Trabajo 1: Filtrado y Muestreo

Álvaro Fernández García

Octubre, 2018

---

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

   a) El cálculo de la convolución de una imagen con una máscara Gaussiana 2D (Usar GaussianBlur). Mostrar ejemplos con distintos tamaños de máscara y valores de sigma. Valorar los resultados.

   La implementación es bastante sencilla:

```
def Gaussiana(img, sigma, tam=None):
    if tam == None:
        tam = (2*int(3*sigma)+1, 2*int(3*sigma)+1)
    blur = cv2.GaussianBlur(img, tam, sigma)
    return blur
```

   Nótese que el parámetro "tam" es opcional, en caso de que este no se especifique, computamos el tamaño de la máscara para que sea representativo en función del σ especificado. Para ello, empleamos la siguiente expresión: $tam = 2 \cdot \lceil 3 \cdot \sigma \rceil + 1$. El resultado representa la longitud del kernel (en una dimensión) que nos garantiza una buena discretización de la función Gaussiana. Para entenderla mejor, aquí se muestra una representación gráfica:

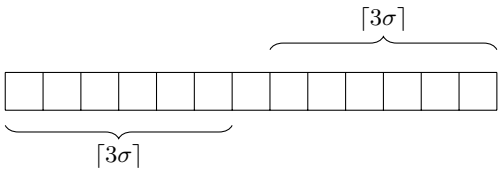


   Figura 1: Ejemplo gráfico de la expresión

   Podríamos haber escogido cualquier otro número para multiplicarlo por σ, no obstante, el 3 es un criterio que da buenos resultados. Por último, aclarar que sumamos 1 por el píxel del centro.

   Para valorar la influencia que tienen tanto el tamaño de σ como el tamaño de la máscara en un filtro Gaussiano, se han realizado tres experimentos:

   1) Filtro Gaussiano con σ variable, pero tamaño fijo: En una distribución gaussiana, el valor de σ determina como de ancha es la campana. A mayor σ mayor anchura.

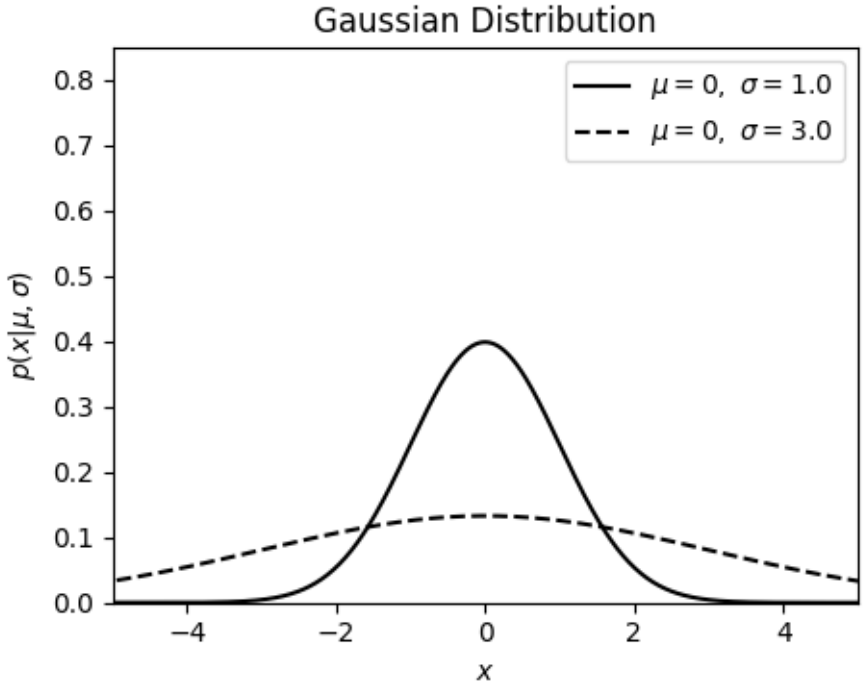


Figura 2: Ejemplo de la influencia de σ en la distribución gaussiana

Cuando se construye un kernel, se muestrea la función para determinar los pesos de cada posición. Si σ toma valores pequeños, los pesos de las posiciones cercanas al píxel central serán mucho mayores que los extremos (por no decir que en los extremos serán casi nulos). Si por el contrario, el valor de σ crece, la diferencia entre los pesos centrales y los extremos no será tan acusada (basta con mirar la gráfica para justificarlo).

En definitiva, cuanto mayor es el σ, los pesos en los extremos de la máscara no serán tan insignificantes, y en consecuencia esos píxeles influirán más al píxel central durante el suavizado (siempre que el tamaño del kernel sea el apropiado). Para demostrarlo, aquí se muestra un ejemplo:

Figura 3: k = 19, σ = 1 y 3 respectivamente

En la primera imagen el σ es demasiado pequeño para la máscara, los pesos en los extremos son casi nulos y la imagen apenas se suaviza. En la segunda, σ es el valor apropiado para el tamaño de la máscara, los pesos se distribuyen correctamente y el

alisado es palpable.

2) Filtro gaussiano con tamaño variable, pero σ fijo: como llevamos diciendo a lo largo de este apartado, el tamaño del kernel debe de garantizarnos una buena discretización para el σ escogido. En el siguiente ejemplo se muestra como actuaría un tamaño demasiado pequeño para un σ grande: al ser pequeña, la máscara sólo reúne los valores de la gaussiana cercanos al centro. Da igual que los pesos en los extremos no sean prácticamente nulos, el tamaño de la máscara no los tiene en cuenta. Además también debemos de considerar el hecho de que, cuanto más pequeño sea el kernel, menos píxeles del entorno influirán en el valor del píxel central:

Figura 4: $\sigma = 3$, k = 5 y 19 respectivamente

Nótese que en el segundo caso el valor de k es un tramaño razonable y sí es apropiado para el σ escogido.

3) Filtro gaussiano con ambos parámetros libres: En definiva cuando se desea aplicar un filtro guassiano no se pueden considerar ambos parámetros de forma independiente sino que deben considerarse en conjunto, sobre todo, que el tamaño de la máscara acompañe al σ escogido y viceversa. Aquí se muestra el último ejemplo, donde hemos utilizado la expresión mencionada al principio de este apartado para calcular el tamaño apropiado para cada σ:

Figura 5: $\sigma = 0.8$, 2.5, 5 y 9 respectivamente

Nota: 0.8 es el mínimo valor de σ para el que se puede discretizar la función gaussiana apropiadamente.

b) Usar getDerivKernels para obtener las máscaras 1D que permiten calcular al convolución 2D con máscaras de derivadas. Representar e interpretar dichas máscaras 1D para distintos valores de sigma.

Entiendo que el enunciado contiene una errata, y que no se refiere a distintos valores de sigma, sino distintos tamaños de máscara.

Para realizar este ejercicio se ha implementado una función que obtiene los distintos kernels derivados (primera y segunda derivada, derivada solo en X y derivada solo en Y), realiza el producto $kernelY \cdot kernelX^T$ para obtener la matriz y posteriormente la imprime por pantalla.

Para facilitar la interpretación utilizaré una máscara pequeña (tamaño 3):

$$\text{Derivada solo en X} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

La derivada solo en X nos permite identificar en la imagen los bordes verticales. Al multiplicar la parte izquierda del píxel central por valores negativos, el centro por 0, y la parte derecha por números positivos, estamos calculando la variación de intensidad en la imagen en torno a ese píxel ($pixel_{siguiente} - pixel_{anterior}$). Si existe una variación, entonces habrá un borde vertical. Además podemos observar que, a la hora de calcular la diferencia, tiene un mayor peso la fila del centro.

$$\text{Derivada solo en Y} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Es exactamente lo mismo que en el caso anterior, pero ahora lo utilizamos para detectar los bordes horizontales. Estos dos últimos casos constituyen el operador de Sobel. Si los aplicamos por separado y luego combinamos ambos resultados obtendremos todos los bordes de la imagen.

$$\text{Primera Derivada} = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$$

Con esta máscara podemos obtener los bordes (tanto horizontales como verticales) de la imagen en una sola pasada. En este caso, los valores de los píxeles alineados con el centro no influyen, pero sí los de las esquinas. Además, se calcula la diferencia de intensidad en ambos sentidos ([-1, 0, 1] y [1, 0, -1]) tanto en vertical como en horizontal.

$$\text{Segunda Derivada} = \begin{bmatrix} 1 & -2 & 1 \\ -2 & 4 & -2 \\ 1 & -2 & 1 \end{bmatrix}$$

El kernel 1D que nos devuelve la segunda derivada es [1, -2, 1], si seguimos la misma filosofía de antes, al aplicar la convolución tendríamos:

$$pixel_{siguiente} - 2 \cdot pixel_{actual} + pixel_{anterior} \tag{1}$$
$$= \quad pixel_{siguiente} - pixel_{actual} - pixel_{actual} + pixel_{anterior} \tag{2}$$
$$= \quad (pixel_{siguiente} - pixel_{actual}) - (pixel_{actual} - pixel_{anterior}) \tag{3}$$

Como vemos, la segunda derivada no utiliza directamente la diferencia de intensidad para detectar los bordes, sino que utiliza la "diferencia de diferencias".

La matriz que observamos se construye al multiplicar $\begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & -2 & 1 \end{bmatrix}$.

Para terminar, aquí se muestra un ejemplo de cómo serían las matrices con tamaño 5:

$$\text{Derivada solo en X: } \begin{bmatrix} -1 & -2 & 0 & 2 & 1 \\ -4 & -8 & 0 & 8 & 4 \\ -6 & -12 & 0 & 12 & 6 \\ -4 & -8 & 0 & 8 & 4 \\ -1 & -2 & 0 & 2 & 1 \end{bmatrix}$$

$$\text{Derivada solo en Y: } \begin{bmatrix} -1 & -4 & -6 & -4 & -1 \\ -2 & -8 & -12 & -8 & -2 \\ 0 & 0 & 0 & 0 & 0 \\ 2 & 8 & 12 & 8 & 2 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

$$1^{\text{a}} \text{ derivada: } \begin{bmatrix} 1 & 2 & 0 & -2 & -1 \\ 2 & 4 & 0 & -4 & -2 \\ 0 & 0 & 0 & 0 & 0 \\ -2 & -4 & 0 & 4 & 2 \\ -1 & -2 & 0 & 2 & 1 \end{bmatrix}$$

$$2^{\text{a}} \text{ derivada: } \begin{bmatrix} 1 & 0 & -2 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ -2 & 0 & 4 & 0 & -2 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & -2 & 0 & 1 \end{bmatrix}$$

Como el kernel es más grande, más píxeles intervienen a la hora de calcular la variación de intensidad y, como resulta lógico, cuanto más nos alejamos del centro, menos importancia tienen en el cálculo. Por lo demás, el funcionamiento es el mismo.

También resulta interesante decir que los bordes que detectemos con kernels más grandes serán más gruesos, debido a que el intervalo en el que calculamos la diferencia es más amplio.

c) Usar la función Laplacian para el cálculo de la convolución 2D con una máscara de Laplaciana-de-Gaussiana de tamaño variable. Mostrar ejemplos de funcionamiento usando dos tipos de bordes y dos valores de sigma: 1 y 3.

Para aplicar una máscara Laplaciana-de-Gaussiana en OpenCV hay que suavizar primero la imagen con una máscara Gaussiana (eliminando así el ruido) y a continuación aplicar el filtro Laplaciano para obtener los límites de la imagen.

No obstante, antes de nada y para ver la influencia de los bordes en el proceso, se los añadiremos manualmente a la imagen original. Además, posteriormente en la función Laplacian se utilizará el mismo borde (este procedimiento se aplicará igualmente para todos los apartados en los que intervengan distintos tipos de bordes).

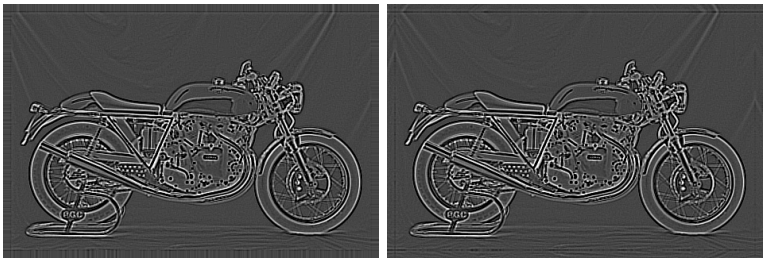
Por último le sumaremos una constante (delta) a todos los píxeles de la imagen para que se aprecien mejor los límites. El código es el siguiente:

```
def LaplacianaGaussiana(img, sigma, ksize, border, delta):
    # Anadimos el borde:
    out = cv2.copyMakeBorder(img, 10, 10, 10, 10, border)
    # Suavizamos con la Gaussiana:
    out = cv2.GaussianBlur(out, (ksize, ksize), sigma,
                           borderType=border)
    # Aplicamos la laplaciana:
    out = cv2.Laplacian(src=out, ksize=ksize, borderType=border,
                        ddepth=-1, delta=delta)
    return out
```

Aquí se muestran ejemplos de su funcionamiento

(a) Bordes Replicados (b) Bordes Reflejados

Figura 6: Laplaciana-de-Gaussiana con $\sigma = 1$



(a) Bordes Replicados (b) Bordes Reflejados

Figura 7: Laplaciana-de-Gaussiana con $\sigma = 3$

La diferencia entre utilizar como sigma 1 o 3 es prácticamente inapreciable, pero tiene su explicación: en ambos casos estamos utilizando 3 como tamaño del kernel y en consecuencia, por lo que hemos explicado en el primer apartado, un tamaño de 3 no es nada

apropiado para un simga de 3, lo que se traduce en que su efecto no es apreciable. ¿Y por qué no aumentar el tamaño? Pues porque eso haría que los límites encontrados por la Laplaciana sean más gruesos (y confusos) y no debemos olvidar que el objetivo de la Laplaciana no es suavizar apropiadamente la imagen, sino encontrar los límites, cosa que se consigue bien con este tamaño.

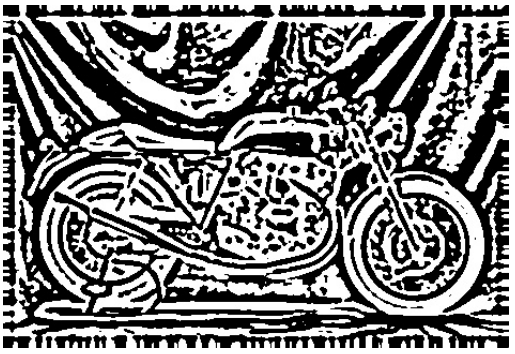


Figura 8: Ejemplo del resultado de una Laplaciana con un kernel de tamaño 19

Por último decir que la influencia de los bordes es inapreciable en los resultados más allá de que, evidentemente, son distintos.

2. IMPLEMENTAR apoyándose en las funciones getDerivKernels, getGaussianKernel, pyrUp(), pyrDown():

   a) El cálculo de la convolución 2D con una máscara separable de tamaño variable. Usar bordes reflejados. Mostrar resultados

   La implementación es simple, primero añadimos el borde a la imagen y a continuación aplicamos con sepFilter2D los kernels pasados como argumento (utilizando en dicha función el mismo borde). El código es el siguiente:

```
def Convolucion2DSeparable(img, kernelx, kernely):
  # Anadir borde reflejado:
  out = cv2.copyMakeBorder(img, 10, 10, 10, 10, cv2.BORDER_REFLECT)
  # Aplicamos la mascara separable:
  out = cv2.sepFilter2D(src=out, kernelX=kernelx, kernelY=kernely,
                        ddepth=-1, borderType=cv2.BORDER_REFLECT)
  return out
```

   Para verificarlo he utilizado un kernel gaussiano, el cual es separable por definición:

$$G_\sigma(x,y) = \frac{1}{2\pi\sigma^2} exp^{-\frac{x^2+y^2}{2\sigma^2}} = (\frac{1}{\sqrt{2\pi}\sigma} exp^{-\frac{x^2}{2\sigma^2}})(\frac{1}{\sqrt{2\pi}\sigma} exp^{-\frac{y^2}{2\sigma^2}})$$

   En las pruebas realizadas se ha calculado el tamaño de la máscara con la expresión del primer apartado del Ejercicio 1. Como valores de sigma he utilizado también los mismos para comprobar si se obtenían resultados equivalentes:

Figura 9: $\sigma = 0.8$, 2.5, 5 y 9 respectivamente

Efectivamente, los resultados son idénticos. Con esto queda demostrado que cuando se dispone de un filtro separable, resulta equivalente aplicar las máscaras 1D por filas y por columnas que aplicar directamente la máscara 2D, ahorrándose de esta forma tiempo de cómputo.

Por último, al igual que en el ejercicio anterior, la influencia del uso de bordes es prácticamente inapreciable.

b) El cálculo de la convolución 2D con una máscara 2D de 1a derivada de tamaño variable. Mostrar ejemplos de funcionamiento usando bordes a cero.

El código de este apartado es el siguiente:

```
def Convolucion2DDerivada(img, dx, dy, k):
  # Anadir bordes a 0
  out = cv2.copyMakeBorder(img, 5, 5, 5, 5, 0)
  # Suavizamos para eliminar el ruido:
  #(si sigma < 0 se calcula en funcion de k)
  out = cv2.GaussianBlur(out, (k,k), -1, borderType=0)
  # Obtenemos los kernels derivados 1D:
  kx, ky = cv2.getDerivKernels(dx=dx, dy=dy, ksize=k, normalize=False)
  # Aplicamos la convolucion:
  out = cv2.sepFilter2D(src=out, kernelX=kx, kernelY=ky, ddepth=-1,
                        borderType=0, delta=70)
  return out
```

Antes de nada, añadimos los bordes negros a la imagen. Seguidamente, la función aplica un suavizado Gaussiano para eliminar el ruido. Veáse que como sigma hemos especificado un -1, esto nos sirve para indicarle a OpenCV que calcule el valor más apropiado de sigma para el tamaño de máscara especificado. Concretamente, utiliza la siguiente expresión:

$$0{,}3 \cdot ((ksize - 1) \cdot 0{,}5 - 1) + 0{,}8$$

A continuación obtenemos los kernels derivados de los órdenes y tamaño especificados y los aplicamos de forma separada a la imagen mediante sepFilter2D (utilizando bordes a 0 y añadiendole un delta a los píxeles para que se aprecien mejor los límites). Es cierto que el enunciado nos dice que realicemos la convolución con una máscara 2D, no obstante no tiene mucho sentido desaprovechar la oportunidad que nos ofrecen los kernels derivados de realizar la convolución de forma separada, de ahí que haya decidido utilizar sepFilter2D.

A continuación tenemos ejemplos de su funcionamiento utilizando distintos tamaños de máscara:

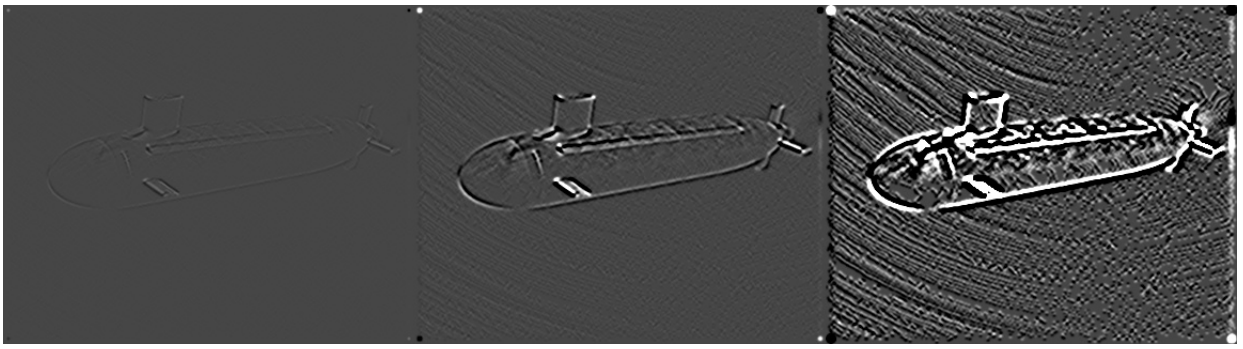


Figura 10: Primera derivada con k = 3, 5 y 7 respectivamente

Como podemos observar, al igual que pasaba en la Laplaciana, cuanto mayor es el tamaño del kernel, más gruesos son los límites que nos devuelve. Con 7 es demasiado acusado y con 3 demasiado pobre, por lo que el tamaño apropiado para este caso sería de 5.

Con respecto a los bordes, en este caso sí que se aprecia su ifluencia: podemos observar la aparición de unos puntos blancos y negros en las esquinas, justo en el límite del borde. Esto se debe a que existe una gran diferencia de intensidad (que es lo que capta la derivada) entre el borde negro y el comienzo de la imagen (que obviamente contiene ruido), producciendo la aparición de esas formas.

c) El cálculo de la convolución 2D con una máscara 2D de 2a derivada de tamaño variable.

Para este ejercicio se ha utilizado la función explicada en el apartado anterior, pero especificando que queremos los kernels de segunda derivada. A continuación se muestran ejemplos:

Figura 11: Segunda derivada con k = 3, 5 y 7 respectivamente

Como principal diferencia respecto de la primera derivada, se puede observar que la segunda requiere de un tamaño de kernel mucho más grande para que su efecto sea palpable, por todo lo demás tiene un comportamiento similar, incluída la aparición de puntos negros y blancos debido a los bordes a 0.

d) Una función que genere una representación en pirámide Gaussiana de 4 niveles de una imagen. Mostrar ejemplos de funcionamiento usando bordes.

Una pirámide Gaussiana no es más que el resultado de la aplicación repetida de un suavizado gaussiano a una imagen, seguido de un submuestreo. Para realizar esto en OpenCV basta con aplicar tantas veces como se desee la función pyrDown.

El código es el siguiente:

```
def PiramideGaussiana(img, tam, borde):
    # Anadir el borde:
    img = cv2.copyMakeBorder(img, 10, 10, 10, 10, borde)

    # Calcular la piramide
    out = [img]
    for _ in range(tam-1):
        img = cv2.pyrDown(img)
        out.append(img)
    return out
```

A continuación se muestran dos ejemplos de pirámide Gaussiana utilizando dos tipos de bordes:

Figura 12: Pirámide Gaussiana de cuatro niveles con bordes reflejados

Figura 13: Pirámide Gaussiana de cuatro niveles con bordes replicados

Al igual que en otros casos de este documento, la influencia de aplicar un tipo u otro de borde es prácticamente inapreciable. También he considerado que la imagen original cuenta como un nivel de la pirámide.

e) Una función que genere una representación en pirámide Laplaciana de 4 niveles de una imagen. Mostrar ejemplos de funcionamiento usando bordes.

Cada nivel de la pirámide Laplaciana representa la diferencia entre la imagen sobremuestrada de un nivel de la pirámide Gaussiana y el propio nivel de la pirámide Gaussiana a excepción del último nivel, el cual se corresponde con el último nivel de la pirámide Gaussiana. Esto último permite la reconstrucción de la imagen original usando las diferencias de los niveles superiores (ya que normalmente, esta técnica se utiliza para compresión de imágenes).

El código que calcula la pirámide Laplaciana es el siguiente:

```
def PiramideLaplaciana(img, tam, borde):
  out = []
  img = cv2.copyMakeBorder(img, 10, 10, 10, 10, borde)
  current = img

  # Calculamos la laplaciana:
  # tam-1 porque el ultimo nivel es el de la gaussiana
  for _ in range(tam-1):
    down = cv2.pyrDown(current)
    up = cv2.pyrUp(down, dstsize=(current.shape[1], current.shape[0]))
    tmp = cv2.subtract(current, up)
    out.append(tmp)
    current = down

  # Anadir el ultimo nivel de la gaussiana
  out.append(down)
  return out
```

Aquí se muestran dos ejemplos de pirámide Laplaciana usando dos tipos de bordes:

Figura 14: Pirámide Laplaciana de cuatro niveles con bordes reflejados

Figura 15: Pirámide Laplaciana de cuatro niveles con bordes replicados

Al igual que en el apartado anterior, he considerado que la imagen original cuenta como un nivel de la pirámide y en este caso sí que resulta imposible observar la influencia de los bordes.

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. 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.

a) 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)

b) Realizar la composición con al menos 3 de las parejas de imágenes

La función que calcula las tres imágenes (alta, baja e híbrida) es la siguiente:

```
def HybridImages(low, high, sigma_low, sigma_high):
    low_frec = Gaussiana(img=low, sigma=sigma_low)
    high_frec = Gaussiana(img=high, sigma=sigma_high)
    high_frec = cv2.subtract(high, high_frec)
    H = cv2.add(low_frec, high_frec)
    return H, low_frec, high_frec
```

Como vemos, para calcular tanto las frecuencias bajas como las frecuencias altas se utiliza la función Gaussiana elaborada en el primer apartado del primer ejercicio. Para obtener las frecuencias altas, simplemente suavizamos la imagen y a continuación le restamos a la original el resultado del suavizado. Cuando se aplica un filtro de paso bajo, se eliminan las frecuencias altas, dejando únicamente las bajas. Si a la imagen original le restamos sólo las frecuencias bajas, el resultado será una imagen en la que quedan únicamente las frecuencias altas.

A continuación, para obtener la imagen híbrida, sumamos ambas imágenes (alta y baja). Para realizar las operaciones con las imágenes se han utilizado las funciones propias de OpenCV, de esta forma nos evitamos los problemas de overflow que pueden surgir al realizar operaciones con enteros sin signo.

En este ejercicio lo complejo es hallar por experimentación los sigmas y tamaños de máscaras apropiados para cada par de imágenes. He decidido no especificar el tamaño de la máscara para ninguno de los dos casos y dejar que se calculen en función del sigma especificado. De esta forma, sólo tendremos que preocuparnos de ajustar apropiadamente los valores de sigma.

Por último, para visualizar en una misma ventana las imágenes alta, baja e híbrida, se ha utilizado la función PintaMI elaborada en el Trabajo 0.

A continuación se muestran las imágenes híbridas realizadas:

Figura 16: $\sigma_{low} = 10$, $\sigma_{high} = 5$

Figura 17: $\sigma_{low} = 6$, $\sigma_{high} = 2$

Figura 18: $\sigma_{low} = 6$, $\sigma_{high} = 3$

Figura 19: $\sigma_{low} = 6$, $\sigma_{high} = 2$

Bonus

1. Cálculo del vector máscara Gaussiano: Sea $fx) = exp(-0{,}5\frac{x^2}{\sigma^2})$ una función donde σ (sigma) representa un parámetro en unidades píxel. Implementar una función que tomando sigma como parámetro de entrada devuelva una máscara de convolución 1D representativa de dicha función. Justificar los pasos dados (Ayuda: comenzar calculando la longitud de la máscara a partir de sigma y recordar que el valor de la suma de los valores del núcleo es 1)

   El código es el siguiente:

```
# Definimos la funcion:
def f(x, sigma):
    return math.exp(-0.5 * ((x**2)/(sigma**2)))

# Funcion que devuelve la mascara Gaussiana:
def myGaussianKernel(sigma):
    # Calculamos el tamanio:
    k = int(3 * sigma)
    # Creamos la mascara:
    mask = [f(x,sigma) for x in range(-k, k+1)]
    mask = np.array(mask)
    # Para que los valores sumen 1:
    mask = mask / np.sum(mask)
    return mask
```

   Comenzamos en primer lugar calculando el tamaño que debería tener la máscara para discretizar apropiadamente el σ especificado (Para ello, hacemos uso de la expresión explicada en el primer ejercicio de esta práctica).

   A continuación, rellenamos una lista con los valores de f(x), desde -tamaño hasta tamaño+1 (el +1 por el píxel del centro), convertimosla lista a un array de numpy y, para asegurarnos de que todos los valores de la máscara suman 1 (ya que para que sea un filtro de suavizado, todos los valores deben ser positivos y deben sumar 1), dividimos elemento a elemento entre la sumatoria total del vector.

2. Implementar una función que calcule la convolución 1D de un vector señal con un vector-máscara de longitud inferior al de la señal, usar condiciones de contorno reflejada. La salida será un vector de igual longitud que el vector señal de entrada. (Ayuda: En el caso de que el vector de entrada sea de color, habrán de extraerse cada uno de los tres vectores correspondientes a cada una de las bandas, calcular la convolución sobre cada uno de ellos y volver montar el vector de salida. Usar las funciones split() y merge()) de OpenCV).

   En primer lugar, he elaborado dos funciones que añaden y quitan los bordes reflejados a la señal:

```
# Funcion para aniadir los bordes reflejados:
def myAddBorders(img, size):
 return np.concatenate((img[:size][::-1], img, img[len(img)-size:][::-1]))

# Funcion para eliminar los bordes:
def myRemoveBorders(img, size):
 return img[size:len(img)-size]
```

   Simplemente le añade a la imagen los extremos del tamaño especificados pero invertidos. Para eliminarlos, devolvemos una copia solo de los elementos comprendidos entre los bordes.

   El siguiente paso es hacer una función que realiza la convolución 1D con un vector señal de un solo canal. Aquí se muestra la función bien comentada y explicada:

```
def convolve1Dvector(signal, kernel):
    # Verificamos que se cumplan las condiciones:
    assert kernel.shape[0] < signal.shape[0]
    assert len(kernel) %2 != 0

    # Aniadimos los bordes:
    borderLen = kernel.shape[0] // 2
    signal = myAddBorders(signal, borderLen)

    # Creamos la salida:
    salida = np.zeros(signal.shape, signal.dtype)

    # Ahora aplicamos la convolucion:
    for i in range( signal.shape[0] - kernel.shape[0] + 1 ):
        newValue = np.sum( signal[i:i+kernel.shape[0]] * kernel )
        salida[i+borderLen] = newValue

    # Eliminamos los bordes:
    signal = myRemoveBorders(signal, borderLen)
    salida = myRemoveBorders(salida, borderLen)

    return salida
```

   Por último, para tener en cuenta las señales a color, se ha diseñado una función de más alto nivel que, internamente, descompone la señal en los tres canales, llama a la función anterior para cada uno de ellos y luego los junta para devolver la salida. En caso de que tenga un solo canal, se llama a la función anterior una sola vez:

```
def myConvolution1D(signal, kernel):
    if len(signal.shape) == 3:
        channels = cv2.split(signal)
        channels[0] = convolve1Dvector(channels[0].flatten(), kernel)
        channels[1] = convolve1Dvector(channels[1].flatten(), kernel)
        channels[2] = convolve1Dvector(channels[2].flatten(), kernel)
        out = cv2.merge(channels)
    else:
        out = convolve1Dvector(signal, kernel)

    return out
```

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

   Teniendo implementada la función convolve1Dvector del apartado anterior, realizar este ejercicio es bastante sencillo. Quizás lo más complejo es que, en el caso de que la imagen dada como argumento sea a color, se debe de realizar la convolución de cada uno de los canales por separado. A continuación se muestra el código de la función implementada:

```
def mySeparable2DConvolution(src, kernelX, kernelY):
    # Separamos los canales:
    channels = cv2.split(src)
    convolved_channels = []

    # Para cada canal aplicamos la convolucion
    for ch in channels:
        # Creamos una imagen temporal para la convolucion:
        tmp1 = np.zeros(ch.shape, ch.dtype)

        # Aplicar kernelX por filas:
        for i in range(tmp1.shape[0]):
            tmp1[i] = convolve1Dvector(ch[i], kernelX)

        # Transponemos para hacer la convolucion por columas
        tmp1 = tmp1.transpose()

        # Creamos la segunda imagen temporal para la convolucion:
        tmp2 = np.zeros(tmp1.shape, ch.dtype)

        # Aplicar kernelY por columnas:
        for j in range(tmp2.shape[0]):
            tmp2[j] = convolve1Dvector(tmp1[j], kernelY)

        # Deshacemos el cambio
        tmp2 = tmp2.transpose()

        convolved_channels.append(tmp2)

    salida = cv2.merge(convolved_channels)

    return salida
```

Nota: en caso de que src sea una imagen de un solo canal (escala de grises), la función cv2.split() devolverá un solo vector y en consecuencia, el bucle se ejecutará una sola vez.

Para cada canal de la imagen se aplica el siguiente proceso:

- Realizamos la convolución 1D de cada una de las filas de la imagen con el kernelX, y almacenmos el resultado en una imagen temporal.
- Transponemos esa imagen temporal para que las columnas puedan ser tratadas como filas.
- Aplicamos la convolución 1D nuevamente de cada una de las filas (que realmente son las columnas) de la imagen temporal transpuesta con el kernelY y almacemos el resultado en una segunda imagen temporal.
- Volvemos a transponer la segunda imagen temporal para dejarla como estaba.

Una vez aplicado esto para todos los canales, se vuelven a juntar y se devuelve el resultado.

Aquí se muestran dos ejemplos de su funcionamiento (uno gris y otro a color), en los que se han suavizado las imágenes utilizando el kernel gaussiano implementado en el primer ejercicio del bonus con un $\sigma = 3$:

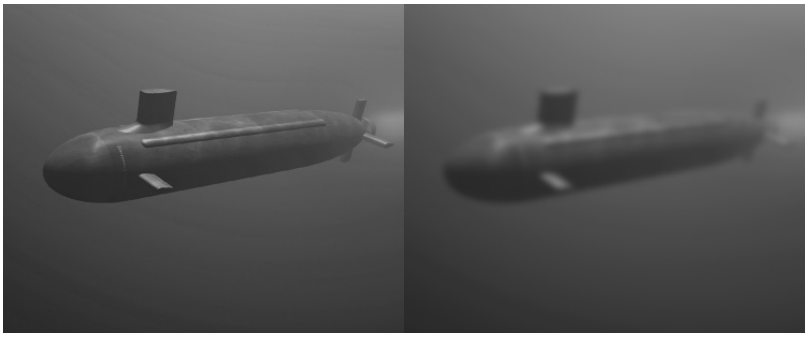


Figura 20: Convolución 2D de una imagen en escala de grises

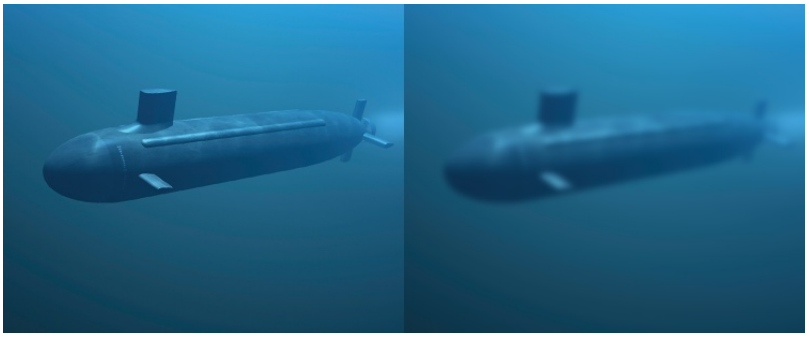


Figura 21: Convolución 2D de una imagen a color

4. Construir una pirámide Gaussiana de al menos 5 niveles con las imágenes híbridas calculadas en el apartado anterior. Mostrar los distintos niveles de la pirámide en un único canvas e interpretar el resultado. Usar implementaciones propias de todas las funciones usadas.

Aquí se muestra el código implementado:

```
def myGaussianPyr(img, levels):
    mask = myGaussianKernel(3)

    # Calcular la piramide
    out = [img]
    for _ in range(levels-1):
        # Estas dos lineas son equivalentes a cv2.pyrDown()
        img = mySeparable2DConvolution(img, mask, mask)
        # Nos quedamos con las filas y columnas pares
        img = img[::2, ::2]
        out.append(img)

    return out
```

Primero suavizamos la imagen con un kernel gaussiano de sigma 3 y a continuación, para bajar la resolución, nos quedamos únicamente con las filas y columnas pares.

Aquí se muestra una pirámide gaussiana de 5 niveles sobre una de las imágenes híbridas de apartados anteriores:

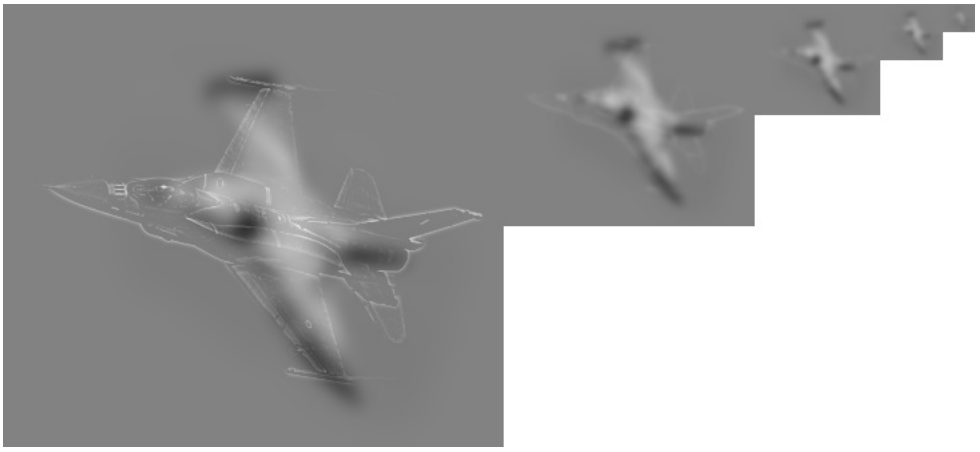


Figura 22: Implementación propia de una pirámide gaussiana de 5 niveles

Como vemos, a medida que la imagen se hace más y más pequeña, deja de apreciarse el avión. En definitiva, tiene el mismo efecto que si nos alejásemos de la imagen. Esto tiene una explicación, y es que para computar un nuevo nivel de la pirámide, se suaviza y se baja su resolución, es decir, cada vez son menos apreciables las frecuencias altas en la imagen.

El avión está representado por las frecuencias altas, si no podemos verlas (ya sea porque la imagen tiene una resolución más pequeña, o porque se va alisando por el suavizado), tampoco podremos apreciar su figura.

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

El código es similar al que se realizó para su versión en escala de grises, con la diferencia de que todos los métodos utilizados están implementados manualmente (a excepción de cv2.split() y cv2.merge())y que además, el proceso a color, requiere que se calcule la hibridación para cada uno de los canales individualmente para posteriormente unirlos de nuevo en el resultado. El código es el siguiente:

```
def myHybridImagesColor(low, high, sigma_low, sigma_high):
    # Declaramos las variables que necesitaremos:
    mask_low = myGaussianKernel(sigma_low)
    mask_high = myGaussianKernel(sigma_high)
    Hyb = []
    Low = []
    Hig = []

    # Separamos los canales:
    low_channels = cv2.split(low)
    high_channels = cv2.split(high)

    # Para cada canal repetimos el siguiente proceso:
    for lch, hch in zip(low_channels, high_channels):
        # Calculamos la img hibrida en ese canal
        # Suavizamos
        low_frec = mySeparable2DConvolution(lch, mask_low, mask_low)
        high_frec = mySeparable2DConvolution(hch, mask_high, mask_high)

        # Hacemos los calculos en coma flotante para evitar problemas:
        high_frec = hch.astype(np.float64) - high_frec.astype(np.float64)
        hibrida = low_frec.astype(np.float64) + high_frec

        # Volvemos a convertir a uint8
        hibrida = np.clip(hibrida, 0, 255).astype(np.uint8)
        high_frec = np.clip(high_frec, 0, 255).astype(np.uint8)

        # Guardamos la informacion calculada para cada imagen
        Hyb.append(hibrida)
        Low.append(low_frec)
        Hig.append(high_frec)

    # Juntamos todos los canales:
    Hyb = cv2.merge(Hyb)
    Low = cv2.merge(Low)
    Hig = cv2.merge(Hig)

    return Hyb, Low, Hig
```

Para finalizar, aquí se muestran las imágenes híbridas realizadas: