
Visión por Computador

Práctica 1: Filtrado y muestreo

Francisco Solano López Rodríguez

Funciones usadas para lectura y representación de imágenes.

```
def read_image(filename, flagColor):
    if flagColor == True:
        return cv2.imread(filename, cv2.IMREAD_COLOR)
    else:
        return cv2.imread(filename, cv2.IMREAD_GRAYSCALE)

def display_image(im):
    cv2.imshow('Imagen', im)

    key = -1

    while(key != 13):
        key = cv2.waitKey(0)

    cv2.destroyAllWindows()

black = [0,0,0]
white = [255,255,255]
font = cv2.FONT_HERSHEY_PLAIN

def display_multiple_images(vim, title = None, col = 3, color = white):
    width_rows = []
    height_rows = []

    for i in range(0,len(vim)):
        vim[i] = cv2.copyMakeBorder(vim[i],20,0,4,4,cv2.BORDER_CONSTANT,value=color)

        if title != None:
            cv2.putText(vim[i],title[i],(10,15), font, 1,(0,0,0), 1, 0)

    for i in range(0,len(vim),col):
        width_rows.append(sum( [ im.shape[1] for im in vim[i:min(i+col,len(vim))]] ))
        height_rows.append(max( [ im.shape[0] for im in vim[i:min(i+col,len(vim))]] ))

    # ancho total del mapa de imagenes
    width = max(width_rows)
    # altura total del mapa de imagenes
    height = sum(height_rows)

    # creación de la matriz para el mapa de imagenes
    image_map = np.zeros( (height, width,3) , np.uint8)

    # rellenamos el mapa
    inicio_fil = 0
    for i in range(0,len(vim),col):
        inicio_col = 0
        # rellenamos una fila de imagenes
        for k in range(i,min(i+col,len(vim))):
            actual_image = vim[k]

            if len(actual_image.shape) < 3:
                actual_image = cv2.cvtColor(actual_image, cv2.COLOR_GRAY2RGB)

            for row in range(actual_image.shape[0]):
                for col in range(actual_image.shape[1]):
                    image_map[inicio_fil+row][inicio_col + col] = actual_image[row][col]
            inicio_col += actual_image.shape[1]
        inicio_fil += height_rows[i//col]

    # Visualization
    display_image(image_map)
```

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.

```
def gaussian_convolution(img, size, sigma):
    """
    Esta función aplica a la imagen pasada un filtro gaussiano
    con el size y sigma pasados.
    """

    img_gb = cv2.GaussianBlur(img,(size,size),sigma)
    return img_gb

# Different sigma values
images_sigma = [image]
titles_sigma = ['Original']

for i in range(1,6):
    sigma = i*1.0
    images_sigma.append(gaussian_convolution(image, 0, sigma))
    titles_sigma.append('Sigma = ' + str(sigma))

display_multiple_images(images_sigma, titles_sigma, 3)
```

Se ha probado tanto variar el valor de sigma como el valor del tamaño. Para variar el valor de sigma se ha fijado el valor de size a 0, esto hace que el valor de size se calcule internamente a partir del valor de sigma.

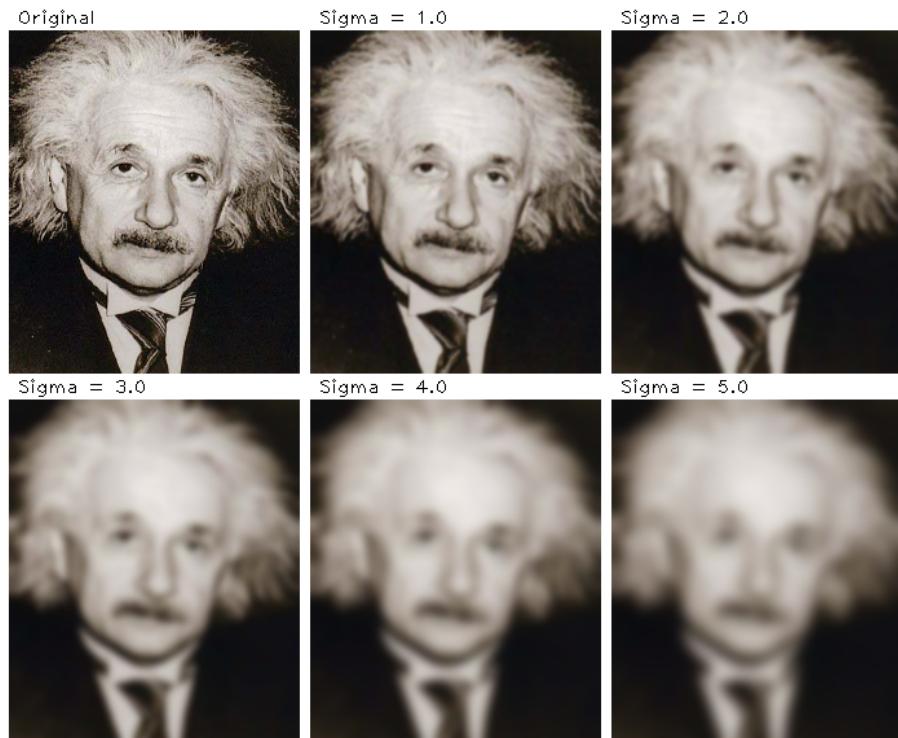


Figura 1: Filtro gaussiano con diferentes valores de sigma

```
# Different size values
images_size = [image]
titles_size = ['Original']

for i in range(1,6):
    size = i*6+1
    images_size.append(gaussian_convolution(image, size, 0))
    titles_size.append('Size = ' + str(size))
```

```
display_multiple_images(images_size, titles_size, 3)
```

Para variar el valor de size se ha pasado el valor 0 a sigma, de esta forma el valor de sigma se calculará a partir del valor pasado en size, según la documentación de la función GaussianBlur, mediante la fórmula $\sigma = 0,3 * ((ksize - 1) * 0,5 - 1) + 0,8$.

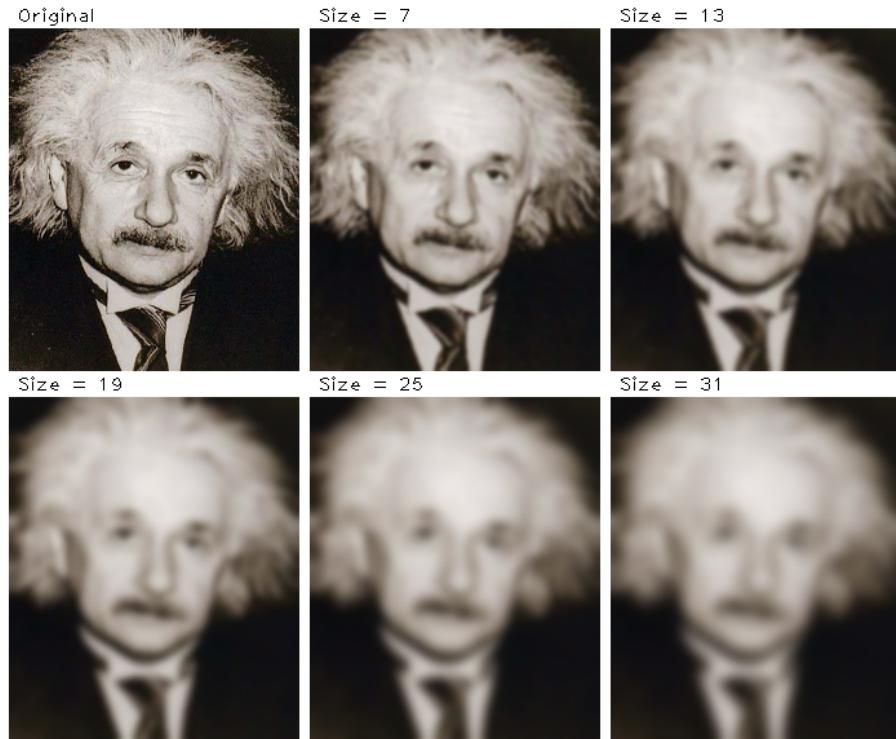


Figura 2: Filtro gaussiano con diferentes valores de size

A la vista de los resultados en las dos imágenes anteriores, vemos que a mayor valor del parámetro sigma, mayor emborronamiento de la imagen y mayor eliminación de altas frecuencias.

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.

```
def get_mask(dx,dy, size=3):
    sobel3x = cv2.getDerivKernels(dx,dy, size, normalize=True)
    return sobel3x[0], sobel3x[1]

dx = get_mask(1,0)
dy = get_mask(0,1)
dx5 = get_mask(1,0,5)

print('\nDerivada respecto de x (size = 3): \n\nVector 1',dx[0].T, '\nVector 2', dx[1].T, '\nMáscara 2D\n', dx[1].dot(dx[0].T))
print('\nDerivada respecto de y (size = 3): \n\nVector 1',dy[0].T, '\nVector 2', dy[1].T, '\nMáscara 2D\n', dy[1].dot(dy[0].T))
print('\nDerivada respecto de x (size = 5): \n\nVector 1',dx5[0].T, '\nVector 2', dx5[1].T, '\nMáscara 2D\n', dx5[1].dot(dx5[0].T))
```

A continuación se muestran los vectores obtenidos y además la matriz de convolución resultante de multiplicar ambos vectores.

```
Derivada respecto de x (size = 3):

Vector 1 [[-0.5  0.   0.5]]
Vector 2 [[0.25  0.5  0.25]]
```

```

Máscara 2D
[[ -0.125 0.       0.125]
 [ -0.25   0.       0.25 ]
 [ -0.125 0.       0.125]]

```

Como podemos ver el vector 1 que será usado para hacer una convolución por horizontal por filas es $[-0.5 \ 0. \ 0.5]$. Al aplicar este vector sobre lineas horizontales constantes el resultado será cero ya que el negativo se cancelará con el positivo. Por ejemplo si lo aplicamos al vector constante a 6: $[6, 6, 6]$ tenemos $6*(-0.5) + 6*0 + 6*0.5 = 0$. Pero si hay un cambio brusco a izquierda y derecha de un pixel al aplicar la convolución el resultado dará un número mayor, por ejemplo si tenemos $[4, 20, 140]$ el resultado tras aplicar la convolución en el pixel de valor 20 nos da $4*(-0.5) + 20*0 + 140*0.5 = 68$. De esta forma podemos intuir que la derivada con respecto a x nos resaltarán bordes verticales como por ejemplo en la siguiente matriz:

$$\begin{bmatrix} 2 & 2 & 150 & 152 & 159 \\ 2 & 2 & 147 & 155 & 155 \\ 2 & 2 & 140 & 153 & 157 \\ 2 & 2 & 2 & 160 & 156 \\ 2 & 2 & 3 & 160 & 155 \\ 2 & 2 & 3 & 155 & 154 \end{bmatrix}$$

```
Derivada respecto de y (size = 3):
```

```

Vector 1 [[0.25 0.5 0.25]]
Vector 2 [[-0.5 0. 0.5]]
Máscara 2D
[[ -0.125 -0.25 -0.125]
 [ 0.       0.       0.      ]
 [ 0.125   0.25   0.125]]

```

En el caso de la derivada en y podemos ver que el vector 2 es $[-0.5 \ 0. \ 0.5]$. El vector dos se usa como vector columna. De esta forma obtendremos el mismo efecto que con la derivada en x solo que ahora se resaltarán los bordes horizontales.

La idea de que la derivada en x resalta bordes verticales y la derivada en y bordes horizontales podrá ser vista en el ejercicio 2.b.

```
Derivada respecto de x (size = 5):
```

```

Vector 1 [[-0.125 -0.25 0.       0.25   0.125]]
Vector 2 [[0.0625 0.25 0.375  0.25   0.0625]]
Máscara 2D
[[ -0.0078125 -0.015625 0.           0.015625  0.0078125]
 [ -0.03125    -0.0625  0.           0.0625     0.03125  ]
 [ -0.046875   -0.09375 0.           0.09375   0.046875  ]
 [ -0.03125   -0.0625  0.           0.0625     0.03125  ]
 [ -0.0078125 -0.015625 0.           0.015625  0.0078125]]

```

Por ultimo se han mostrado los vectores separables con la derivada respecto a x pero de con size igual a 5. La idea es la misma que la de antes, en las zonas con lineas horizontales constantes los negativos se contrarrestaran con los positivo y el resultado será cero y de la misma forma un cambio brusco en una fila hará que al aplicar la convolución se resalte dicho cambio ya que el resultado será significantemente superiora cero.

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 la función laplacian previamente se ha realizado una convolución gaussiana para suavizar la imagen y con ello eliminar ruido, para posteriormente

```

image = read_image('imagenes/bicycle.bmp', True)
image_gray = read_image('imagenes/bicycle.bmp', False)

```

```

def laplacian(img, sigma, borde):
    # Aplicamos una convolucion gaussiana para eliminar ruido
    img_sigma = gaussian_convolution(img, 0, sigma)
    # Devolvemos el laplaciano de a imagen suavizada
    return cv2.Laplacian(img_sigma, -1, ksize=3)

image_laplace1 = laplacian(image, 1, cv2.BORDER_CONSTANT)
image_laplace3 = laplacian(image, 3, cv2.BORDER_REPLICATE)

images_laplacian = [image_laplace1, image_laplace3]
titles_laplacian = ['Laplacian sigma = 1', 'Laplacian sigma = 3']

display_multiple_images(images_laplacian, titles_laplacian, 2)

```



Figura 3: Laplaciano

Podemos apreciar como con el valor de sigma igual a 1 se ve la imagen mas nítida que con sigma igual a 3.

2. IMPLEMENTAR apoyándose en las funciones getDerivKernels, getGaussianKernel, pyrUp(), pyrDown(), escribir funciones los siguientes

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

Para la convolución separable he hecho uso de la función `sepFilter2D` a la cual se le pasan las máscaras separables para realizar la convolución separada.

```

def separable_convolution(img, kernelx, kernely, border):
    """
    Esta función realiza el cálculo de la convolución con máscaras separables
    """
    img = cv2.sepFilter2D(img, -1, kernelx, kernely, border)
    return img

```

Para probar la convolución separada he implementado una convolución gaussiana separable, la cual obtiene el kernel mediante la función `getGaussianKernel`. La función `getGaussianKernel` solo devuelve un vector, ya que al ser la matriz de convolución gaussiana una matriz simétrica las máscaras separables son iguales.

```

def separable_gaussian_convolution(img, size, sigma, border = cv2.BORDER_DEFAULT):
    """
        Filtro gaussiano utilizando convolución separable
    """
    kernel = cv2.getGaussianKernel(size, sigma)
    return separable_convolution(img, kernel, kernel, border)

```

Por último he comparado la convolución gaussiana mediante `GaussianBlur` (haciendo uso de la función `gaussian_convolution` del ejercicio 1.a), con la convolución gaussiana separable.

```

image_gc = gaussian_convolution(image_gray, 7, 2)
image_sgc = separable_gaussian_convolution(image_gray, 7, 2, cv2.BORDER_REFLECT)
titles = ['Convolucion mascara 2D', 'Convolucion separable']

```

```
display_multiple_images([image_gc, image_sgc], titles)
```

Convolución mascara 2D



Convolución separable



Figura 4: Convolución separable

Los resultados son idénticos usando ambas funciones.

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

Para el cálculo de la convolución de la primera derivada he utilizado la función `getDerivKernel`, la cual devuelve los vectores separados de la convolución derivada correspondiente. Después he hecho uso de la función implementada en el apartado anterior para calcular la convolución separable con las máscaras devueltas por `getDerivKernel`.

```
def separable_first_derivate(img, dx, dy, size, border):
    """
    Cálculo de la convolución separable con máscaras de primera derivada.

    Los argumentos dx y dy son booleanos true indica que se realiza
    la derivada correspondiente.

    Nota: dx y dy no pueden ser falsos a la vez.
    """

    if dx or dy:
        kernel = cv2.getDerivKernels(dx*1, dy*1, size)
        return separable_convolution(img, kernel[0], kernel[1], border)
```

He calculado la derivada primera respecto a x, la derivada primera con respecto a x y la derivada con respecto a x e y.

Para los bordes he usado a opción `cv2.BORDER_CONSTANT`, que pone los bordes constantes a cero.

```
image_gray = read_image('imagenes/motorcycle.bmp', False)

# Derivada con respecto a x
image_dx = separable_first_derivate(image_gray, True, False, 3, cv2.BORDER_CONSTANT)

# Derivada con respecto a y
image_dy = separable_first_derivate(image_gray, False, True, 3, cv2.BORDER_CONSTANT)

# Derivada con respecto a x e y
image_dxdy = separable_first_derivate(image_gray, True, True, 3, cv2.BORDER_CONSTANT)

images = [image_gray, image_dx, image_dy, image_dxdy]
titles = ['Original', 'dx', 'dy', 'dx dy']
display_multiple_images(images, titles, 2)
```

A continuación se muestran los resultados. Podemos apreciar lo que se dijo en el ejercicio 1.b, la derivada en x resalta los contornos verticales, mientras que la derivada en y resalta la horizontales.

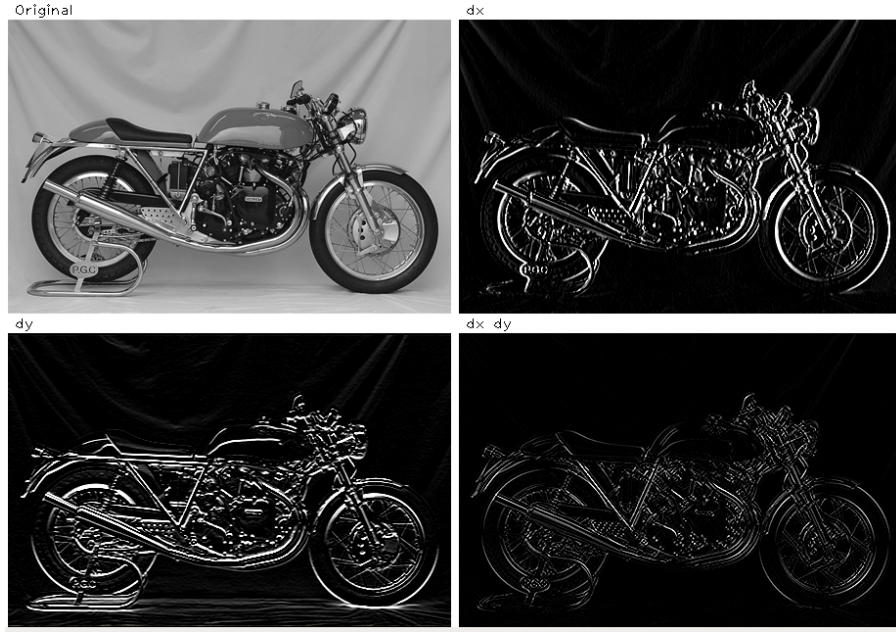


Figura 5: Derivadas primeras

Ahora veamos los efectos de utilizar utilizar tamaños de máscara diferentes para la misma derivada.

```
image3_dx = separable_first_derivate(image_gray, True, False, 3, cv2.BORDER_CONSTANT)
image5_dx = separable_first_derivate(image_gray, True, False, 5, cv2.BORDER_CONSTANT)
image7_dx = separable_first_derivate(image_gray, True, False, 7, cv2.BORDER_CONSTANT)

images = [image_gray, image3_dx, image5_dx, image7_dx]
titles = ['Original', 'dx size mask = 3', 'dx size mask = 5', 'dx size mask = 7']
display_multiple_images(images, titles, 2)
```

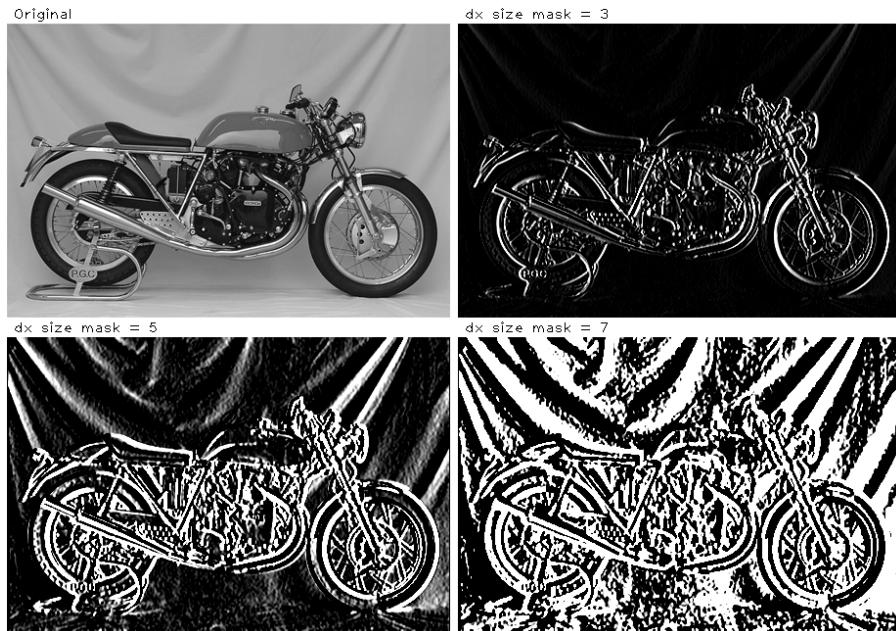


Figura 6: Derivadas primera con diferentes tamaños de máscara

Podemos ver que a mayor tamaño de la máscara se ve mayor grosor en las siluetas

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

La implementación de la segunda derivada es similar a la de la primera como se puede observar.

```
def separable_second_derivate(img, dx, dy, size, border):
    """
    Cálculo de la convolución separable con máscaras de segunda derivada.

    Los argumentos dx y dy son booleanos true indica que se realiza
    la segunda derivada correspondiente.

    Not:a dx y dy no pueden ser falsos a la vez.

    if dx or dy:
        kernel = cv2.getDerivKernels(dx*2, dy*2, size)
        return separable_convolution(img, kernel[0], kernel[1], border)
```

He calculado la convolución de la segunda derivada con respecto a x, con respecto a y, y con respecto a x e y.

```
image_dx = separable_second_derivate(image_gray, True, False, 3, cv2.BORDER_CONSTANT)
image_dy = separable_second_derivate(image_gray, False, True, 3, cv2.BORDER_CONSTANT)
image_dxdy = separable_second_derivate(image_gray, True, True, 3, cv2.BORDER_CONSTANT)

images = [image_gray, image_dx, image_dy, image_dxdy]
titles = ['Original', 'dx2', 'dy2', 'dx2 dy2']
display_multiple_images(images, titles, 2)
```

A continuación podemos ver los resultados.

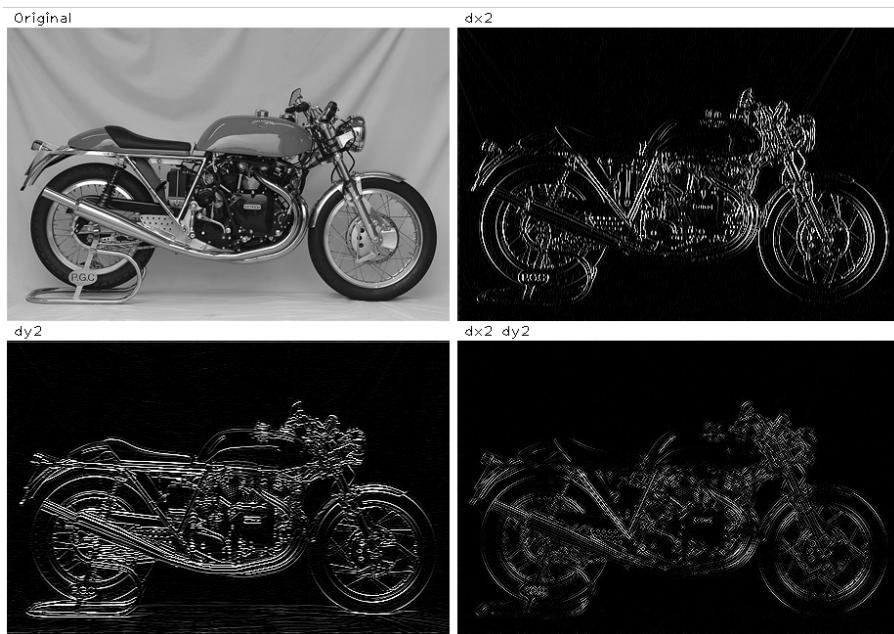


Figura 7: Derivadas segundas

Veamos al igual que hicimos en el apartado anterior el efecto de utilizar tamaños de máscara diferentes.

```
image3_dx = separable_second_derivate(image_gray, True, False, 3, cv2.BORDER_CONSTANT)
image5_dx = separable_second_derivate(image_gray, True, False, 5, cv2.BORDER_CONSTANT)
image7_dx = separable_second_derivate(image_gray, True, False, 7, cv2.BORDER_CONSTANT)

images = [image_gray, image3_dx, image5_dx, image7_dx]
titles = ['Original', 'dx size mask = 3', 'dx size mask = 5', 'dx size mask = 7']
```

```
display_multiple_images(images, titles, 2)
```

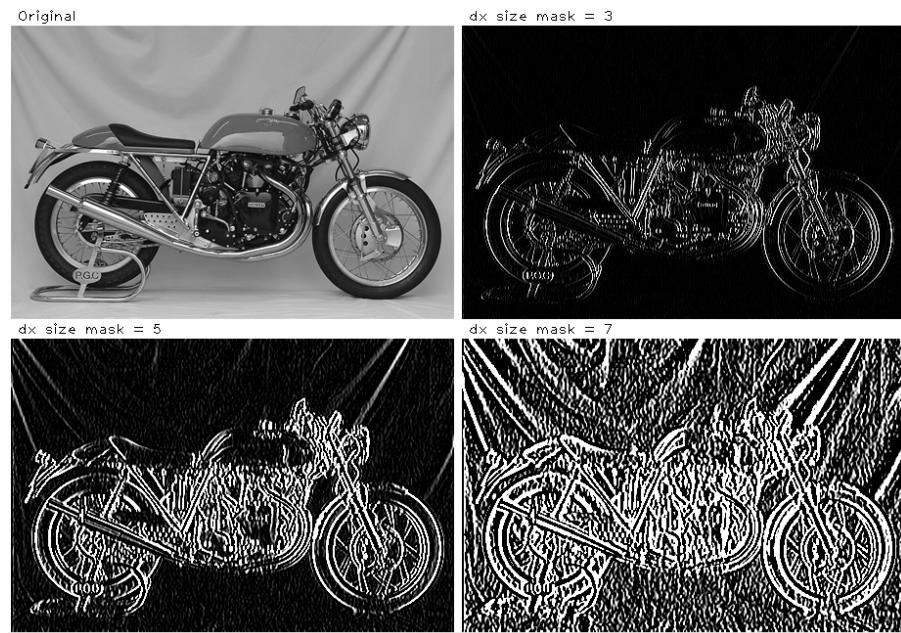


Figura 8: Derivadas segundas con distintos tamaños

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

```
image_gray = read_image('imagenes/cat.bmp', False)

def gaussian_pyramid(img, level=4, border = cv2.BORDER_DEFAULT):
    ''' Esta función genera la representación de una piramide gaussiana del
    nivel indicado (por defecto nivel 4)
    '''
    images = [img]

    # Almacenamos todas las imagenes reducidas
    for i in range(level):
        images.append(cv2.pyrDown(images[i], borderType = border))

    display_multiple_images(images, None, level+1, black)

gaussian_pyramid(image_gray, 4, cv2.BORDER_REFLECT)
```



Figura 9: Pirámide gaussiana

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

Para esta función he usado pyrDown y pyrUp. Para la generación de la pirámide he reducido la imagen, después la he aumentado al tamaño original y he restado a la imagen original la otra imagen obtenida mediante reducción y aumento. El proceso se ha reiterado los las imágenes reducidas.

Al usar pyrUp he utilizado el parámetro dstsize para asegurarme de obtener el tamaño de la imagen antes de ejecutar pyrDown, ya que podría haber problemas si la imagen tuviera tamaño impar en la altura o la anchura.

```
image_gray = read_image('imagenes/motorcycle.bmp', False)

def laplacian_pyramid(img, level=4, border = cv2.BORDER_DEFAULT):
    '''
    Esta función genera la representación de una piramide laplaciana del
    nivel indicado (por defecto nivel 4)
    '''

    images=[]
    image_actual = img
    image_down = None

    # Cálculamos las imagenes y las guardamos en un vector
    for i in range(level):
        w = image_actual.shape[1]
        h = image_actual.shape[0]
        image_down = cv2.pyrDown(image_actual, borderType = border)
        im = cv2.pyrUp(image_down, dstsize=(w, h), borderType = border)
        images.append(cv2.subtract(image_actual, im))
        image_actual = image_down

    images.append(image_down)

    display_multiple_images(images, None, level+1, black)

laplacian_pyramid(image_gray, 4)
```

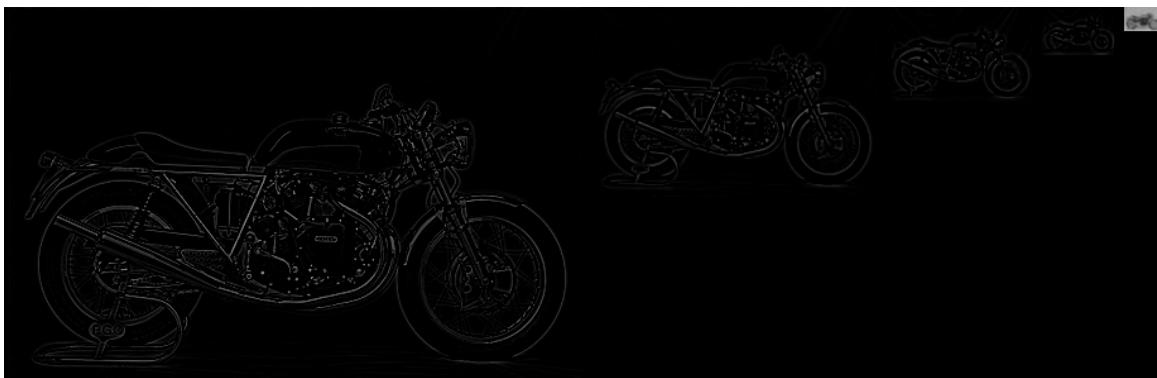


Figura 10: Pirámide laplaciana

3. Imágenes Híbridas:

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).

He obtenido las altas frecuencias restando a la imagen original la imagen suavizada con una convolución gaussiana. He utilizado para la resta la función `subtract`, la cual trunca tras la resta los valores resultantes, así no tendremos problema si el resultado de una resta es negativo ya que lo truncará a cero.

Las bajas frecuencias las he obtenido simplemente aplicando un filtro gaussiano.

```

def hybrid_images(img1, img2, sigma1 = 1.5, sigma2 = 1.5):
    # Obtenemos las imágenes de alta frecuencia y baja frecuencia
    high_frec = cv2.subtract(img1, cv2.GaussianBlur(img1,(0,0),sigma1))
    low_frec = cv2.GaussianBlur(img2,(0,0),sigma2)

    # Hacemos una suma ponderada de las imágenes
    hybrid = cv2.addWeighted(high_frec, 0.6, low_frec, 0.4, 50)

    titles = ['Alta frecuencia', 'Baja frecuencia', 'Hibrida']
    display_multiple_images([high_frec, low_frec, hybrid], titles, 3)

```

2. Realizar la composición con al menos 3 de las parejas de imágenes.

```

img1 = read_image('imagenes/einstein.bmp', False)
img2 = read_image('imagenes/marilyn.bmp', False)
hybrid_images(img1, img2, 5,3)

```



Figura 11: Imagen híbrida

```

img1 = read_image('imagenes/cat.bmp', False)
img2 = read_image('imagenes/dog.bmp', False)
hybrid_images(img1, img2, 4,4)

```



Figura 12: Imagen híbrida

```

img1 = read_image('imagenes/plane.bmp', True)
img2 = read_image('imagenes/bird.bmp', True)
hybrid_images(img1, img2, 4,2)

```



Figura 13: Imagen híbrida

```
img1 = read_image('imagenes/motorcycle.bmp', True)
img2 = read_image('imagenes/bicycle.bmp', True)
hybrid_images(img1, img2, 4,3)
```



Figura 14: Imagen híbrida

```
img1 = read_image('imagenes/fish.bmp', True)
img2 = read_image('imagenes/submarine.bmp', True)
hybrid_images(img1, img2, 4,2)
```



Figura 15: Imagen híbrida

BONUS

1. Cálculo del vector máscara Gaussiano. 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)

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

El tamaño de la máscara resultante será de $2*\text{int}(3*\sigma)+1$, de esta forma nos aseguramos que el tamaño de la máscara es impar. Además elegir este tamaño hace que cojamos el 99 por ciento de la masa de la función de densidad de la distribución normal.

```
def gaussian_kernel(sigma):
    # kernel de tamaño 6*sigma+1
    kernel = [[f(x, sigma)] for x in range(int(-3*sigma), int(3*sigma)+1)]

    kernel = np.asarray(kernel)
    kernel = kernel/sum(kernel)

    return kernel
```

Se ha comparado los resultados de hacer una convolución con el kernel obtenido con la función `getGaussianKernel` y mi propia función `gaussian_kernel`.

```
sigma = 2
# Kernel 1 con función de cv2 y kernel2 con mi función
kernel1 = cv2.getGaussianKernel(sigma*6+1, sigma)
kernel2 = gaussian_kernel(sigma)

print('Kernel1: ', kernel1.T)
print('Kernel2: ', kernel2.T)
```

```
Kernel1:  [[0.0022182  0.00877313  0.02702316  0.06482519  0.12110939  0.17621312
 0.19967563  0.17621312  0.12110939  0.06482519  0.02702316  0.00877313
 0.0022182 ]]
Kernel2:  [[0.0022182  0.00877313  0.02702316  0.06482519  0.12110939  0.17621312
 0.19967563  0.17621312  0.12110939  0.06482519  0.02702316  0.00877313
 0.0022182 ]]
```

Podemos ver que ambos kernel son iguales.

2. Implementar una función que calcule la convolución 1D de un vectorseñal 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)

Los pasos de la implementación de la convolución 1D son los siguientes:

1. Primero calculo los bordes reflejados de izquierda y derecha del vector.
2. Creo un vector extendido con dichos bordes para poder pasar la máscara por todas las posiciones del vector.
3. Aplico la convolución haciendo la sumatoria del producto la máscara con la porción del vector señal que corresponda. Para hacerlo más rápido he utilizado el operador `.dot` de numpy, el cual realiza un producto escalar.

```
def convolution_1D(signal, mask):
    signal_size = len(signal)
    mask_size = len(mask)
    border_size = (mask_size-1)//2

    # Creamos los bordes izquierdo y derecho
    left_border = signal[border_size-1:-1]
    right_border = signal[signal_size:signal_size-border_size-1:-1]
    # Creamos una extensión de signal añadiendo los bordes
    extension = np.concatenate((left_border, signal, right_border))
```

```

convolution = signal.copy()

for i in range(border_size, border_size+signal_size):
    convolution[i-border_size] = np.dot(extension[i-border_size:i-border_size+mask_size], mask)

return convolution

def convolution_1D_color(signal, mask):
    b, g, r = cv2.split(signal)

    b = convolution_1D(b, mask)
    g = convolution_1D(g, mask)
    r = convolution_1D(r, mask)

    return cv2.merge([b,g,r])

```

He realizado una prueba sencilla sobre un vector, para comprobar que funciona correctamente.

```

v = [1,2,3,4,5,6,7,8,9]
m = [-1,0,1]

k = convolution_1D(v,m)

print('Vector original: ', v)
print('Tras aplicar la convolución: ', k)

```

Resultado obtenido.

```

Vector original: [1, 2, 3, 4, 5, 6, 7, 8, 9]
Tras aplicar la convolución: [1, 2, 2, 2, 2, 2, 2, 2, 1]

```

Si nos fijamos en la salida podemos ver que el resultado es correcto.

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

Para implementar esta función me he ayudado de la función `convolucion_1D` del apartado anterior. La convolución se ha hecho iterando la imagen por filas y realizando la convolución 1D con cada fila y el kernelx, después se ha iterado por columnas para realizar la convolución 1D con las columnas de la imagen y el kernely.

Tras realizar la convolución he llamado a la función `truncate_matrix`, para truncar a cero los valores negativos y truncar a 255 aquellos pixeles que sobrepasen dicho valor.

```

def truncate_matrix(matrix, down=0, up=255):
    for i in range(matrix.shape[0]):
        for j in range(matrix.shape[1]):
            if matrix[i,j] > up:
                matrix[i,j] = up
            elif matrix[i,j] < down:
                matrix[i,j] = down
    return matrix

def separable_convolution2(img, kernelx, kernely):
    img = img.astype(np.float32)

    if len(img.shape) != 3:
        # Iteramos la imagen por filas y aplicamos la convolución con kernelx
        for i in range(img.shape[0]):
            img[i,:] = convolution_1D(img[i,:], kernelx)
        # Iteramos la imagen por columnas y aplicamos la convolución con kernely
        for j in range(img.shape[1]):

```

```

    img[:,j] = convolution_1D(img[:,j], kernely)

    img = truncate_matrix(img)
else:
    b, g, r = cv2.split(img)

    for i in range(img.shape[0]):
        b[i,:] = convolution_1D(b[i,:], kernelx)
        g[i,:] = convolution_1D(g[i,:], kernelx)
        r[i,:] = convolution_1D(r[i,:], kernelx)

    for j in range(img.shape[1]):
        b[:,j] = convolution_1D(b[:,j], kernely)
        g[:,j] = convolution_1D(g[:,j], kernely)
        r[:,j] = convolution_1D(r[:,j], kernely)

    b = truncate_matrix(b)
    g = truncate_matrix(g)
    r = truncate_matrix(r)

    img = cv2.merge([b,g,r])

return img

```

Hagamos una prueba para comparar mi función de convolución separada con la de opencv. Para ello voy a hacerlo con máscaras de derivada con respecto a x, comparando con la función del ejercicio 2.b) `separable_first_derivate`.

```

dx = cv2.getDerivKernels(1, 0, 3)

im1 = separable_convolution2(image, dx[0], dx[1])
im2 = separable_first_derivate(image, True, False, 3, cv2.BORDER_CONSTANT)

titles = ['Convolucion propia', 'Convolucion opencv']

display_multiple_images([im1, im2], titles, 2)

```

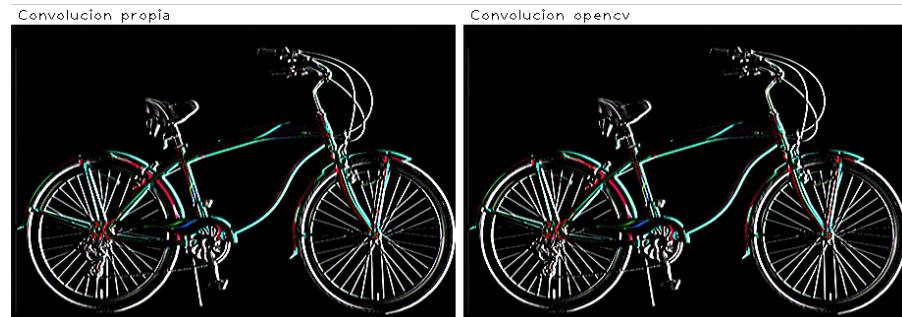


Figura 16: Convolución separable propia y de opencv

Podemos ver que son idénticas lo cual nos da bastante confianza sobre la función implementada.