



UNIVERSIDAD DE GRANADA

VISIÓN POR COMPUTADOR
GRADO EN INGENIERÍA INFORMÁTICA

TRABAJO 1

FILTRADO Y DETECCIÓN DE REGIONES

Autor

Vladislav Nikolov Vasilev

Rama

Computación y Sistemas Inteligentes



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE
TELECOMUNICACIÓN

CURSO 2019-2020

Índice

| | |
|---|-----------|
| 1. EJERCICIO SOBRE FILTROS BÁSICOS | 2 |
| 1.1. Apartado A | 2 |
| 1.2. Apartado B | 12 |
| 2. EJERCICIO SOBRE PIRÁMIDES Y DETECCIÓN DE REGIONES | 13 |
| 2.1. Apartado A | 13 |
| 2.2. Apartado B | 13 |
| 2.3. Apartado C | 13 |
| 3. IMÁGENES HÍBRIDAS | 14 |
| 3.1. Apartado 1 | 14 |
| 3.2. Apartado 2 | 14 |
| 3.3. Apartado 3 | 14 |
| 4. BONUS | 15 |
| 4.1. Convolución 2D propia | 15 |
| 4.2. Imágenes híbridas a color | 15 |
| 4.3. Imágen híbrida propia | 15 |
| Referencias | 16 |

1. EJERCICIO SOBRE FILTROS BÁSICOS

USANDO LAS FUNCIONES DE OPENCV: escribir funciones que implementen los siguientes puntos:

- A) El cálculo de la convolución de una imagen con una máscara 2D. Usar una Gaussiana 2D (GaussianBlur) y máscaras 1D dadas por getDerivKernels). Mostrar ejemplos con distintos tamaños de máscara, valores de sigma y condiciones de contorno. Valorar los resultados.
- B) Usar la función Laplacian para el cálculo de la convolución 2D con una máscara normalizada de Laplaciana-de-Gaussiana de tamaño variable. Mostrar ejemplos de funcionamiento usando dos tipos de bordes y dos valores de sigma: 1 y 3.

1.1. Apartado A

Para realizar todas las pruebas, vamos a utilizar una imagen en blanco y negro. De esta forma, los resultados se podrán ver de forma más clara. Se puede utilizar cualquiera de las imágenes que se han proporcionado, pero vamos a utilizar la foto del gato. A continuación se puede ver la imagen en cuestión:

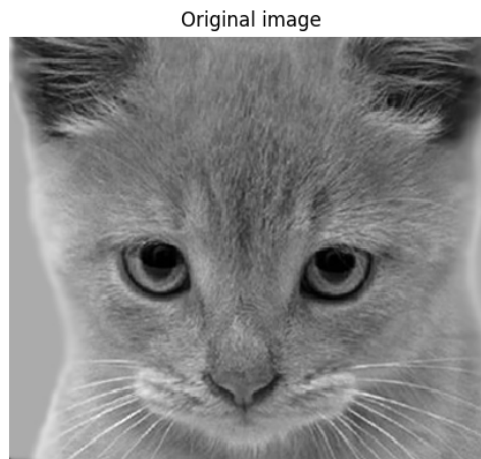


Figura 1: Imágen original del gato en blanco y negro.

Antes de ver cuáles son los resultados y compararlos. Hace falta aclarar algunos puntos:

- Cuando se cargan las imágenes, se convierten a *float64*, para así tener más precisión a la hora de hacer las posteriores operaciones (aplicar filtros, sumar o restar imágenes, etc.), además de que así no nos limitamos a usar solo valores en el rango $[0, 255]$.
- Las operaciones para aplicar filtros de OpenCV aplican una correlación. Por tanto, para aplicar una convolución, se debe realizar un *flip* del *kernel*; esto es, darle la vuelta en el eje *X* y en el eje *Y*. Como en todos los casos trabajaremos con *kernels* separables, solo será necesario darles la vuelta en un sentido.
- Respecto al punto anterior, al trabajar con *kernels* separables, con tal de mejorar la eficiencia, se puede aplicar primero el *kernel* del eje *X* y, sobre el resultado, aplicar el *kernel* en el eje *Y*. Esto es mucho más rápido que aplicar directamente un *kernel* 2D sobre la imagen, debido a que el número de operaciones es mucho menor.

Como en este ejercicio se piden aplicar *kernels* de Gaussiana y de derivada, se han hecho funciones específicas las cuáles pueden ser consultadas en el código. Estas funciones son *gaussian_kernel()* y *derivative_kernel()*, respectivamente. No se va a especificar exactamente lo que hacen debido a que solo obtienen los *kernels* con los parámetros adecuados (valores de σ y tamaño de cada *kernel* para el caso de la Gaussiana, y tamaño del *kernel* y número de veces que se deriva en cada eje en el caso de la derivada).

Lo que sí que es importante destacar es que estas dos funciones utilizan una común para realizar la convolución. Como se dijo anteriormente, OpenCV realiza como tal la correlación, pero se puede aplicar una convolución realizando unos pocos cambios. La función que se puede ver a continuación recibe un *kernel* para cada eje y aplica el filtro mediante la convolución:

```
1 def apply_kernel(img, kx, ky, border):
2     """
3     Funcion que aplica un kernel separable sobre una imagen,
4     realizando una convolucion
5
6     Args:
7         img: Imagen sobre la que aplicar el filtro
8         kx: Kernel en el eje X
9         ky: Kernel en el eje Y
10        border: Tipo de borde
11    Return:
12        Devuelve una imagen filtrada
13    """
```

```
14     # Hacer el flip a los kernels para aplicarlos como una
      convolucion
15     kx_flip = np.flip(kx)
16     ky_flip = np.flip(ky)
17
18     # Realizar la convolucion
19     conv_x = cv.filter2D(img, cv.CV_64F, kx_flip.T,
20                          borderType=border)
21     conv = cv.filter2D(conv_x, cv.CV_64F, ky_flip,
22                       borderType=border)
23
24     return conv
```

Como se puede ver, se tiene que hacer un *flip* de los *kernels* para poder aplicar una convolución. Al aplicar cada *kernel* de forma separada mediante *filter2D*, se consigue una mayor eficiencia (como se ha mencionado anteriormente). Es importante destacar que primero se aplica el *kernel* sobre el eje de las *X* y, sobre el resultado obtenido, se aplica el *kernel* en el eje *Y*. También es importante destacar que, cuando se aplica el *kernel* sobre el eje *X*, se le pasa la traspuesta del *kernel*. Esto se debe a que OpenCV proporciona los *kernels* como vectores columnas, y para pasarlo por las filas, necesitamos que sea un vector fila. Al traponer el vector columna obtenemos, como parece lógico, un vector columna.

Otra cosa muy importante a destacar son los *kernels* que se van a utilizar. Uno de ellos es el *kernel* Gaussiano, el cuál es simétrico tanto en el eje *X* como en el *Y*. Al aplicarlo, por tanto, se podría ahorrar la operación del *flip*, ya que lo va a dejar igual. Sin embargo, debido a que se ha implementado la función para que sea genérica, se realiza esta operación. El otro *kernel* que se utiliza es el de las derivadas, que no es más que un *kernel* de Sobel, el cuál combina alisamiento Gaussiano con la derivada. A diferencia del anterior *kernel*, este no es simétrico, y por tanto, la operación de *flip* no lo va a dejar igual; por tanto, no puede ser ahorrada en este caso.

Con esto comentado, ya podemos empezar a hablar de los resultados que se han obtenido. Para ello, vamos a comenzar comentando los resultados que se obtienen para el filtro Gaussiano.

Lo primero que se ha probado es un *kernel* de 5×5 , variando los valores de σ para ver cómo cambiaba. A continuación se pueden ver los resultados:

5 × 5 Gaussian Blur with $\sigma = 1$ and BORDER_REPLICATE(a) Filtro Gaussiano con $\sigma = 1$.5 × 5 Gaussian Blur with $\sigma = 3$ and BORDER_REPLICATE(b) Filtro Gaussiano con $\sigma = 3$.Figura 2: Filtro Gaussiano con diferentes tamaños de σ .

Como se puede ver, comparando cualquiera de las imágenes de la figura 2 con 1, existen ciertas diferencias. Podemos ver claramente como al aplicar el filtro Gaussiano se ha perdido cierto detalle, ya que se están eliminando las frecuencias altas. Además, si comparamos las figuras 2a y 2b, podemos ver que a medida que aumentamos el tamaño de σ , se van perdiendo más detalles, ya que se emborrona más.

Se ha probado también a variar el tamaño de la máscara, conservando el mismo valor de σ , para ver qué es lo que sucede. A continuación, se puede ver lo que se ha obtenido:

5 × 5 Gaussian Blur with $\sigma = 3$ and BORDER_REPLICATE

(a) Filtro Gaussiano de tamaño 5 × 5.

11 × 11 Gaussian Blur with $\sigma = 3$ and BORDER_REPLICATE

(b) Filtro Gaussiano de tamaño 11 × 11.

Figura 3: Filtro Gaussiano con diferentes tamaños de *kernel*.

Tal y como se puede ver en la figura 3, al modificar el tamaño del *kernel* se han obtenido diferentes resultados. Esto se debe a que más píxels forman parte de la máscara (como es obvio), y por tanto, se tiene en cuenta una mayor parte de la Gaussiana. Por tanto, existe cierta dependencia entre esto dos valores (el tamaño del *kernel* y el valor de σ). A mayor σ , mayor tendrá que ser el tamaño de la máscara, para así poder modelizar mejor el comportamiento de la función con los píxels vecinos. Si no se adaptase el tamaño del *kernel*, se tendría que solo se coge una parte pequeña de la función Gaussiana más cercana al centro, ignorando el comportamiento de los píxels más lejanos. Tampoco hay que coger un tamaño demasiado grande si el σ es pequeño, ya que entonces habrán muchos píxels que tendrán un valor muy próximo a 0 (aquellos que estén más alejados). En resumen, que hay escoger de forma proporcional el tamaño del *kernel* y el σ .

Para ver cómo afecta el tipo de borde a la imagen resultante, se han realizado una serie de pruebas variando el borde utilizado a la hora de aplicar los *kernels*. Para que los efectos fuesen notables, se ha tenido que escoger un tamaño de máscara mucho más grande a los utilizados anteriormente. Esto se debe a que con una máscara pequeña no se puede apreciar muy bien el resultado, debido a que no se coge mucha región fuera de los bordes de la imagen. El tamaño de la máscara es de 101×101 , permitiendo que se salga lo suficiente de la imagen para ver la influencia. A continuación se pueden ver los resultados que se han obtenido:

101 × 101 Gaussian Blur with $\sigma = 15$ and BORDER_REFLECT

(a) Borde reflejando la imagen.

101 × 101 Gaussian Blur with $\sigma = 15$ and BORDER_CONSTANT

(b) Borde constante.

101 × 101 Gaussian Blur with $\sigma = 15$ and BORDER_DEFAULT

(c) Borde por defecto (reflejado sin contar el primer píxel).

Figura 4: Filtro Gaussiano con diferentes tipos de bordes.

Como se puede ver, existen ciertas diferencias dependiendo de qué tipo de borde se escoja. Si comparamos las figuras 4a y 4c con la figura 4b, se puede ver claramente que los resultados no son los mismos, ya que en este último caso se utiliza un valor constante para las partes más allá de los bordes de las imágenes. Por tanto, la imagen queda como si estuviese enmarcada.

Las diferencias entre las figuras 4a y 4c son mucho más sutiles, pero notables. Se puede ver como por ejemplo en la figura 4c la esquina inferior izquierda es más oscura, mientras que en la figura 4a es más clara. Aunque los dos tipos de borde reflejen la imagen, la forma en la que lo hacen es distinto. *BORDER_DEFAULT* no considera el píxel del borde, mientras que *BORDER_REFLECT* sí que lo hace. Todo esto se puede ver mejor explicado aquí.

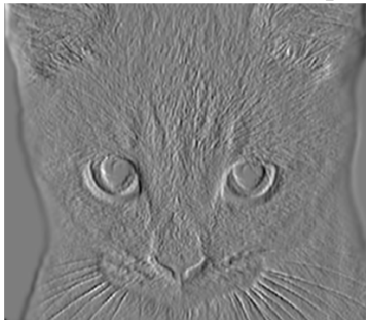
Por último, y antes de pasar al *kernel* de derivada, se ha probado a variar el σ que se ha aplicado en cada eje, manteniendo constante el tamaño del *kernel*. A continuación se pueden ver cuáles han sido los resultados obtenidos:

11 × 11 Gaussian Blur with $\sigma_x = 5$, $\sigma_y = 2$ and BORDER_REPLICATE11 × 11 Gaussian Blur with $\sigma_x = 2$, $\sigma_y = 5$ and BORDER_REPLICATE(a) Filtro Gaussiano con $\sigma_x = 5$ y $\sigma_y = 2$. (b) Filtro Gaussiano con $\sigma_x = 2$ y $\sigma_y = 5$.Figura 5: Filtro Gaussiano variando el σ en cada eje.

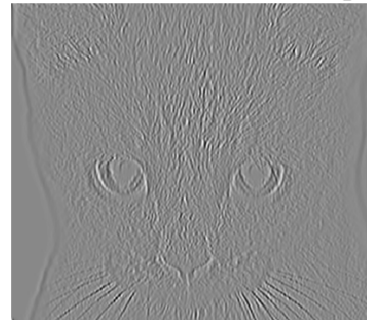
Como se puede ver claramente, hay diferencias, debido a que el alisamiento solo se ha hecho en uno de los ejes en vez de en los dos. Se puede ver como en la figura 5a el pelo de la frente del gato parece que va hacia la derecha, mientras que en la figura 5b el pelo de la frente parece que va hacia abajo. Además, en esta segunda imagen se puede ver como los bigotes se han emborronado más que en el primer caso, donde son más apreciables.

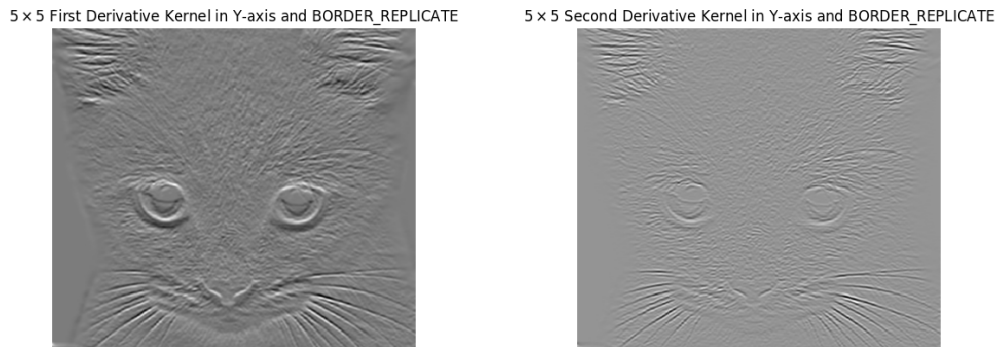
Con esto ya comentado, pasemos a hablar del *kernel* de las derivadas. Lo primero que podemos hacer es ver qué influencia tiene la derivada dependiendo del eje sobre el que se aplique y según el número de veces que se derive en cada uno de los ejes. Para ello, vamos a probar la primera y la segunda derivada en cada uno de los ejes de forma separada (esto es, que no derivaremos a la vez). A continuación se pueden ver los resultados:

5 × 5 First Derivative Kernel in X-axis and BORDER_REPLICATE



5 × 5 Second Derivative Kernel in X-axis and BORDER_REPLICATE

(a) Filtro de primera derivada en el eje X . (b) Filtro de segunda derivada en el eje X .

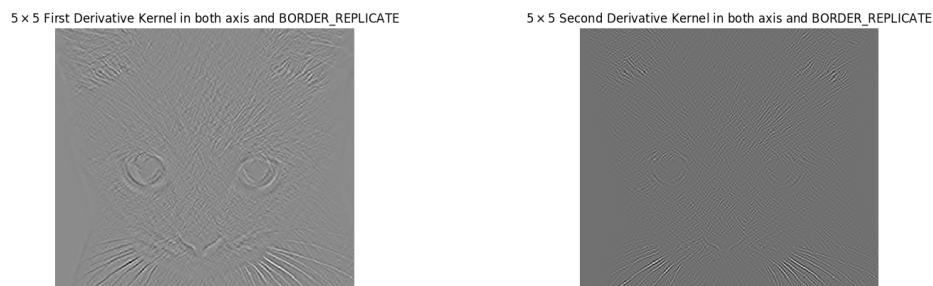


(c) Filtro de primera derivada en el eje Y . (d) Filtro de segunda derivada en el eje Y .

Figura 6: Filtro de derivada variando el número de diferenciaciones en cada eje.

Al pasarlo por el eje X , como se puede ver en las figuras 6a y 6b, se puede ver que se van quedando aquellos bordes verticales, ya que los horizontales se ven muy poco o nada, como es por ejemplo el caso del pelo de las orejas, el cuál casi no se ve, o algunos de los bigotes. En cambio, al pasarlo por el eje Y , tal y como se puede ver en las figuras 6c y 6d, se van quedando aquellos bordes que horizontales. Se puede ver que el pelo en las orejas se puede ver mejor, además de que algunos de los bigote están mejor definidos que en el caso anterior. Lo único que se va perdiendo son los bordes del gato como tal, haciendo que, a medida que se va derivando más en el eje Y , se distinga menos donde empieza el gato y donde está el fondo.

También se ha decidido estudiar qué efecto tiene pasar el *kernel* de las derivadas tanto en el eje X como en el Y a la vez, ya que la función *getDerivKernel* permite especificar el número de diferenciaciones a realizar en cada eje. Los resultados que se han obtenido son los siguientes:



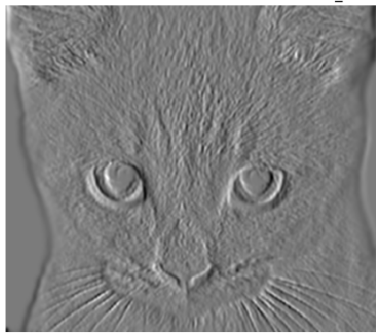
(a) Filtro de primera derivada en ambos ejes. (b) Filtro de segunda derivada en ambos ejes.

Figura 7: Filtro de derivada variando el número de diferenciaciones en los dos ejes.

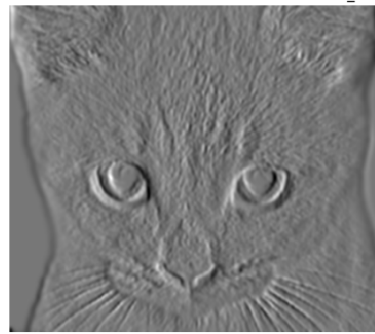
A diferencia de lo que se ve en la figura 6, en la figura 7 se observa que aquellos bordes que destacan más son los que están en diagonal (es decir, una combinación de los dos ejes). Los horizontales y los verticales casi no se notan, como por ejemplo es el caso de los contornos del gato, ya que aquí es incluso más difícil o casi imposible distinguir donde empieza el gato y donde empieza el fondo.

Ahora, tal y como hicimos antes, pasemos a ver qué pasa si mantenemos el número de diferenciaciones constante y aumentamos el tamaño del *kernel*. Para ello, vamos a establecer que en todos los casos se obtiene la primera derivada en el eje *X*. Los resultados de estas variaciones del tamaño del *kernel* se pueden ver a continuación:

7 × 7 First Derivative Kernel in X-axis and BORDER_REPLICATE

(a) Filtro de tamaño 7×7 .

11 × 11 First Derivative Kernel in X-axis and BORDER_REPLICATE

(b) Filtro de tamaño 11×11 .

15 × 15 First Derivative Kernel in X-axis and BORDER_REPLICATE

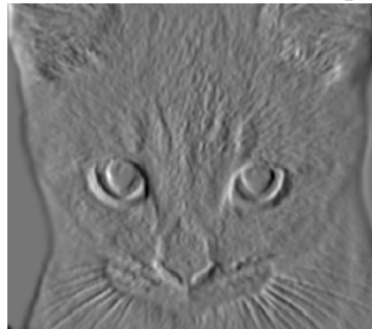
(c) Filtro de tamaño 15×15 .

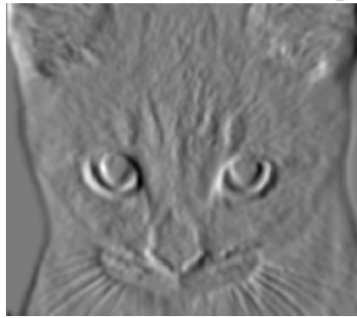
Figura 8: Filtro de primera derivada en el eje *X* variando el tamaño del *kernel*.

Como resultado más obvio, se puede ver que a medida que va aumentando el tamaño del *kernel*, se va emborronando más la imagen. Esto se debe a que uno de

los *kernels* de Sobel hace un suavizado, de forma que se va eliminando el ruido. A mayor tamaño del *kernel*, más se va a emborronar. Por tanto, aquellos bordes muy finos se van a ir perdiendo. Por ejemplo, si comparamos la figura 8a con la figura 8c, podemos ver que los bordes o contornos que se pueden ver en la primera figura en la frente del gato han ido desapareciendo o haciéndose menos notables en esta última. De esta forma, aumentar el tamaño del *kernel* parece que solo nos aporta más suavizado. Un tamaño de *kernel* relativamente pequeño es suficiente para extraer los contornos.

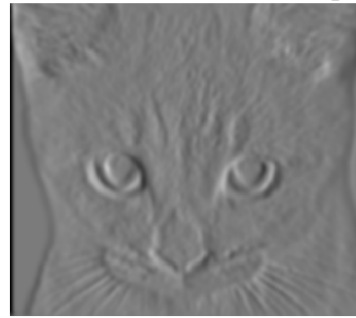
Por último, vamos a ver como variando el tipo de borde cambia la imagen que obtenemos. Para ello, tal y como hicimos antes, vamos a fijar un tamaño de *kernel* relativamente grande para poder apreciar el efecto del borde. En este caso, estamos limitados a un tamaño máximo de 31, ya que OpenCV no permite obtener más. Por tanto, vamos a usar este tamaño de *kernel* a la hora de variar los bordes, ya que se ha considerado el más adecuado para la labor. A continuación se pueden ver los resultados:

31 x 31 First Derivative Kernel in X-axis and BORDER_REFLECT



(a) Borde reflejando la imagen.

31 x 31 First Derivative Kernel in X-axis and BORDER_CONSTANT



(b) Borde constante.

31 x 31 First Derivative Kernel in X-axis and BORDER_DEFAULT



(c) Borde por defecto (reflejado sin contar el primer píxel).

Figura 9: Filtro de primera derivada en el eje X variando el tipo de borde.

En este caso, sucede algo parecido a lo que se podía ver en la figura 4. Podemos ver que la figura más diferente de todas es la figura 9b. Sin embargo, no parece que haya mucha diferencia destacable o notable entre las figuras 9a y 9c, con lo cuál podríamos decir que permiten obtener un resultado más o menos parecido, sin demasiadas diferencias. Parece que en este caso el tipo de borde que más afecta es el constante, aquél en el que se supone un valor constante para todos los píxels fuera de la imagen. Por tanto, si utilizamos cualquier otro tipo de borde no notaríamos mucha diferencia, lo cuál no sucedía en el caso del filtro Gaussiano, ya que sí que había algunas diferencias, aunque fuesen muy pocas.

1.2. Apartado B

2. EJERCICIO SOBRE PIRÁMIDES Y DETECCIÓN DE REGIONES

IMPLEMENTAR funciones para las siguiente tareas:

- A) Una función que genere una representación en pirámide Gaussiana de 4 niveles de una imagen. Mostrar ejemplos de funcionamiento usando bordes y justificar la elección de los parámetros.
- B) Una función que genere una representación en pirámide Laplaciana de 4 niveles de una imagen. Mostrar ejemplos de funcionamiento usando bordes.
- C) Construir un espacio de escalas Laplaciano para implementar la búsqueda de regiones usando el siguiente algoritmo:
 - a. Fijar sigma
 - b. Repetir para N escalas
 - I. Filtrar la imagen con la Laplaciana-Gaussiana normalizada en escala
 - II. Guardar el cuadrado de la respuesta para el actual nivel del espacio de escalas
 - III. Incrementar el valor de sigma por un coeficiente k . (1.2-1.4)
 - c. Realizar supresión de no-máximos en cada escala
 - d. Mostrar las regiones encontradas en sus correspondientes escalas. Dibujar círculos con radio proporcional a la escala.

2.1. Apartado A

2.2. Apartado B

2.3. Apartado C

3. IMÁGENES HÍBRIDAS

Mezclando adecuadamente una parte de las frecuencias altas de una imagen con una parte de las frecuencias bajas de otra imagen, obtenemos una imagen híbrida que admite distintas interpretaciones a distintas distancias (ver hybrid images project page).

Para seleccionar la parte de frecuencias altas y bajas que nos quedamos de cada una de las imágenes usaremos el parámetro sigma del núcleo/-máscara de alisamiento gaussiano que usaremos. A mayor valor de sigma mayor eliminación de altas frecuencias en la imagen convolucionada. Para una buena implementación elegir dicho valor de forma separada para cada una de las dos imágenes (ver las recomendaciones dadas en el paper de Oliva et al.). Recordar que las máscaras 1D siempre deben tener de longitud un número impar.

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). El valor de sigma más adecuado para cada pareja habrá que encontrarlo por experimentación.

1. 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)
2. Realizar la composición con al menos 3 de las parejas de imágenes
3. Construir pirámides gaussianas de al menos 4 niveles con las imágenes resultado. Explicar el efecto que se observa.

3.1. Apartado 1

3.2. Apartado 2

3.3. Apartado 3

4. BONUS

4.1. Convolución 2D propia

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

4.2. Imágenes híbridas a color

Realizar todas las parejas de imágenes híbridas en su formato a color (solo se tendrá en cuenta si la versión de gris es correcta).

4.3. Imagen híbrida propia

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.

Referencias

- [1] Texto referencia
<https://url.referencia.com>