Trabajo-1: Filtrado y Detección de regiones

Visión por Computador.

UNIVERSIDAD DE GRANADA

E.T.S.I. INFORMÁTICA Y TELECOMUNICACIÓN

UNIVERSIDAD DE GRANADA

Departamento de Ciencias de la Computación e Inteligencia Artificial

Grado en Ingeniería Informática.

Curso 2019-2020.

Daniel Bolaños Martínez

76592621-E

danibolanos@correo.ugr.es

Grupo 2 - Jueves 11:30h

Índice

1. Implementar las siguientes funciones usando OpenCV.

1.1. Cálculo de la convolución de una imagen con una máscara 2D.

1.1.1. Convolución de una imagen con una Gaussiana 2D.

Para el primer caso, se nos pide calcular la convolución de una imagen con una Gaussiana 2D haciendo uso de la función **getGaussianKernel** de *OpenCV*. La función implementada tiene la siguiente cabecera:

```
def funcionGaussiana(im, sigmaX, sigmaY=0, ksizeX=0, ksizeY=0,
    border=cv2.BORDER_DEFAULT):
```

Como parámetros obligatorios, acepta la imagen a la que queremos calcular la convolución y *sigmaX* como la desviación estándar en la dirección de X dejando como valores por defecto *sigmaY*=0, al que asignaremos el mismo valor que *sigmaX* y *ksize*=(0,0) que se obtendrá a partir de la relación vista en clase que toma el valor del 95 % de la distribución normal de la Gaussiana centrada en el origen:

$$ksize_{x,y} = 2 \cdot \lfloor 3\sigma_{x,y} \rfloor + 1$$

El ancho del kernel, debe ser impar y positivo, por lo que se comprobará si es par y en ese caso, se sumará 1 a su valor. Además si no se especifica un valor del tipo de borde como parámetro, por defecto se aplicará *BORDER_DEFAULT*.

Para aplicar la convolución a la imagen, procederemos de la siguiente forma:

- Convertimos los valores de la matriz de **uint8** a **float64**, este proceso será común al resto de funciones y sólo revertiremos el cambio cuando mostremos las imágenes por pantalla.
- Haciendo uso de la función **getGaussianKernel** de *OpenCV*, calculamos el kernel de la Gaussiana en la dirección X y en la de Y.

```
kerX = cv2.getGaussianKernel(ksize=ksizeX, sigma=sigmaX)
kerY = cv2.getGaussianKernel(ksize=ksizeY, sigma=sigmaY)
```

La función **getGaussianKernel** devuelve un vector columna, por lo que para aplicar correctamente el kernel en la dirección de X, debemos transponerlo, usando, en mi caso, la función **transponse** de *numpy*.

Si aplicásemos directamente las máscaras creadas a la imagen haciendo uso de la función **filter2D** de *OpenCV*, estaríamos calculando la correlación de la imagen, cuando lo que se pide es la convolución. Para solucionarlo, debemos revertir el orden de los elementos de cada vector, para ello podemos usar la función **flip** de *numpy*. En este apartado específicamente, podemos obviar este paso, debido a que los kernels son simétricos respecto al centro y por tanto aplicar **flip**, los dejaría igual.

Finalmente, aplicamos las máscaras en ambas direcciones sobre la imagen original usando la función **filter2D** dando como resultado, la convolución de la imagen inicial. Especificamos el parámetro *ddepth*=-1 para conservar la profundidad de la imagen original.

```
imblur = cv2.filter2D(im, ddepth=-1, kernel=kerX, borderType=
    border)
imblur = cv2.filter2D(imblur, ddepth=-1, kernel=kerY,
    borderType=border)
```

A continuación, se mostrarán los resultados obtenidos:

Figura 1: GaussianBlur con distintos sigmas

Aumentando el valor de sigma, mayor será el tamaño de la máscara y aumentará el número de píxeles vecinos que intervienen en el suavizado de la imagen, perdiendo la nitidez de la misma y provocando que la imagen se vea más borrosa.

Si aumentamos el valor de una de las dos coordenadas del parámetro *ksize* podremos apreciar una deformación en esa dirección. Fijando el valor $\sigma = 10$ y especificando un valor diferente de *ksize* en cada dirección la deformación es evidente.

Figura 2: $\sigma = 10$, $ksize = (29, 1)$, $ksize = (1, 29)$

También se puede apreciar el mismo efecto, si usamos un valor muy grande de *sigmaX* y uno pequeño de *sigmaY* y dejamos que se calculen los respectivos *ksize* a partir de su valor.

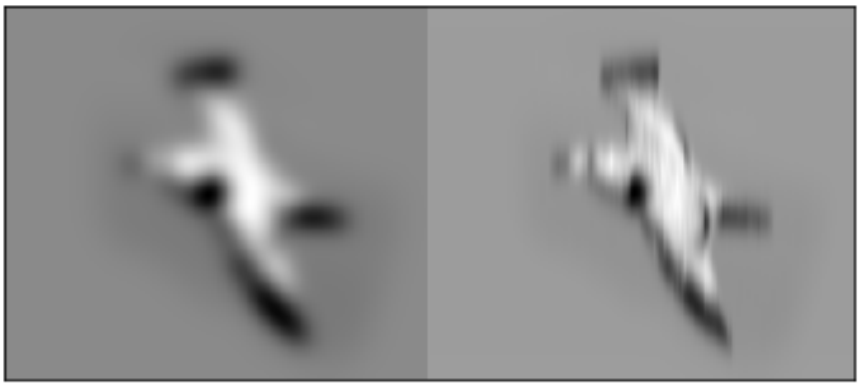

Figura 3: $\sigma_X = 10 \;\; \sigma_Y = 0$, $\sigma_X = 0 \;\; \sigma_Y = 10$

También podemos observar variaciones si aplicamos diferentes condiciones de contorno, especificando el tipo de borde empleado cuando la matriz de suavizado se sale del rango de la imagen. Para este ejercicio he utilizado tres tipos de bordes, además del por defecto:

- Constantes (*BORDER_CONSTANT*) completa la imagen con un valor de pixel constante, en nuestro caso añade un borde negro por defecto. Ejemplo: iiiiii/abcdefgh/iiiiiii con el valor de 'i' especificado.
- Replicados (*BORDER_REPLICATE*) asigna el mismo valor del pixel vecino más cercano. Ejemplo: aaaaaa/abcdefgh/hhhhhhh
- Reflejados (*BORDER_REFLECT*) refleja el valor de los píxeles de forma antisimétrica a la original. Ejemplo: fedcba—abcdefgh—hgfedcb

Para los 4 casos, he fijado el valor de $\sigma = 10$ y un tamaño de máscara de $ksize = (21, 21)$ para mejorar la observación de los bordes. El funcionamiento del borde constante es mucho más claro al ojo humano que el resto por lo explicado anteriormente.

Figura 4: Condición de contorno CONSTANT

Figura 5: Condición de contorno REPLICATE

Figura 6: Condición de contorno REFLECT

Figura 7: Condición de contorno DEFAULT

Además he comparado los resultados de la función programada utilizando **getGaussianKernel** con los de la función **GaussianBlur** que implementa *OpenCV*. Fijado un $\sigma = 10$ los resultados de ambos casos, son bastante similares como podemos observar en las imágenes.

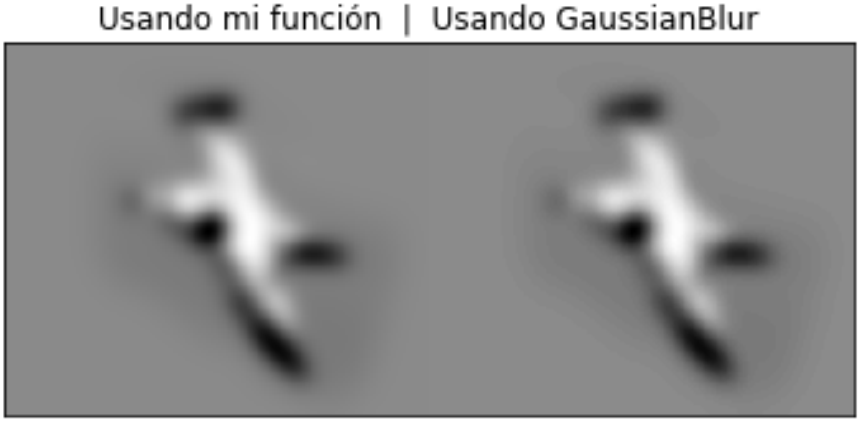

Figura 8: Convolución de imágenes con $\sigma = 10$

1.1.2. Convolución de una imagen con máscaras 1D.

Para el segundo caso, se nos pide calcular la convolución de una imagen usando máscaras de derivadas. Estas máscaras nos permiten detectar frecuencias altas que se relacionan con las zonas donde existen saltos o fronteras en las imágenes. La función implementada tiene la siguiente cabecera:

```
def convMasks1D(im, dx, dy, ksize, border=cv2.BORDER_DEFAULT):
```

Como parámetros obligatorios, acepta la imagen a la que queremos calcular la convolución, *dx* y *dy* como los órdenes de las derivadas sobre cada eje y *ksize* como el tamaño del kernel. Además si no se especifica un valor del tipo de borde como parámetro, por defecto se aplicará *BORDER_DEFAULT*.

Para aplicar la convolución a la imagen, procederemos de la siguiente forma:

- Convertimos los valores de la matriz de **uint8** a **float64**.
- Haciendo uso de la función **getDerivKernel** de *OpenCV*, calculamos el kernel de orden dx en la dirección X y dy en la de Y. Hacemos uso de la variable *normalize* de la función para normalizar los coeficientes de la máscara.

```
kerX, kerY = cv2.getDerivKernels(dx=dx, dy=dy, ksize=ksize,
    normalize=True)
```

La función **getDerivKernel** también devuelve un vector columna, por lo que al igual que en la función anterior, debemos transponer *kerX*, usando, en mi caso, la función **transponse** de *numpy*.

Para calcular la convolución en lugar de la correlación, esta vez sí, debemos revertir el orden de los elementos de cada vector (tanto *kerX* como *kerY*) usando la función **flip** de *numpy*.

Finalmente, aplicamos las máscaras en ambas direcciones sobre la imagen original usando la función **filter2D**, dando como resultado la convolución de la imagen inicial. Especificamos el parámetro *ddepth*=-1 para conservar la profundidad de la imagen original.

```
imconv = cv2.filter2D(im, ddepth=-1, kernel=kerX, borderType=
    border)
imconv = cv2.filter2D(imconv, ddepth=-1, kernel=kerY,
    borderType=border)
```

A continuación, se mostrarán los resultados obtenidos:

Figura 9: Convolución con getDerivKernels()

Podemos observar como fijando *ksize*=5 y cambiando los valores de los órdenes como el nivel de sensibilidad disminuye conforme reducimos el orden de las derivadas y como en la imagen central, tomando dx=3 y dy=0 se puede apreciar como la sensibilidad del ruido del eje X es mayor (se pierden detalles de los bordes en esa dirección).

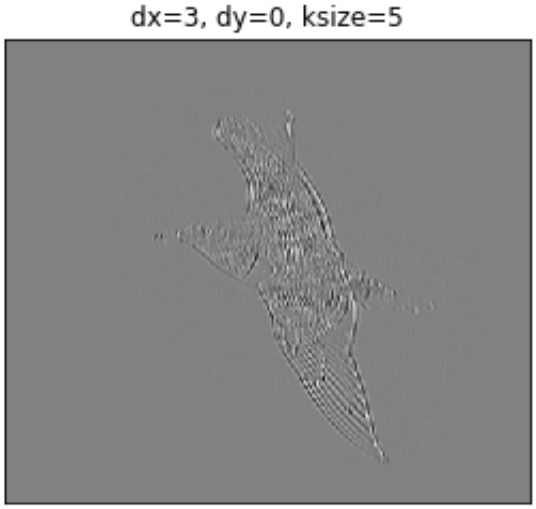

Figura 10: $dx - dy = 3$

El parámetro *ksize* usado en la función **getDerivKernels** actúa de la forma siguiente: a mayor valor, la función será más sensible a los cambios de frecuencia y detectará las fronteras a mayor escala. Podemos notar cómo fijando el orden de las derivadas en ambos ejes, y variando el valor de *ksize* la sensibilidad a cambios de fronteras, aumenta de forma directamente proporcional.

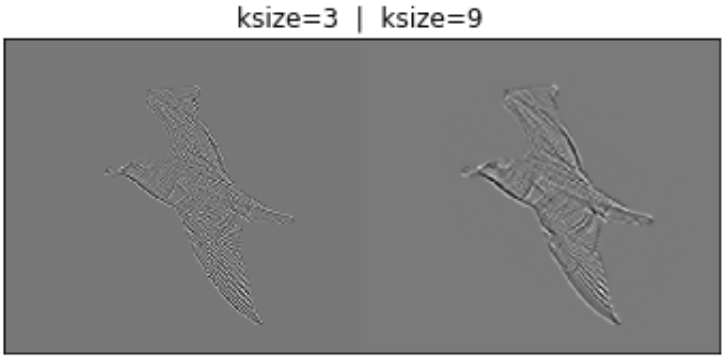

Figura 11: Fijo $d_x = d_y = 1$ con $ksize$ variable

1.2. Calcular la convolución 2D de una imagen con una máscara normalizada de Laplaciana-de-Gaussiana de tamaño variable.

Para este apartado se nos pide calcular la convolución de una imagen con una máscara normalizada de Laplaciana-de-Gaussiana.

La función Laplaciana consiste en la derivada de orden 2 de la Gaussiana en ambas variables por lo que se usa para detectar los bordes de la imagen sea cual sea su dirección. Esta función detecta el contraste de las imágenes y suele aplicarse después de usar la Gaussiana para reducir la sensibilidad al ruido.

La fórmula utilizada en el cálculo de la función Laplaciana es:

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

La función implementada tiene la siguiente cabecera:

```
def laplacianaGaussiana(im, sigma, ksize=1, border=cv2.
    BORDER_DEFAULT):
```

Como parámetros obligatorios, acepta la imagen a la que queremos calcular la convolución, *sigma* como el valor de la desviación estándar y *ksize* como el tamaño del kernel, que si no se especifica tomará el valor de 1 para que funcione de forma similar a la función **Laplacian** de *OpenCV*. Además si no se especifica un valor del tipo de borde como parámetro, por defecto se aplicará *BORDER_DEFAULT*.

Para aplicar la convolución a la imagen, procederemos de la siguiente forma:

- Convertimos los valores de la matriz de **uint8** a **float64**.
- Aplicamos a la imagen una convolución con una máscara Gaussiana haciendo uso de la función del apartado anterior. De esta forma el filtro será Laplaciano-Gaussiano como se nos pide y además mejoraremos la sensibilidad al ruido.
- Usando la función **getDerivKernel**, calculamos el kernel de orden 2 en ambas direcciones, tal y como está definida la Laplaciana. Hacemos uso de la variable *normalize* de la función para normalizar los coeficientes de la máscara.

```
imblur = funcionGaussiana(im, sigmaX=sigma, border=border)
kerX, kerY = cv2.getDerivKernels(dx=2, dy=2, ksize=ksize,
    normalize=True)
```

Al igual que hemos hecho en el apartado anterior con la convolución de máscaras separables 1D, debemos transponer el vector *kerX* y aplicar la función **flip** a los dos kernels, por las razones ya explicadas.

Finalmente, aplicamos las máscaras en ambas direcciones sobre la imagen a la que se le aplicó la máscara Gaussiana. Para ello, usamos la función **filter2D** por separado y luego sumamos las dos imágenes resultantes (por

definición del Laplaciano), de esta forma, obtendremos la convolución pedida.

```
imX = cv2.filter2D(imblur, ddepth=-1, kernel=kerX, borderType=
    border)
imY = cv2.filter2D(imblur, ddepth=-1, kernel=kerY, borderType=
    border)
return imX+imY
```

A continuación, se mostrarán los resultados obtenidos:

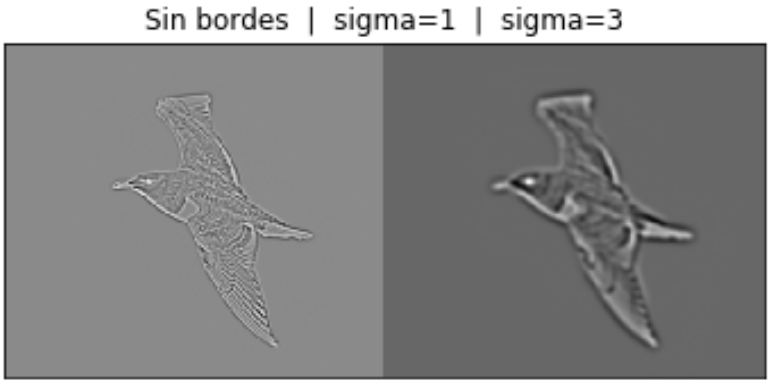

Figura 12: $\sigma = 1$, $\sigma = 3$

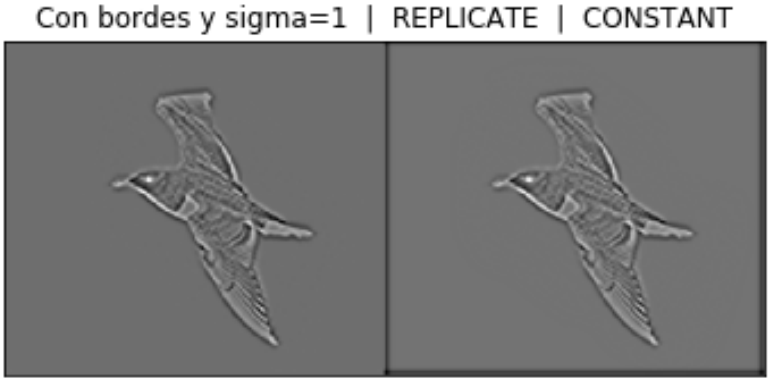

Figura 13: $\sigma = 1$ $ksize = 21$

El sigma pasado como parámetro a la función interviene en la **Gaussiana** pero tendrá mayor relevancia en la función **Laplaciana** donde un valor más pequeño del sigma aumentará la tolerancia a los bordes.

He usado los mismos dos tipos de bordes del apartado anterior (*BORDER_CONSTANT* y *BORDER_REPLICATE*).

Tomando *ksize*=21, podemos observar mejor cómo actúa cada tipo de borde. En estas imágenes al no contener el fondo con ningún nivel de detalle no se pueden apreciar muy bien los bordes. El constante se puede ver más que el replicado por su funcionamiento. En el caso del borde constante, la imagen tiene un borde oscuro que no aparece al replicar los bordes.

2. Implementar funciones para las siguientes tareas.

2.1. Representación en pirámide Gaussiana de 4 niveles de una imagen.

Se nos pide programar una función que genere la representación de una pirámide Gaussiana de 4 niveles de una imagen. La función implementada tiene la siguiente cabecera:

```
def pirGaussiana(im, size, border=cv2.BORDER_DEFAULT):
```

Como parámetros obligatorios, acepta la imagen sobre la que vamos a construir la pirámide Gaussiana y el número de niveles que queremos representar. En mi caso, represento 4 niveles junto con la imagen original, lo que hacen un total de 5 imágenes. Si no se especifica un valor del tipo de borde como parámetro, por defecto se aplicará *BORDER_DEFAULT*. El tipo de borde se aplicará sobre cada imagen en el momento de aplicar el filtro Gaussiano.

Para realizar la función, primero he tenido que modificar el método **concatenaImgs** de la práctica inicial, ya que para la correcta visualización de

la pirámide, no podemos hacer uso de la función **resize** de *OpenCV*, ya que queremos reducir el tamaño de las imágenes sucesivas y si usásemos **resize** se pixelarían y obtendrían el tamaño de la original.

La modificación consiste en utilizar la función de *OpenCV* **copyMakeBorder** y añadir un borde constante a la imagen de manera que pueda seguir concatenándose a las otras pero sin perder su tamaño.

El proceso para cada nivel de la pirámide es el siguiente:

- Convertimos los valores de la matriz de **uint8** a **float64**.
- Aplicamos un suavizado Gaussiano con nuestra **funcionGaussiana** sobre la imagen original. Fijamos los valores de $\sigma = 5$ y *ksize* se calcula en función de *sigma* como hemos mencionado en el ejercicio anterior. He tomado $\sigma = 5$ para que tanto el suavizado como los bordes sean mejor observables.
- Hacemos uso de la función **reducirIm** que actúa como *subsampling* eliminando las filas y columnas impares de la matriz imagen para reducir su tamaño.
- Repetiremos el proceso sobre cada imagen generada tantas veces como número de niveles de la pirámide hayamos especificado, de forma que se pierda el mínimo número de información con cada reducción.

La pirámide obtenida, muestra sucesivas imágenes que parten de la original y a las que se le aplica tanto una reducción de tamaño como un suavizado Gaussiano. Se puede observar la diferencia entre ambas pirámides de forma más clara conforme las imágenes se van reduciendo, el borde constante aparece de forma más sutil en la Figura 15.

Figura 14: Pirámide Gaussiana de 4 niveles (Sin Bordes)

Figura 15: Pirámide Gaussiana de 4 niveles (Borde Constante)

2.2. Representación en pirámide Laplaciana de 4 niveles de una imagen.

Se nos pide programar una función que genere la representación de una pirámide Laplaciana de 4 niveles de una imagen. La función implementada tiene la siguiente cabecera:

```
def pirLaplaciana(im, size, border=cv2.BORDER_DEFAULT):
```

Como parámetros obligatorios, acepta la imagen a la que vamos a realizarle la pirámide Laplaciana y el número de niveles que queremos representar. En mi caso, represento 4 niveles. Si no se especifica un valor del tipo de borde como parámetro, por defecto se aplicará *BORDER_DEFAULT*. El tipo de borde se aplicará sobre cada imagen en el momento de aplicar el filtro Gaussiano.

En la pirámide Laplaciana, cada nivel de la pirámide muestra las frecuencias altas de la imagen que equivalen a resaltar los bordes de la misma.

He tenido que programar una función **aumentarIm** que funciona como *upsampling* a la que le paso dos parámetros, la imagen a aumentar y la imagen origen que contiene el tamaño de filas y columnas que tengo que alcanzar en ese aumento. Es necesario conocer las dimensiones de la imagen original ya que no podemos saber de otra forma si la imagen original tenía columnas o filas impares y por tanto, esto podría provocar que a la hora de operar con ambas imágenes, tuviesemos problemas con las dimensiones de las matrices.

El proceso de aumentar la imagen consiste en replicar las filas y columnas adyacentes (fila *i-1*, columna *j-1*) en las posiciones impares de la matriz hasta alcanzar el mismo número de dimensiones que la matriz original (si fuese necesario, se añade una fila o columna más para ello). Además, debemos aplicar un filtro Gaussiano a la imagen aumentada.

Se toma la decicisión de replicar las filas y columnas en lugar de añadir por ejemplo filas y columnas de ceros, para evitar reducir la intensidad de los píxeles originales al realizar el autocompletado y obtener resultados erróneos.

El proceso de construcción de la pirámide es el siguiente:

- Construimos la pirámide Gaussiana de *n+1* niveles. No es necesario convertir la matriz a **float64** puesto que **pirGaussiana** ya devuelve el resultado en ese formato.
- Restamos a la imagen en la posición i la imagen *i+1* a la que hemos aplicado la función **aumentarIm**.

- Introducimos en una lista la imagen resultante y repetimos el proceso hasta n.

La pirámide obtenida, muestra sucesivas imágenes a las que se le aplica tanto una reducción de tamaño como un resultado similar al filtro Laplaciano. En este caso, la diferencia de la pirámide con y sin bordes es más evidente, debido a que el color que toma el borde es blanco en la Figura 17.

Figura 16: Pirámide Laplaciana de 4 niveles (Sin Bordes)

Figura 17: Pirámide Laplaciana de 4 niveles (Borde Constante)

2.3. Construir un espacio de escalas Laplaciano para implementar la búsqueda de regiones.

Se nos pide construir un espacio de escalas Laplaciano que implemente la búsqueda de regiones usando un algoritmo definido. La función implementada tiene la siguiente cabecera:

```
def busqRegiones(im, n, umbral=15, sigma=1):
```

Como parámetros obligatorios, acepta la imagen sobre la que vamos a construir el espacio de búsqueda de regiones, n como el número de escalas Laplaciano, *umbral* como el valor de los píxeles a partir del cual dibujaremos círculos sobre ellos y *sigma* el valor de la desviación estándar de la Laplaciana-Gaussiana aplicada en cada escala.

Tomamos los valores por defecto: *umbral*=15 para visualizar el mayor número de píxeles y *sigma*=1 para empezar la búsqueda desde un valor razonablemente pequeño e ir incrementándolo en cada escala.

Además, se definen dentro de la función, dos constantes (*cte* y K) siendo el coeficiente de incremento de sigma en cada escala y la constante proporcionalidad de la escala del radio de los círculos respectivamente. Fijamos los valores de *cte*=1.2 $\in$ [1,2, 1,4] y K=4 para obtener un visualizado óptimo de los círculos dibujados.

La función **supresionNoMax**, toma como parámetros *im* como la imagen a la que vamos a aplicar la supresión, *pos* como la posición de esa imagen en el vector de escalas y *vim* como el vector de escalas. Distingo varios casos para evitar salirnos del rango del vector *vim*, en el caso de que la imagen a la que aplicamos la supresión se encuentre en la posición 0 o n-1 del vector.

Luego, para cada píxel llamo a la función **neighbors** a la que paso como parámetros: la imagen actual, las coordenadas (i ,j) del píxel del que obtendremos su entorno de vecinos adyacentes y un flag que por defecto es igual a 0 y que cuando vale 1 añade el pixel de la posición (i,j) a la lista de vecinos. Esto último lo implementamos para calcular la lista de vecinos en las escalas superior e inferior y añadir el elemento central de la matriz 3x3.

Si el elemento actual tiene la posición 0 en el vector de escalas, no tendrá lista de vecinos inferior; de igual manera, si se encuentra en la posición n-1, no tendrá lista de vecinos superior. En estos casos, tomaremos máximo inferior y máximo superior igual a 0.0 respectivamente. Finalmente calcularemos el máximo en el entorno cúbico como el máximo entre los máximos inferior, superior y actual (que no añade el pixel (i,j)). En el caso de que el pixel (i,j) sea máximo del cubo, dejamos su valor intacto, si no, ponemos en la copia (para no modificar el píxel), su valor a 0.0.

El algoritmo programado en la construcción del espacio de búsquedas Laplaciano, es el siguiente:

- Convertimos los valores de la matriz de **uint8** a **float64**.
- Fijamos $\sigma = 1$.
- Usamos un bucle **for** para repetir n (número de escalas) veces el siguiente proceso. Para cada iteración i:
  - Calcular la Laplaciana-Gaussiana de la imagen con σ_i y multiplicar su resultado por σ_i^2.
  - Elevar cada término de la matriz anterior al cuadrado.
  - Guardar en resultado en el vector de escalas.
  - Multiplicamos el valor de sigma por la constante (cte) de escalado.
- Usamos otro bucle **for** para aplicar la supresión de no máximos sobre cada elemento del vector de escalas, para implementarla con entornos cúbicos. Para cada iteración i:
  - Aplicar la supresión de no máximos a la imagen resultante con la función **supresionNoMax** teniendo en cuenta el resto de escalas para el cálculo de los vecinos en entornos cúbicos.
  - Normalizar la imagen obtenida usando la función **normalizaRGB** de la práctica 0.
  - Guardar los índices de los píxeles cuyo valor supere al del *umbral* en una lista.

- Pintar los círculos usando la función **circle** de *openCV* sobre los píxeles almacenados en la lista tanto sobre la imagen normalizada somo sobre la original.
- Guardamos ambas imágenes en un vector como tupla [imNormalizada, imOriginal].

- Devolvemos el vector de tuplas.

Nota: Es importante guardar las posiciones de los píxeles antes de dibujar los círculos, ya que **circle** al pintar sobre la imagen, añade nuevos píxeles en el color especificado (blanco) y esto causaría contradicciones.

En las Figuras 18 y 19, podemos observar los resultados obtenidos para 2 escalas con *umbral*=50. Los círculos se dibujan sobre los píxeles con un valor mayor a 20. Vemos, como los píxeles destacados, corresponden con los bordes de la imagen, mostrando una figura de la misma con fondo negro. También podemos observar como al aumentar la escala, aumenta el número de píxeles en las zonas destacadas.

Figura 18: Espacio de búsqueda Laplaciano Einstein

Figura 19: Espacio de búsqueda Laplaciano Einstein

En las Figuras 20 y 21, observamos los resultados obtenidos para la primera escala sobre la misma imagen con el parámetro *umbral* diferente. Es evidente que cuanto menor sea el umbral, mayor será el número de píxeles destacados con los círculos.

Figura 20: Espacio de búsqueda Bird: umbral=50

Figura 21: Espacio de búsqueda Bird: umbral=25

3. Implementar una función que genere las imágenes de baja y alta frecuencia a partir de las parejas de imágenes para generar imágenes híbridas.

3.1. Escribir una función que muestre las tres imágenes (alta, baja e híbrida).

Se nos pide generar imágenes híbridas a partir de las parejas de imágenes dadas e implementar una función que muestre por pantalla las tres versiones pedidas (alta, baja, híbrida). La función implementada tiene la siguiente cabecera:

```
def imgsHibridas(im1, im2, sigma1, sigma2):
```

Para hacer las imágenes híbridas, haremos uso de las parejas de imágenes proporcionadas que tienen posiciones y formas similares. Las imágenes se pasarán como los parámetros *im1* (bajas frecuencias) e *im2* (altas frecuencias) de la función. Además le pasaremos dos sigmas, cada uno será usado de forma independiente en la obtención de las frecuencias bajas de la imagen correspondiente.

Para cada pareja de imágenes, aplicaremos a una un filtro de paso bajo y a la otra un filtro de paso alto y finalmente combinaremos ambos resultados provocando el efecto óptico de ver una imagen u otra dependiendo de a la distancia desde la que observemos. Este efecto se debe a que el ser humano detecta mejor las frecuencias altas desde cerca y las bajas desde lejos.

Para crear la imagen de frecuencias bajas, aplicaremos **funcionGaussiana** a *im1* usando *sigma1*, esta función, como ya hemos dicho antes, tiene la peculiaridad de eliminar las frecuencias altas, por lo que obtendremos el resultado que queremos.

Para obtener las frecuencias altas de *im2*, primero obtendremos las bajas procediendo de la misma forma explicada anteriormente y luego restaremos a la imagen original la imagen en paso bajo, quedándonos con las frecuencias altas de la imagen.

Finalmente combinaremos la imagen en paso bajo con la imagen en paso alto para obtener la imagen híbrida. Para mezclar ambas imágenes las sumaremos.

Para obtener mejor resultado final, es importante que ajustemos bien el tamaño de los sigmas, que variará dependiendo de las imágenes utilizadas, y que como ya sabemos actuará con mayor o menor sensibilidad sobre el filtro de paso bajo.

3.2. Realizar composición con al menos 3 parejas de imágenes.

Los resultados obtenidos para tres parejas de imágenes, se pueden observar en las Figuras 22, 23, 24.

Figura 22: Gato-Perro: $\sigma_1 = 8$ $\sigma_2 = 5$

Figura 23: Marilyn-Einstein: $\sigma_1 = 5$ $\sigma_2 = 3$

Figura 24: Bicicleta-Moto: $\sigma_1 = 9$ $\sigma_2 = 3$

3.3. Crear pirámide Gaussiana e 4 niveles con las imágenes híbridas.

Se nos pide crear una pirámide Gaussiana de 4 niveles a partir de las imágenes híbridas. Si aplicamos la función creada en el apartado *2A* sobre alguna de las imágenes resultantes en la hibridación, podemos visualizar el mismo efecto que obtenemos al observar la imagen de cerca y lejos con el ojo humano.

Con la pirámide Gaussiana, en cada nivel, estamos alejando y desenfocando la imagen, igual que lo hace el ojo humano al alejarse de una fotografía. Por lo que podemos, sin necesidad de alejarnos, ver como actúa el efecto sobre cada pareja de imágenes.

Figura 25: Pirámide Gaussiana Gato-Perro

Figura 26: Pirámide Gaussiana Einstein-Marilyn

Figura 27: Pirámide Gaussiana Moto-Bicicleta

4. Bonus

4.1. Implementar con código propio la convolución 2D con cualquier máscara 2D de números reales usando máscaras separables.

No ha sido implementado.

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

Se nos pide generar imágenes híbridas a color a partir de las parejas de imágenes dadas e implementar una función que muestre por pantalla las tres versiones pedidas (alta, baja, híbrida). Figuras 28, 29 y 30. La función implementada tiene la siguiente cabecera:

```
def imgsHibridasColor(im1, im2, sigma1, sigma2):
```

Las funciones para pintar ya están diseñadas para que dibujen en color, por lo que simplemente adaptamos el código de la función que genera imágenes híbridas para que en el caso de que añadamos una imagen en grises y otra a color, pase ambas a color y proceda a generar su híbrida. También podemos observar los resultados de las pirámides Gaussianas de 4 niveles con las imágenes híbridas a color. Figuras 31, 32 y 33.

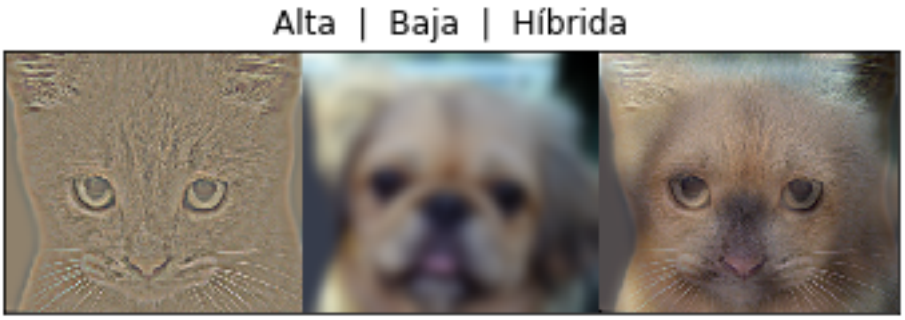

Figura 28: Gato-Perro Color: $\sigma_1 = 8$ $\sigma_2 = 5$

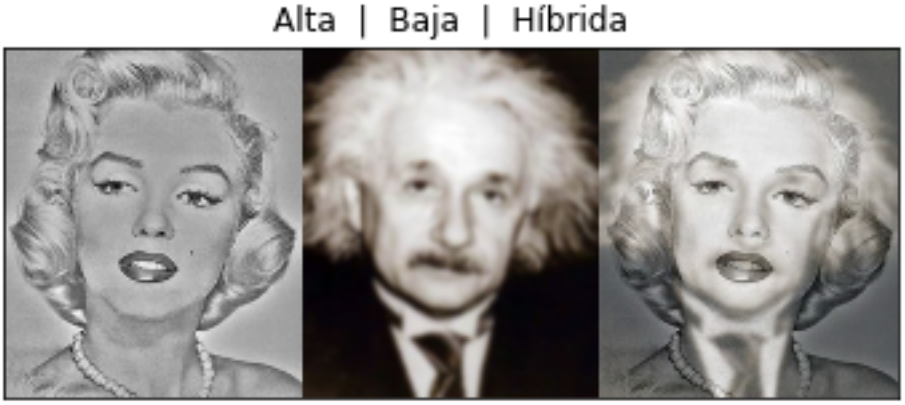

Figura 29: Marilyn-Einstein Color: $\sigma_1 = 5$ $\sigma_2 = 3$

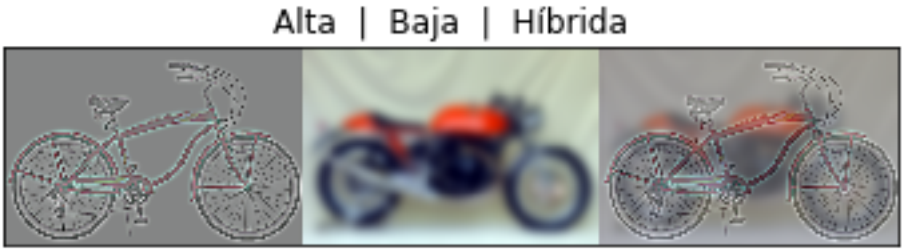

Figura 30: Bicicleta-Moto Color: $\sigma_1 = 9$ $\sigma_2 = 3$

Figura 31: Pirámide Gaussiana Gato-Perro Color

Figura 32: Pirámide Gaussiana Einstein-MarilynColor

Figura 33: Pirámide Gaussiana Moto-BicicletaColor

4.3. Realizar una imagen híbrida con al menos una pareja de imágenes de su elección que hayan sido extraídas de imágenes más grandes.

Debemos generar una imagen híbrida similar a las del ejercicio 3, pero con imágenes elegidas por nosotros. Para ello utilizo la función del ejercicio 3 para hibridar la imagen elegida no sin antes recortarla de forma manual para que pueda ser utilizada en la hibridación.

Las imágenes escogidas, por su parecido en forma y fondo, han sido las siguientes:

Figura 34: Balón imagen original

Figura 35: Sandía imagen original

Una vez aplicado el recorte de forma manual, obtengo las siguientes imágenes, modificadas y en formato bmp.

Figura 36: Balón imagen recortada

Figura 37: Sandía imagen recortada

Finalmente, aplicamos la función **imgsHibridas** sobre ambas imágenes. Tomando como valores de sigma: *sigma1*=10 y *sigma2*=5. Mostramos a continuación los resultados obtenidos:

Figura 38: Balón-Sandía: $\sigma_1 = 10$ $\sigma_2 = 5$

Figura 39: Balón-Sandía Color: $\sigma_1 = 10$ $\sigma_2 = 5$

Figura 40: Pirámide Gaussiana 4 niveles

5. Funciones externas

Para la realización de la práctica, he utilizado algunas funciones implementadas en la práctica 0 que me han servido de ayuda al leer imágenes y facilitar la muestra de resultados en las imágenes modificadas.

- **leeimagen(filename, flagColor)**: Lee una imagen con el flag del color especificado. 1 para color y 0 para blanco y negro.
- **normalizaRGB(im)**: Normaliza una matriz de flotantes (en escala de grises o a color) y la escala al rango 0-255.
- **pintaI(im, title)**: Normaliza la imagen a escala RGB, la muestra por pantalla y le pone un título.
- **concatenaImgs(vim)**: Normaliza una lista de imágenes a escala RGB, añade un borde (*BORDER_CONSTANT*) superior hasta cuadrar el número de filas de todas las imágenes y las concatena horizontalmente.
- **pintaAll(vim, titles)**: Concatena las imágenes de una lista, las muestra por pantalla como una sola y añade un título.

6. Bibliografía

[1]. Documentación de OpenCV: https://docs.opencv.org/4.1.0/index.html

[2]. Diapositivas de clase.