

# Práctica 1

Visión por Computador:

Filtrado y Muestreo

Óscar David López Arcos, 75571640-B  
[odlarcos@correo.ugr.es](mailto:odlarcos@correo.ugr.es)  
Grupo: Miércoles 11:30-13:30

# **Índice**

Funciones genéricas	3
Ejercicio 1	3
Ejercicio 2	5
Ejercicio 3	7
BONUS	9

## Funciones genéricas

**pintaMismoMarco(vim, TITULO):** Concatena las imágenes incluidas en la lista vim para mostrarlas en un mismo marco bajo el título indicado. Si una imagen es más pequeña que la anterior se le incluyen filas negras para igualar ambos tamaños.

**leeImagen(filename, flagColor):** Devuelve la imagen leída desde el archivo “filename”. Si flagColor=0 la imagen se leerá en escala de grises, y si flagColor=1, en RGB.

**showImagenWait( imagen, nombre ):** Muestra la imagen y espera la pulsación de una tecla para continuar.

**concatenacionTitulos(nfil, ncol, imagenes, titulos):** Muestra en la consola las imágenes con sus respectivos en una cuadrícula de máximo 9x9, haciendo uso de la clase pyplot.

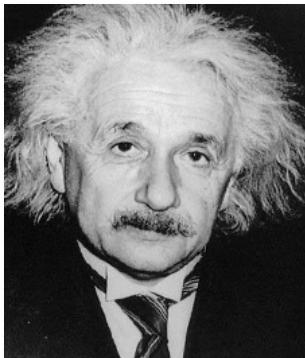
## Ejercicio 1

### a) Cálculo de la convolución de una imagen con una máscara Gaussiana 2D.

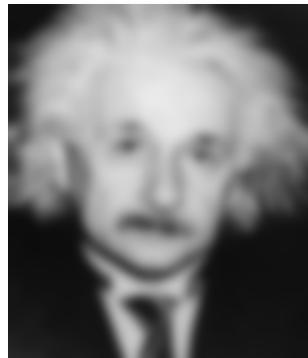
La función **GaussianBlur** acepta como parámetros ksize como un sigma. Sin embargo, si fijamos ksize a 0, éste será calculado a partir del sigma según la fórmula estudiada en clase ( $ksize = 2 [3\sigma] + 1$ ) para que la distribución sea lo más óptima posible.

A mayor valor de sigma, mayor tamaño de la máscara y, por tanto, más píxeles vecinos influirán en el cálculo del suavizado. Como consecuencia, los píxeles se irán pareciendo más a sus vecinos y la imagen quedará más borrosa.

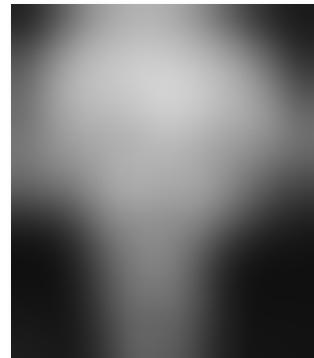
Esto mismo puede comprobarse con los ejemplos utilizados en este ejercicio:



Original



$\sigma = 4$



$\sigma = 20$

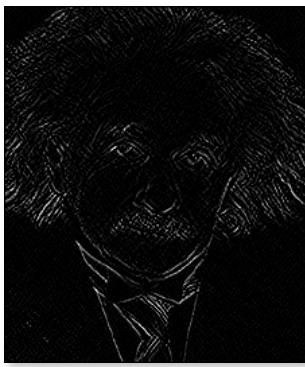
### b) Obtención de máscaras 1D para calcular la convolución 3D con máscaras de derivadas

Las máscaras de derivadas permiten detectar las frecuencias altas en las imágenes (zonas donde hay fronteras o saltos). Conforme aumentamos las derivadas, menos marcados serán los bordes.

El parámetro ksize de la misma función **getDerivKernels** juega también un papel importante: con uno pequeño se detectan mejor características, y con uno alto detecta los bordes a mayor escala. Es decir, uno alto será más sensible a los cambios de frecuencia.

Al aplicar los kernels obtenidos a la imagen (**sepFilter2D**) fijamos la profundidad a -1 para que la imagen generada tenga igual profundidad a la original.

En el último ejemplo puede verse claramente como al no aplicar ninguna derivada en el eje X, los cambios de frecuencias sólo se detectan en el eje Y.



$\delta x = 1, \delta y = 1, ksize = 3$



$\delta x = 1, \delta y = 1, ksize = 5$



$\delta x = 2, \delta y = 1, ksize = 5$



$\delta x = 0, \delta y = 1, ksize = 3$

c) Usar la función **Laplacian** para el cálculo de la convolución 2D con una máscara Laplaciana-de-Gaussiana de tamaño variable.

La función Laplaciana es la segunda derivada de la Gaussiana y se aplica tanto en x como en y, de esta forma es capaz de detectar los bordes (contrastos) sea cual sea la dirección de estos. Suele aplicarse a una imagen suavizada por un Gaussiano, para reducir de este modo su alta sensibilidad al ruido.

He utilizado dos tipos de bordes, para ello he definido la función **addBorders** que a su vez llama a la función de openCV **copyMakeBorder**. Los dos bordes utilizados han sido bordes constantes (BORDER\_CONSTANT) y replicados (BORDER\_REPLICATE). Los bordes no supondrán ningún cambio en el resultado final de la imagen. (Nota: Aunque el tamaño lógico de los bordes sería KSIZE/2, en esta práctica se han fijado a 10 para una mejor visualización)

El sigma aplicado al filtro Gaussiano tiene ahora más relevancia como puede comprobarse en los ejemplos. Un sigma más pequeño dará como resultado un filtro más sensible al ruido que resaltará bordes con menor tolerancia.



$\sigma = 1$



$\sigma = 3$



$\sigma = 1, B = \text{Constant}$



$\sigma = 1$  , **B** = Constant



$\sigma = 1$  , **B** = Replicate



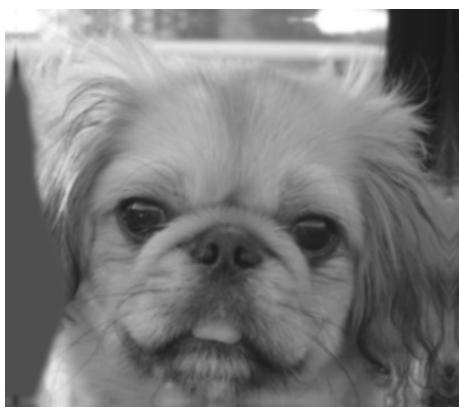
$\sigma = 1$  , **B** = Replicate

## Ejercicio 2

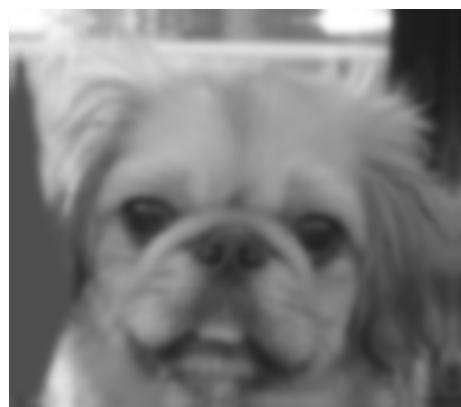
- a) **Cálculo de la convolución 2D con una máscara separable de tamaño variable. Usar bordes reflejados.**

Para el uso de los bordes reflejados, usaremos la misma función ya definida **copyMakeBorder**, esta vez con el parámetro **BORDER\_REFLECT**.

Para la obtención del kernel, la función **getGaussianKernel** nos devolverá uno simétrico que podremos emplear en **sepFilter2D**. Si leemos la documentación, vemos que en este caso (a diferencia de GaussianBlur) es el parámetro sigma el que se calculará a partir de ksize si lo inicializamos a negativo.



$\sigma = 5$



$\sigma = 20$

Como resultado, a mayor valor de ksize mayor suavizado, como explicamos en el apartado 1a.

- b) **Cálculo de la convolución 2D con una máscara 2D de 1<sup>a</sup> derivada de tamaño variable. Usar bordes a cero.**

Del mismo modo que en el apartado 1b, la función que utilizaremos será **getDerivKernels**, con parámetros muy similares aplicando únicamente la primera derivada. Para insertar bordes a 0 volvemos a utilizar la opción **BORDER\_CONSTANT**, esta vez con valor 0.

En estos ejemplos se ve cómo afecta el ksize al resultado final de la imagen, siendo más sensible los cambios de frecuencias conforme aumenta.



$$\sigma = 3$$



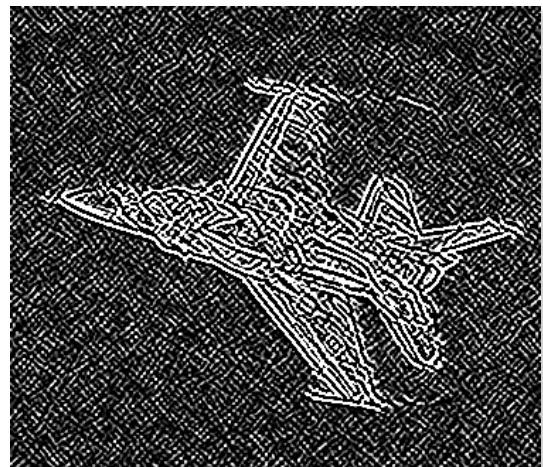
$$\sigma = 9$$

c) **Cálculo de la convolución 2D con una máscara 2D de 2<sup>a</sup> derivada de tamaño variable.**

Aplicamos la segunda derivada del mismo modo.



$$\sigma = 3$$



$$\sigma = 9$$

d) **Pirámide Gaussiana**

Para la pirámide Gaussiana he definido la función **piramideGaussiana(im, tam)**, que hace uso de la función **pyrDown** para reducir la imagen, previamente suavizada por un GaussianBlur. Este proceso se repetirá sobre la imagen generada tantas veces como niveles queramos en nuestra pirámide, de forma que en cada reducción se pierda el mínimo de información posible (al estar repartida entre los píxeles vecinos).

e) **Pirámide Laplaciana**

Del mismo modo, la pirámide Laplaciana (**piramideLaplaciana(im, tam)**) hace previo uso de un suavizado y pyrDown. En este caso, sin embargo, a la imagen generada se le aplicará un **pyrUp** y lo que se almacenará en un vector será resta entre la original y esta última. Así, únicamente guardaremos las altas frecuencias, que nos servirán para reconstruir la imagen en mayor calidad desde la última fase de la pirámide Laplaciana (es decir, desde el pyrDown del último nivel).

Para este ejercicio ha sido necesario implementar la función **ajustar**, que se asegura de trabajar con imágenes con dimensiones pares debido a problemas de truncamiento en los pyrDown.

## Pirámide Gaussiana



## Pirámide Laplaciana



## Ejercicio 3

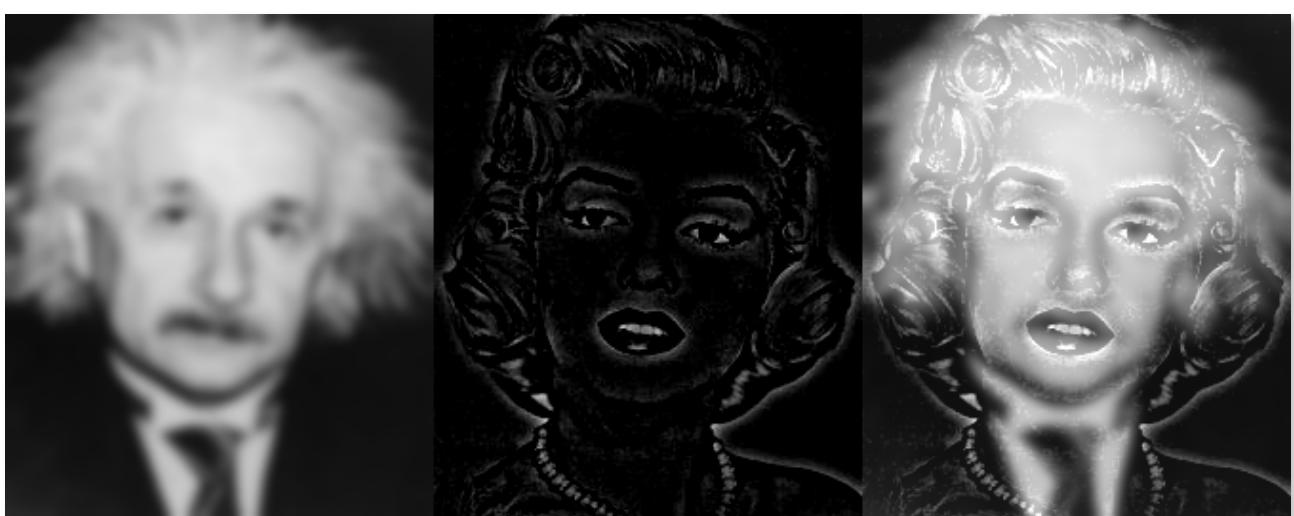
### Imágenes híbridas

Para poder realizar este ejercicio, en primer lugar, necesitamos parejas de imágenes donde los elementos a superponer aparezcan en posiciones similares. Una vez conseguido esto, aplicaremos a una imagen un filtro de paso alto y a la otra un filtro de paso bajo. El ojo humano tiene la particularidad de apreciar mejor las frecuencias altas desde cerca y las frecuencias bajas desde lejos, lo que hará posible que este efecto funcione.

Para el filtro de paso bajo será tan sencillo como aplicar un filtro Gaussiano de los vistos anteriormente, ya que tienen la particularidad de eliminar las altas frecuencias. Por otro lado, para el filtro de paso alto, necesitaremos restar a la imagen original la misma imagen suavizada, quedándonos así únicamente con las frecuencias más elevadas.

Por último, para mezclar ambas imágenes sólo tendremos que sumarlas. Es conveniente utilizar los operadores sobrecargados de openCV para todas las operaciones (**add**, **subtract**, **multiply**) para asegurarnos que trabajamos manteniendo la coherencia.

Tras ciertos ajustes en los sigmas y algún que otro parámetro que ayude a enfatizar determinadas frecuencias (**paramSharp**) los resultados han sido estos.



## BONUS

### 1- Cálculo del vector máscara Gaussiano.

Para este ejercicio, hemos realizado a mano la labor que ejerce la función **getGaussianKernel**, estableciendo el tamaño de la máscara al más óptimo según el sigma y calculando los diferentes valores evaluando la función según proximidad al centro. Por último, normalizamos para que la suma de los valores del núcleo sea igual a 1. (Ver código **mascaraConvolucion1D**)

### 2- Implementar una función que calcula la convolución 1D de un vector señal

El primer paso para abordar este ejercicio es saber si estamos trabajando con imágenes grises o RGB. Para ello comprobamos el tipo del primer dato: si es una lista será RGB ([r,g,b]) y, por el contrario, si es un entero será escala de grises. Además, en caso de RGB separará cada canal (**split**) y los volverá a unir al finalizar el proceso (**merge**). (Función **convolucionVectorMascara**)

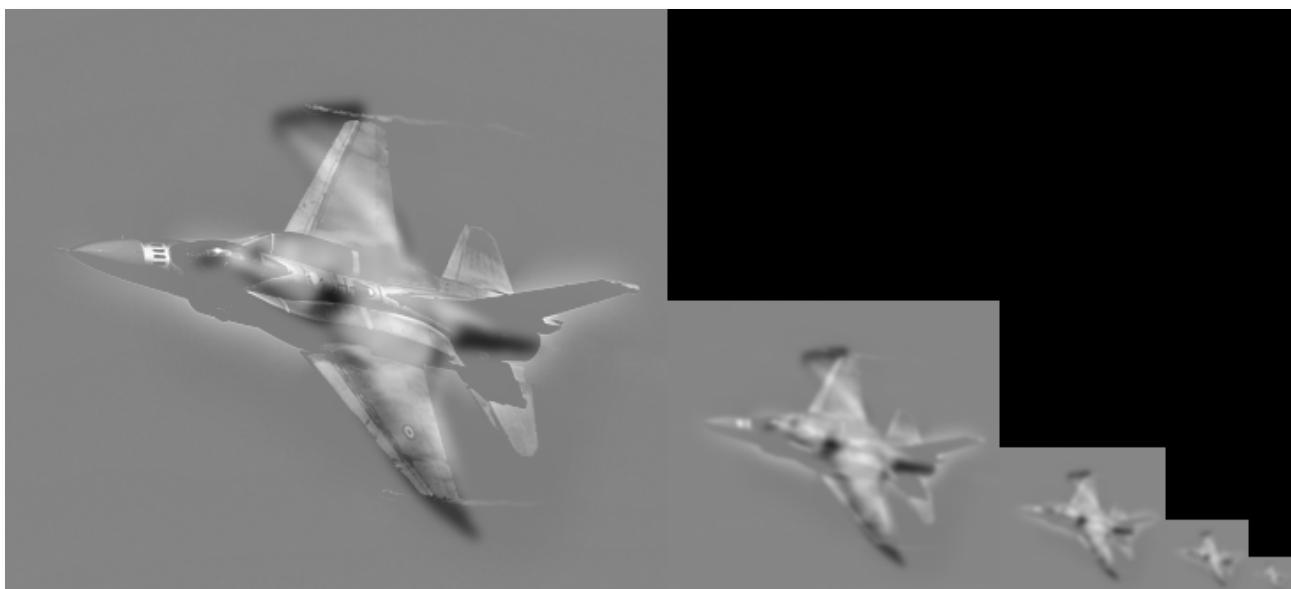
Nuestro siguiente movimiento será insertar el último elemento al final de la señal y el primero al principio tantas veces como indique el ksize (es decir, la longitud de la máscara/2). (Función **insertarBordesReflejados**)

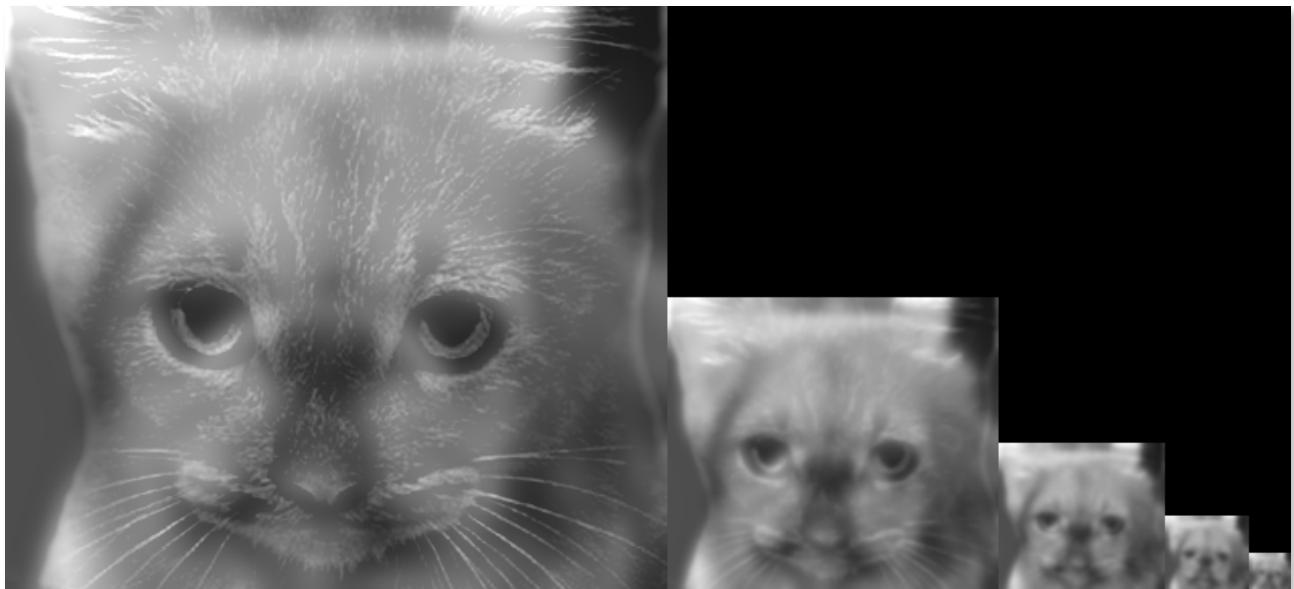
Ahora estamos en condiciones de aplicar la convolución, realizando la operación matricial para todos los “pixeles” comprendidos entre ksize y el tamaño de la imagen - ksize. (Función **calculoConvolucion**)

### 3- Implementar con código propio la convolución 2D con cualquier máscara 2D

### 4 - Construir la pirámide Gaussiana con las imágenes híbridas

El código para este apartado lo realicé dentro de la misma función **imagen\_hibridas** y consiste simplemente en llamar a **piramideGaussiana** con las imágenes híbridas creadas.





## 5 - Realizar todas las parejas de imágenes híbridas

Siguiendo la estructura para imágenes grises, la función `imagen_hibridas_color` origina las siguientes imágenes híbridas:



