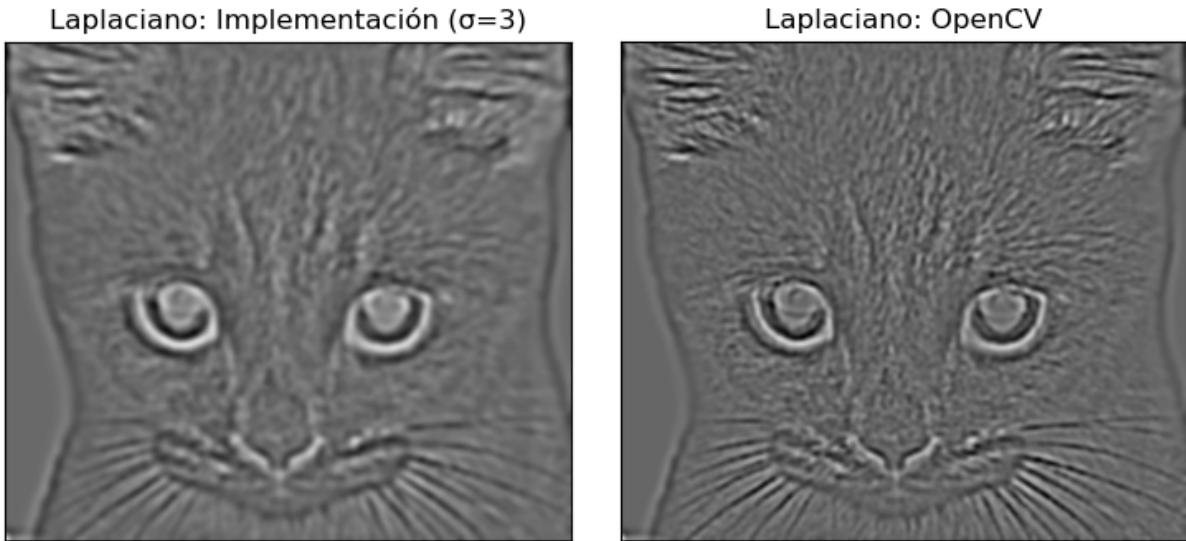


Figura 10: Operador Laplaciano, Comparativa para $\sigma = 3$

3. Implementación de Funciones Avanzadas

3.1. Generación de Pirámide Gaussiana

Para la generación de la pirámide, se implementó la función `gaussianPyramid(img, maxLevel, sigma, borderType)`, se realiza la pirámide gaussiana a una imagen hasta el nivel máximo especificado, la convolución con un sigma dado y un tipo de borde, el cual se pasa a la convolución.

La función es relativamente simple, se obtiene una máscara gaussiana utilizando el método previamente implementado `gaussianMask()` y se inserta la imagen original como la primera en una lista de imágenes.

Luego, se tiene un bucle el cual convoluciona la imagen pasada por parámetro con el filtro de alisamiento utilizando `convolveImage()` y luego la imagen se reduce de tamaño con una sencilla función auxiliar llamada `subSample()` que utiliza el indexado de Numpy para reducir a la mitad la imagen quitando filas y columnas pares. Una vez realizado esto, se añade la imagen generada a la lista y se repite el proceso anterior con esa misma imagen, así sucesivamente hasta llegar al nivel indicado.

Se implementó una función equivalente, `gaussianPyrCV(img, maxLevel, p_borderType)`, pero esta función hace uso del método de OpenCV llamado `pyrDown(src, borderType)` el cual toma una imagen y la reduce, por defecto, también a la mitad la imagen.

Para la elección del σ , los autores Burt y Adelson proponen un $\sigma = 2$ para una reducción del tamaño de imagen en 2, aunque proponen también una máscara gaussiana de tamaño 5 [1, 4, 6, 4, 1] que es exactamente lo que utiliza OpenCV internamente para producir la pirámide.

En el caso de la implementación, se probó con entonces con $\sigma = 2$ dada la recomendación de los autores, por otro lado, como la implementación de la máscara gaussiana solo permite definir un sigma o un tamaño de máscara se probó también utilizando $\sigma = 0,6$, generando la máscara tanto

con el sigma como un tamaño de 5 y finalmente por medio de pruebas se utilizó $\sigma = 1$ prefiriendo el último valor pues es el que más se asemeja a lo obtenido por OpenCV.

Implementación ($\sigma=1$)



OpenCV



Figura 11: Comparativas de Pirámides Gaussianas

Se implementó otro método auxiliar, `gaussCompare(pyr1, pyr2)` el cual genera la media de las diferencias en distancia euclídea para toda la pirámide usando `getDiff()` y en efecto la menor diferencia ocurre cuando $\sigma = 1$.

σ	Diferencia Euclídea Promedio	
0.6	7.2116	
0.6	5.67	T=5
1	0.3596	
2	9.8158	

Cuadro 1: Error cuadrado de pirámide gaussiana, implementación con valores σ vs OpenCV

Ahora, se presentan las pirámides (Figura 11) gaussianas generadas por estos métodos utilizando bordes reflejados, se muestran por motivos de espacio solamente la que posee el mejor valor de sigma y la de OpenCV, en la ejecución del programa se puede observar cada pirámide con los sigmas indicados y compararlas detalladamente (y con mejor calidad).

3.2. Generación de Pirámide Laplaciana

De manera similar al ejercicio anterior, se implementó la función `laplacianPyramid(p_gaussPyr)` que genera la pirámide Laplaciana partiendo de la pirámide Gaussiana, para ello, se toma la imagen más pequeña de la pirámide Gaussiana, lo que sería “el tope” de la pirámide y con ella, inicialmente, dentro de un bucle se realiza:

- Se expande la imagen hasta el tamaño de la imagen anterior en la pirámide con `cv.resize()` utilizando interpolación bilineal.
- Se resta la imagen anterior de la pirámide con la actual expandida.
- Se almacena esa imagen en la lista y se repite hasta tener los mismos niveles que la pirámide gaussiana tomando la imagen anterior como la imagen actual.
- Esto se repite hasta recorrer todos los niveles de la pirámide Gaussiana original.

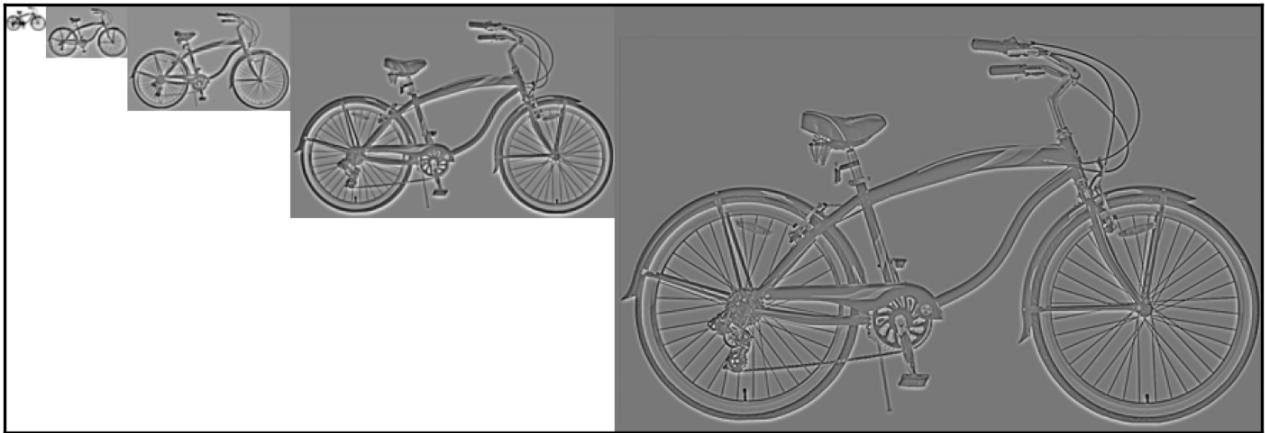
Finalmente, se devuelve una lista de imágenes que conforman la pirámide laplaciana.

Se crea también otra función llamada `laplacianPyrCV(p_gaussPyr)` la cual realiza el mismo cálculo pero hace uso de la función `cv.pyrUp` para incrementar el tamaño de la imagen, el resto del procedimiento se mantiene básicamente igual que en la otra función.

En la figura 12 se puede notar la ligera diferencia en implementaciones, aún utilizando la misma pirámide de origen.

La diferencia recae en la implementación de OpenCV de la función, pues no utiliza como tal filtrado bilineal, sino que inyecta filas y columnas de ceros y luego convoluciona eso con una máscara de alisamiento equivalente a la que utiliza en `pyrDown()` multiplicada por 4; ya que el resto del proceso es exactamente igual en las funciones implementadas.

Implementación



OpenCV

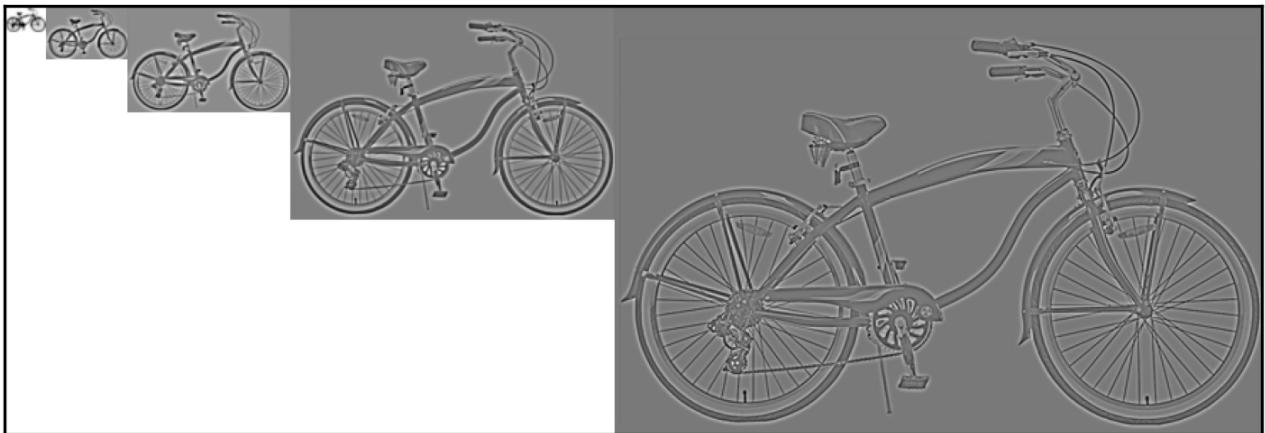


Figura 12: Pirámide Laplaciana: OpenCV vs Implementación

3.3. Recuperación de imagen por medio de Pirámide Laplaciana

Para reconstruir una imagen dada la pirámide laplaciana se implementó la función `recoverImg(lapPyr)` que toma como único parámetro el vector que contiene la pirámide laplaciana, el procedimiento para reconstruir la imagen comienza con obtener la imagen base de la Laplaciana, esto es, la imagen más pequeña de todas y el tope de la pirámide gaussiana y en un bucle:

- Se expande la imagen actual al tamaño de la imagen siguiente –de mayor tamaño– en la pirámide con interpolación bilineal usando el método `cv.resize()`.
- Se suma la imagen expandida con la imagen siguiente de mayor tamaño.
- Esta imagen resultante será la imagen actual de la siguiente iteración.
- Se repite hasta que se recorre toda la pirámide laplaciana.

El bucle finaliza cuando se recorre toda la pirámide y se devuelve la imagen reconstruida, se puede

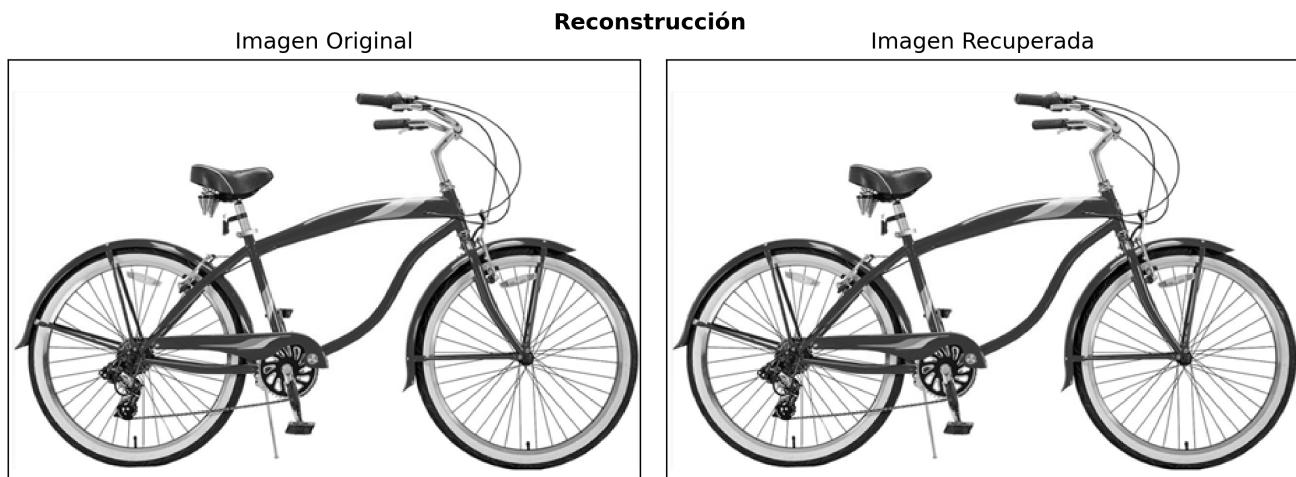


Figura 13: Reconstrucción de una imagen por la pirámide Laplaciana

ver ahora una comparativa visual entre las imágenes de la bicicleta que se han estado usando en estos ejercicios en la Figura 13.

Visualmente es idéntica, no solamente eso, al obtener la diferencia de distancia euclídea promedio se obtiene un error de $4,6533 \times 10^{-17}$ que, considerando los errores de redondeo, se puede decir que la imagen reconstruida es exactamente la misma a la original, esto soporta la teoría, en donde se sabe que al desenfocar existen píxeles redundantes por lo tanto al reducir el tamaño quitando filas y columnas la información se mantiene, además que los detalles también se encuentran en las diferencias entre las imágenes a diferentes escalas –lo que es la pirámide laplaciana–, de allí puede reconstruirse nuevamente la imagen.

4. Bonus

4.1. Generación de imágenes híbridas en escala de grises

Se implementó la función `genHybridImg(highF, hFSigma, lowF, lFSigma)` que toma dos imágenes, una para ser de alta frecuencia y otra de baja frecuencia, junto con los sigmas respectivos.

La generación de la imagen de alta frecuencia se realiza por la Laplaciana –con el sigma por parámetro–, puesto que la misma es un filtro de paso alto y por lo tanto realza los bordes de la imagen como bien se ha visto anteriormente; por medio de pruebas se encontró que para mejorar más el efecto de imagen híbrida se debe de invertir los niveles de gris, esto se traduce a multiplicar todos los valores de la imagen por -1 .

La generación de la imagen de baja frecuencia se realizó con una máscara gaussiana de sigma especificado por parámetro y se convolucionó esta máscara con la imagen.

Una vez que se tienen ambas imágenes, antes de unirlas se normalizan pues pueden contener valores negativos, especialmente la Laplaciana y el no realizar esto evita que se produzca el efecto.

Finalmente, las dos imágenes se suman y se obtiene la imagen híbrida, luego la función genera un diccionario para almacenar las dos imágenes base junto con la imagen híbrida y la función devuelve

ese diccionario junto con la imagen híbrida sola.

Se obtuvieron tres ejemplos de imágenes híbridas, las Figuras 14, 15, 16, el proceso de obtención de los sigmas fue basado en prueba y error con orientación del artículo científico “*Hybrid Images*” de Oliva, Torralba y Schyns; curiosamente el valor de los mismos se mantuvo entre unos rangos, al menos para las imágenes seleccionadas.

El sigma de las imágenes de alta frecuencia se encontró daba el mejor efecto entre $1 \leq \sigma_{HI} \leq 2$ y el sigma de las imágenes de baja frecuencia $\sigma_{LO} = 9$.

En cada una de las imágenes se puede observar el efecto buscado: la imagen más grande, que también puede interpretarse como la imagen “más cerca” del observado, domina la imagen de alta frecuencia sobre la de baja frecuencia, es decir, uno se ve más centrado en los bordes de la imagen que en su color y mientras la imagen se hace más pequeña –“se aleja”–, empieza a tomar dominancia la imagen de baja frecuencia, los bordes son menos notables y lo que quedan son las formas difuminadas las cuales son interpretadas como otro objeto.

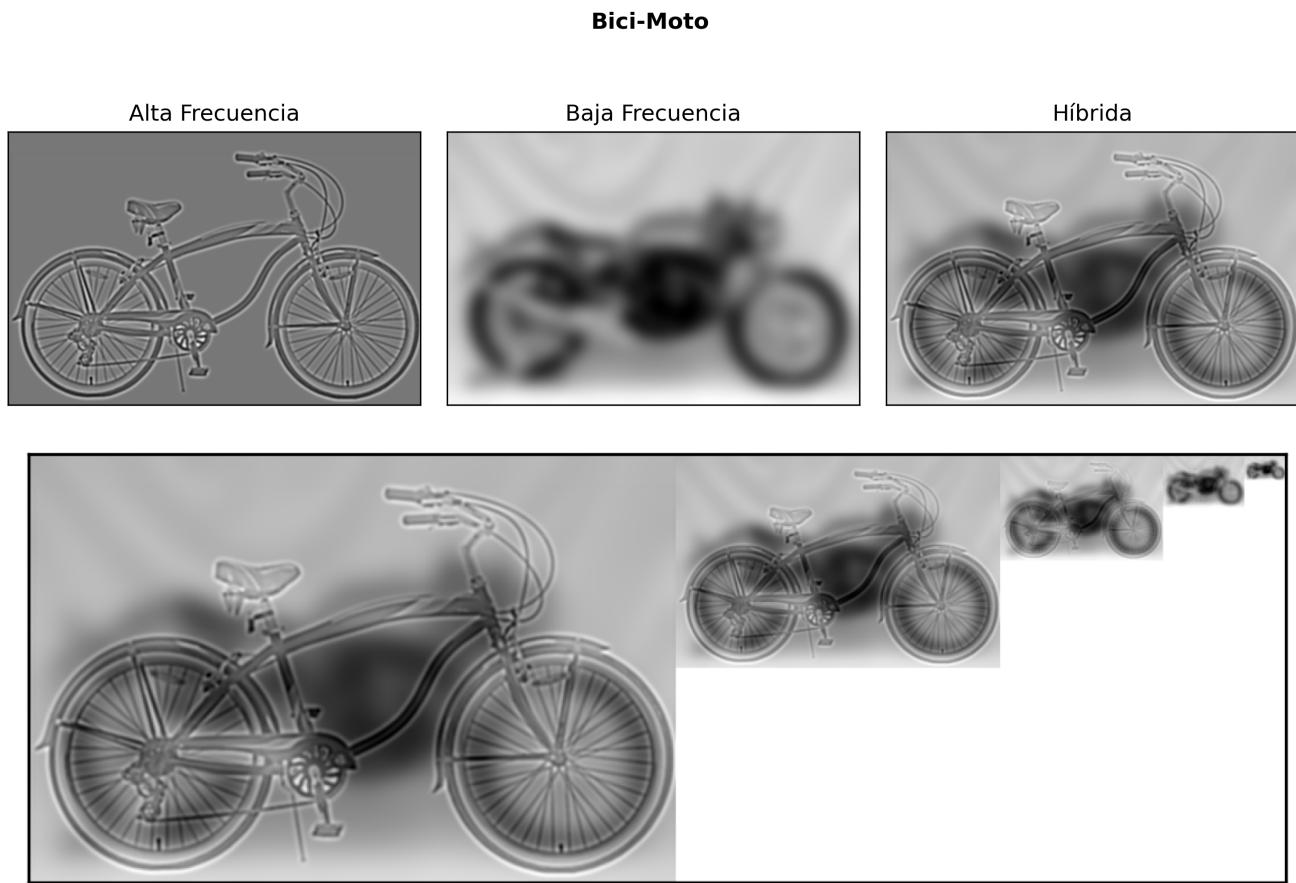


Figura 14: Imagen Híbrida entre una bicicleta y una moto ($\sigma_{HI} = 1,5; \sigma_{LO} = 9$)

Pájaro-Avión

Alta Frecuencia



Baja Frecuencia



Híbrida

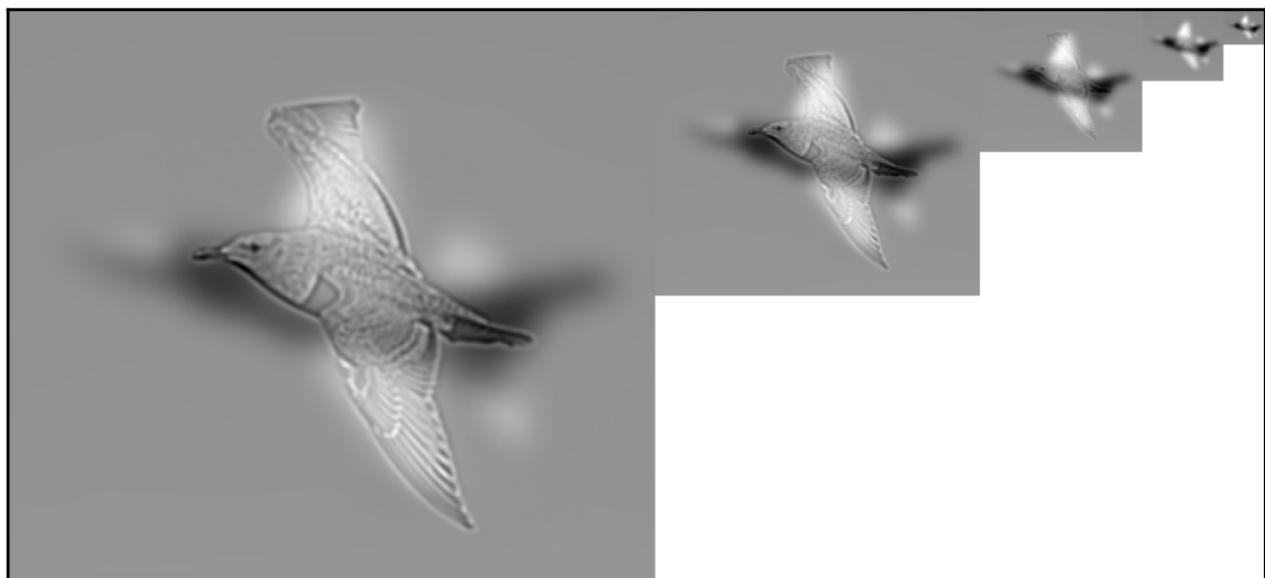


Figura 15: Imagen Híbrida entre un pájaro y un avión caza ($\sigma_{HI} = 1,5$; $\sigma_{LO} = 9$)

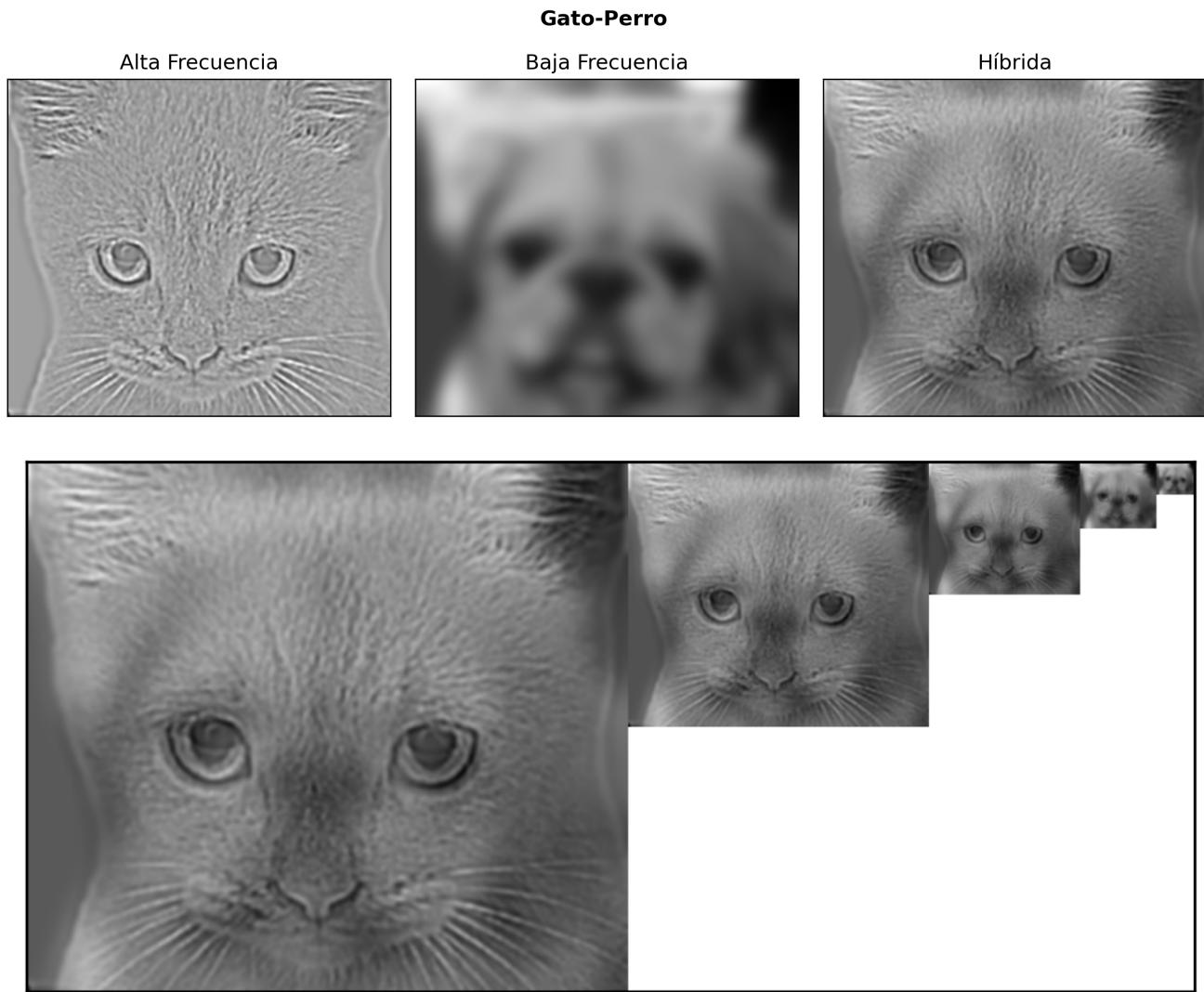


Figura 16: Imagen Híbrida entre un gato y un perro ($\sigma_{HI} = 2$; $\sigma_{LO} = 9$)

4.2. Generación de imágenes híbridas a color

El proceso fue exactamente similar al de generar imágenes híbridas a escala de grises, de hecho, utilizan la misma función `genHybridImg()` comentada en el ejercicio anterior: el cambio más grande ocurrió en la función `convolveImage()`, donde se tiene ahora un condicional para detectar si la imagen pasada es a color o en escala de grises.

Las mayores diferencias en el código es la manera en que se repite la máscara, pues ahora se tiene que repetir en un nuevo eje, por eso cuando se utiliza `np.repeat()` se indica `np.newaxis` con una longitud de 3.

La convolución es idéntica solo que se añade la tercera columna en el indexado y para realizar la transposición, se utiliza la función `np.swapaxes()` para intercambiar los ejes X e Y en vez de `np.transpose()` pues esta última daba problemas con matrices de más de dos dimensiones.

Se muestran a continuación las imágenes híbridas anteriores a color en las Figuras 17, 18 y 19;

todas utilizan los mismos valores de sigma que sus versiones en escala de grises.

Moto-Bicicleta

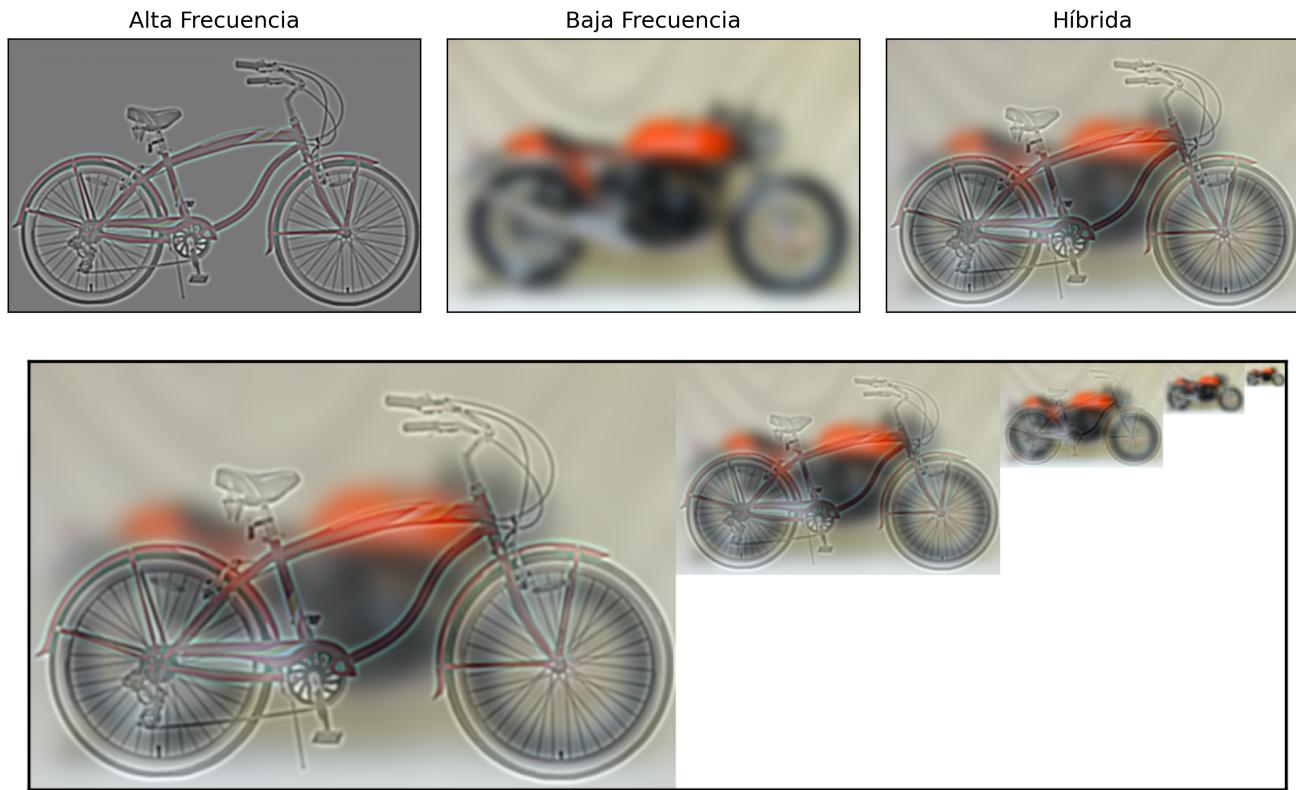


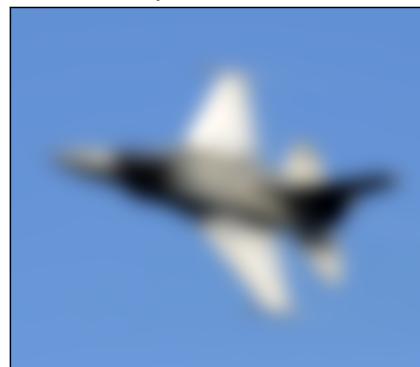
Figura 17: Imagen Híbrida entre una bicicleta y una moto a color.

Pájaro-Avión

Alta Frecuencia



Baja Frecuencia



Híbrida

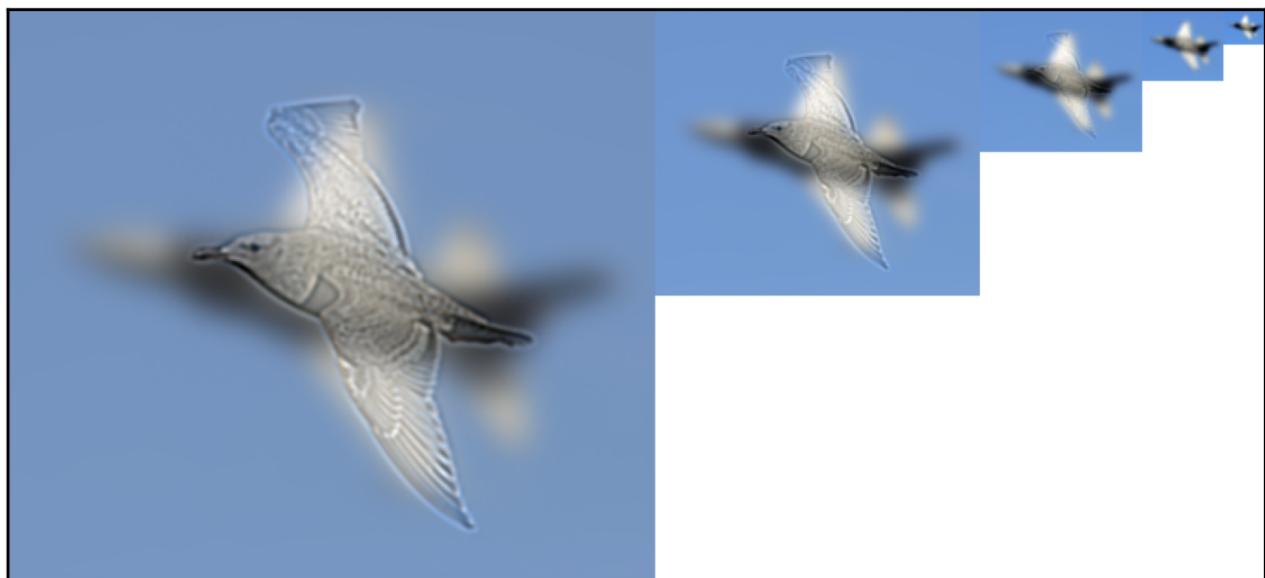


Figura 18: Imagen Híbrida entre un pájaro y un avión caza a color

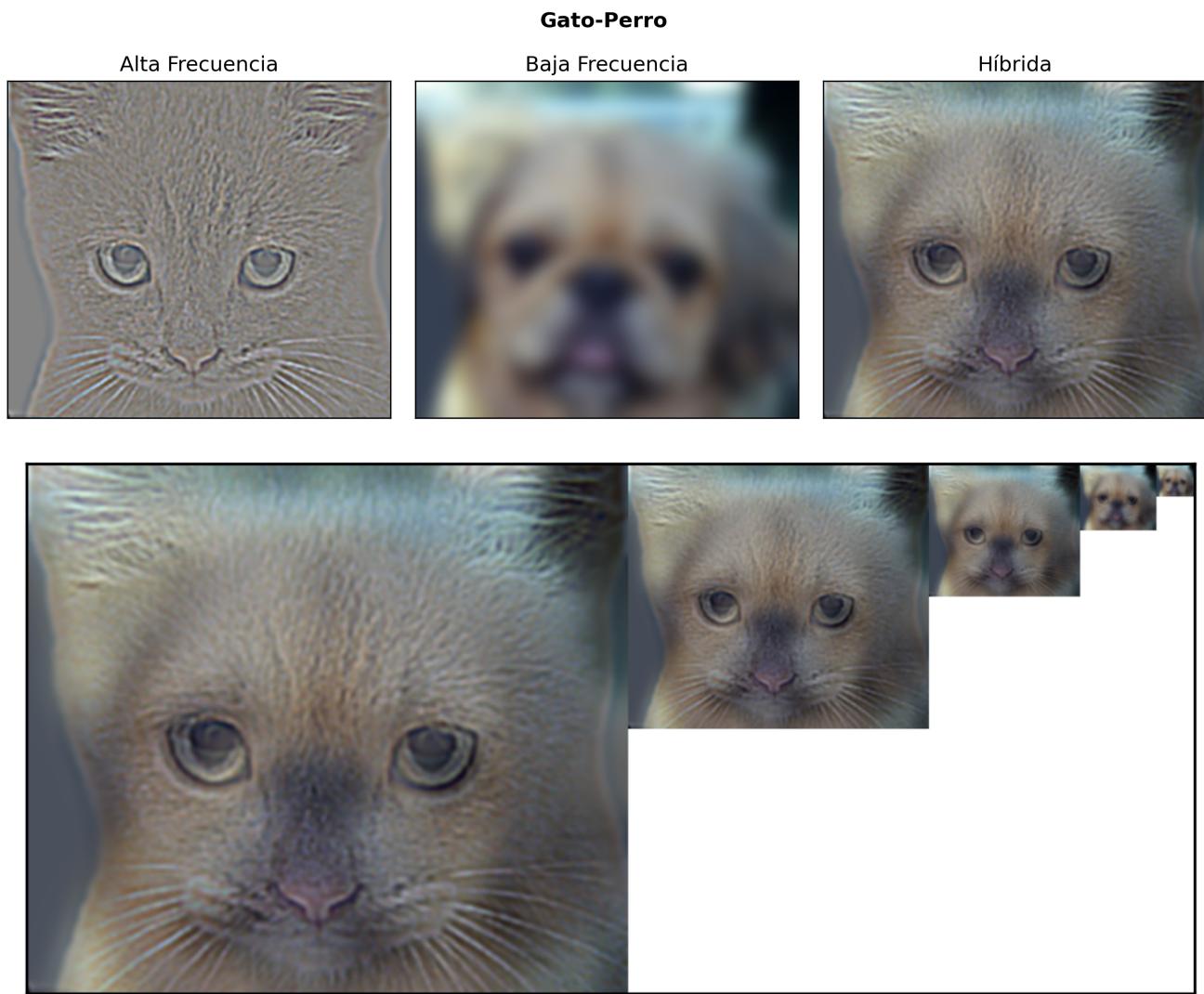


Figura 19: Imagen Híbrida entre un gato y un perro a color

4.3. Imágenes híbridas a elección libre

La selección de las imágenes se realizó siguiendo las recomendaciones del artículo científico, es decir, las imágenes de alta frecuencia deberían ser con bordes más prominentes mientras que las imágenes de baja frecuencia deberían ser aquellas con formas más suaves.

Se decidió seleccionar dos imágenes de automóviles, puesto que los coches tienen por lo general una forma parecida pero al mismo tiempo, se tienen coches que poseen muchos ángulos rectos –sobre todo coches antiguos– mientras que los modernos poseen curvas más suaves para ser más aerodinámicos.

Adicionalmente, una manera de alinearlos es alinear las ruedas, de esa manera es más fácil lograr el efecto pues el ojo no notará que hay manchas redondas donde en la imagen de alta frecuencia no estarían las ruedas.

Ahora bien, se decidió utilizar como imagen de alta frecuencia el Ford Modelo T ya que al ser uno

Pasado a Futuro

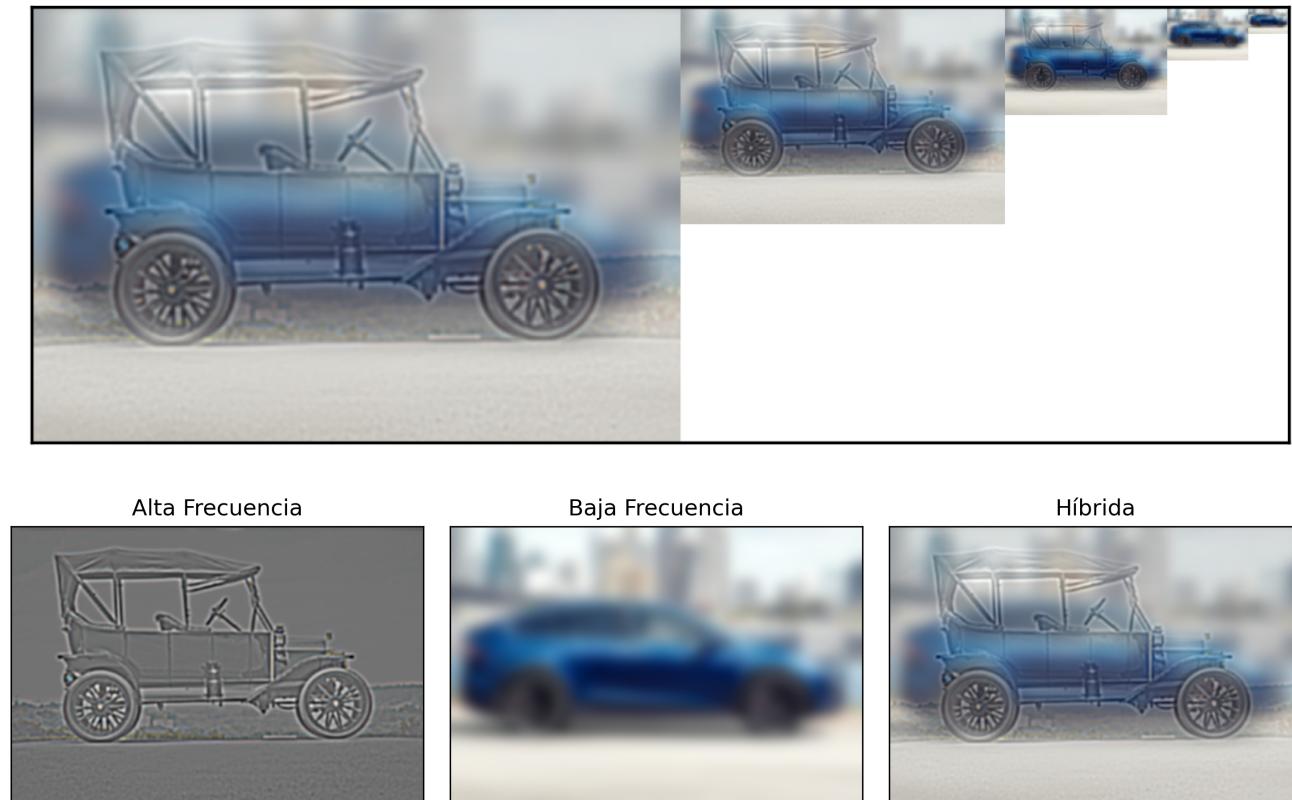


Figura 20: Imagen híbrida de elección libre

de los modelos más antiguos de coches posee una forma menos aerodinámica, con muchos más ángulos rectos la cual hace que resalten más esos bordes cuando la imagen se mira de cerca.

Para la imagen de baja frecuencia se eligió una foto en el mismo ángulo de un Tesla Modelo X que posee una forma curva perfecta para que no interfiera con los ángulos rectos del Modelo T.

Ambas imágenes fueron obtenidas de páginas de fondos de pantalla: PeaxPk para el Tesla y 1Zoom para el Ford. Posteriormente fueron editadas utilizando Photoshop para, en primer lugar escalarlas a una resolución más razonable, para que el lado más largo fuese de 512 píxeles y además se recortaron, movieron y se orientaron de manera que estuviesen alineadas las imágenes para que el efecto fuese más convincente.

Una vez realizado esto, las imágenes son cargadas por el software de la práctica y se convolucionan exactamente igual que el resto de las imágenes aquí presentes haciendo uso de las funciones previamente implementadas. (Ver Figura 20).