Visión por Computador

Grado en Informática Industrial y Robótica (GIIROB)

Práctica 3:

Procesamiento de imagen

Contenido:

Autor: José Miguel Valiente González

Bibliografía:

- J. Howse & J. Minichino. “Learning OpenCV 4 Computer Vision with Python 3” (third Ed.) Packt Publishing Ltd. 2020.
- Sandigan Dei, “Python Image Processing. Cookbook”. Packt Publñishing Ltd. 2020.
- B.R. Hunt; R.L. Lipsman, J.M. Rosenberg. “A guide to MATLAB”. Cambridge University Press. 2001.
- Rafael Gonzalez, Richard Woods, Steven Eddins. ”Digital Image Processing Using Matlab”. Second Edition, McGraw Hill, 2010
- Official documentation for Python 3.10 https://docs.python.org/es/3/tutorial/index.html

Como ya se indicó en una práctica anterior, una imagen es la proyección de un mundo tridimensional complejo sobre el plano de un sensor. La imagen en sí está formada por puntos en dicho plano con unas coordenadas y un valor asociado (nivel de gris o color). Hay tres tipos de operaciones que podemos realizar sobre la imagen: las que transforman las coordenadas de los puntos, como rotaciones, traslaciones, escalados; que son transformaciones puramente geométricas, las que transforman el valor de color o gris del punto, que llamamos operaciones de espacios de color, y las operaciones que transforman el valor de cada punto mediante algún algoritmo complejo, que llamamos de procesamiento de imagen.

En esta práctica nos vamos a centrar en las operaciones de **procesamiento de imagen**, para lo cual vamos a revisar el amplio conjunto de funciones que dispone el **PYTHON** para este propósito.

Objetivos

- Describir algunas de las funciones de procesamiento de imagen disponibles en las bibliotecas: **OpenCV**, **Matlpotlib** o **Pillow**.

Material

- Programas: IDE Thonny, Python 3.10 o similar, Matplotlib, OpenCV y Pillow.
- Archivos de imagen y *.py disponibles en PoliformaT.

1. Introducción

En la práctica anterior hemos visto que las bibliotecas **Pillow**, **Matplotlib** y **OpenCV** tienen múltiples funciones para trabajar con imágenes. La biblioteca **scikit-Image** también dispone de múltiples herramientas, pero de momento nos quedaremos con las primeras.

También recordar lo importante que es el uso de las ayudas**.** Dada la gran cantidad de funciones que incorporan todas estas herramientas, así como la diversidad de parámetros, es conveniente tener siempre abierta una ventana de ayuda para consultar durante el trabajo con el entorno.

A continuación, vamos a repasar algunas de las funciones de procesamiento de imágenes (*Image Processing)* dentro de cada grupo de técnicas de procesamiento que se indicó en la teoría, que eran:

- Operadores puntuales
- Técnicas basadas en el histograma
- Operadores espaciales
- Pseudo-color
- Detección de bordes

Pero primero vamos a describir las funciones para obtener el histograma de una imagen.

2. El histograma de una imagen

Como ya sabemos, el histograma de una imagen de niveles de gris es un vector de valores que representan la frecuencia de aparición de los distintos niveles de gris en la imagen. Para el caso de imágenes de 256 niveles de gris – 8 bits por pixel – el histograma contendrá 256 números enteros, que se pueden representar gráficamente, como se aprecia en la figura 1.

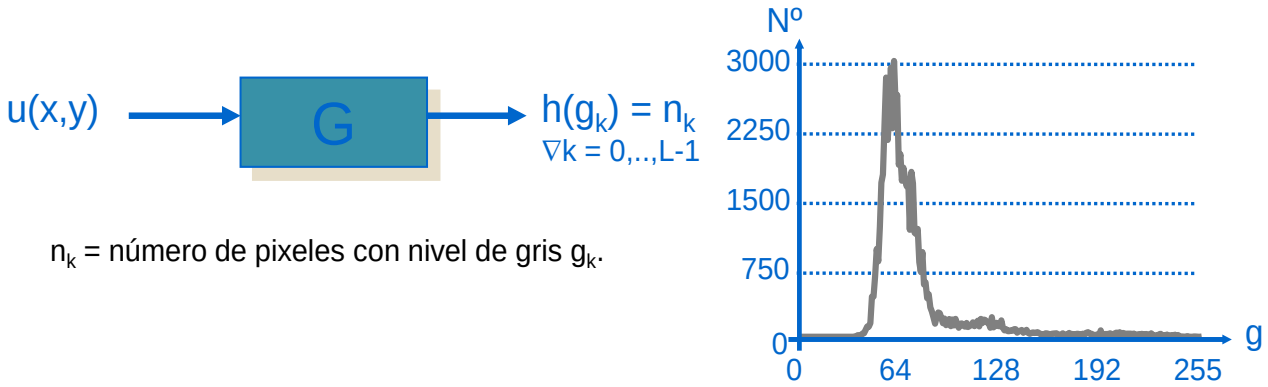

Figura 1.- Histograma de una imagen de niveles de gris.

En realidad, hacer un histograma de un conjunto de datos no es más que agruparlos en contenedores (bins) de acuerdo con su valor. Para este propósito mtplotlib.pyplot (como plt) cuenta con diversas funciones que pueden trabajar con vectores o matrices, como las funciones hist(X), hist2D(X,Y), hexbin(), etc. Todas estas funciones son similares a las encontradas en MATLAB, pero los parámetros en Python pueden cambiar. La función principal que calcula y dibuja un histograma es la siguiente:

- N,bins,patches = plt.hist(X, bins, range=None, density=False, weights=None, cumulative=False, histtype='bar' ,..)

El problema que tiene esta función cuando trata con imágenes (X es un array 2D) es que considera cada columna de la matriz 2D como un *dataset* y calcula el histograma y el gráfico para cada columna, lo cual es extremadamente lento. Para evita esto hay que 'aplanar' la imagen 2D empleando la función numpy.ravel(). El siguiente código emplea esta técnica.

```
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image

img = np.asarray(Image.open('building4.jpg'))
plt.hist(img.ravel(),256,[0,256],ec='b')
plt.show()
```

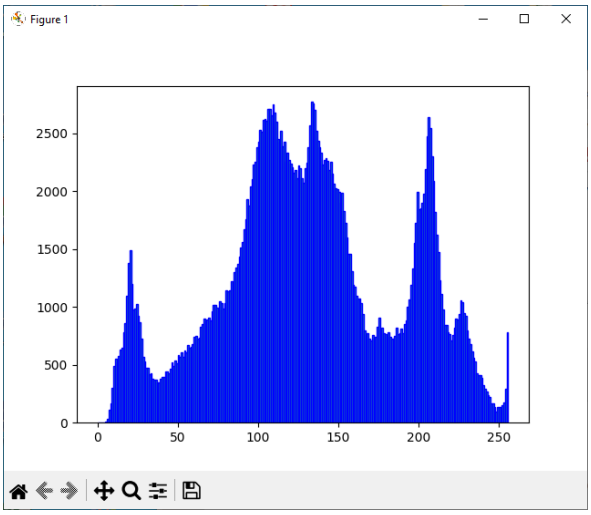

Figura 2 – Histograma resultante del código anterior.

Como alternativa al plt.hist se pueden emplear las funciones de cálculo de histogramas que ofrece **NumPy**, que son np.histogram(), np.histogram2d(), np.histogramdd(), etc.. La función principal sería:

- hist, bins = np.histogram(X, bins, range=None, normed=None, weights=None, density=None)

- Parámetros:
  - X: array. El histograma se calcula con la matriz aplanada a una dimensión.
  - bins:
    - Número de bins (defecto=10) de igual tamaño en el rango de valores.
    - Secuencia de intervalos de bins [10, 20, 34, 45].
    - String: 'auto', 'fd', 'doane','scott', 'stone','sqrt', …
  - range: tupla(x_min,x_max) o None.
  - density: True si queremos la función de densidad, de forma que la suma de todos los valores de densidad es uno.

$$density = \frac{counts}{sum(counts) \cdot diff(bins)}$$

  - weights: Pesos de cada valor de X.
  - normed: No usado en la última versión.
- Devuelve:
  - hist: Array con el histograma (cuentas en cada bin).
  - bins: Array con los bordes de cada bin.(length(hist)-1)

Puesto que la función no visualiza nada, podemos emplear las funciones de *matplotlib.pyplot* para este propósito. La figura 2 muestra el resultado del código siguiente.

```
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image

img = np.asarray(Image.open('building4.jpg'))
hr, edges_r = np.histogram(img[:,:,0],256)

fig, axs = plt.subplots(1, 2, figsize=(12,4)) # Dos subplots en horizontal
axs[1].stairs(hr, edges_r, label='Red histogram', ec='r')
axs[1].set_title("Step Histograms")
axs[1].legend()

plt.sca(axs[0])   # Para poner en el subplot izquierdo la imagen
plt.imshow(img)

plt.show()
```

El uso de las funciones anteriores, de **NumPy** o **matplotlib.pyplot**, es porque son muy ágiles y permiten una gran flexibilidad en las visualizaciones.

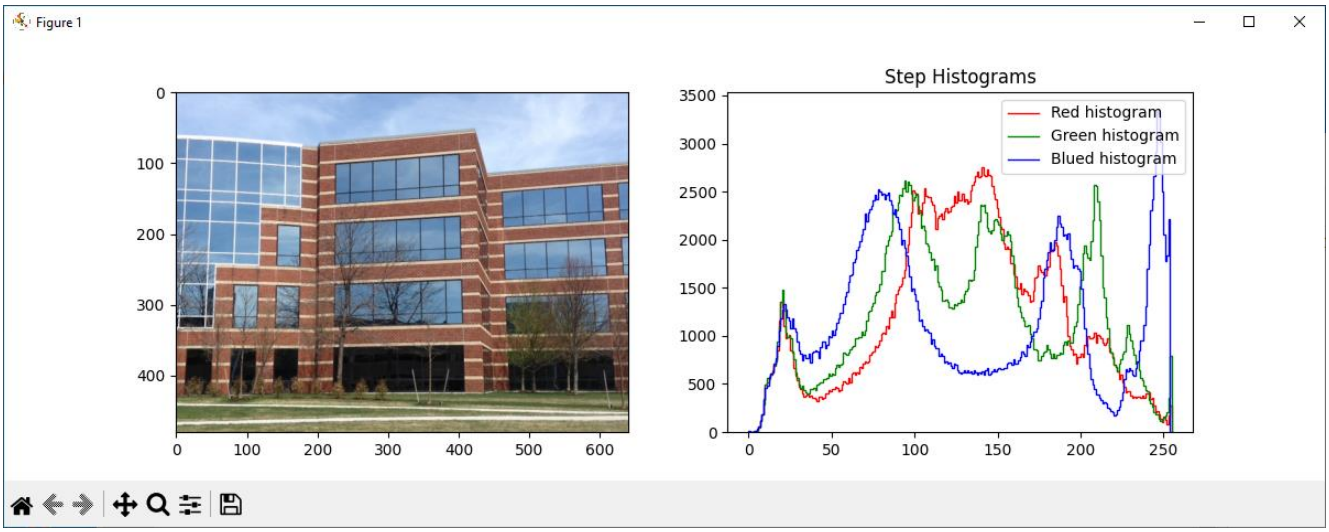

Figura 3 – Histograma RGB resultante del código anterior.

Observe que podemos cambiar las propiedades de los objetos gráficos empleando las funciones de *matlpotlib.pyplot*. Para ello consulte las ayudas en:
https://matplotlib.org/stable/api/pyplot_summary.html

Los histogramas se pueden normalizar como indica la Tabla 1, donde v_i: es el valor del contenedor (bin) - c_i: es el número de elementos del bin - w_i: es la anchura del bin - N es el número total de datos de entrada.

Type	Bin values	Descripción
'count' (default)	$v_i = c_i$	Cuenta o frecuencia de observaciones
'countdensity'	$v_i = {}^{c_i}/_{w_i}$	Frecuencia escalada con la anchura del bin
'cumcount'	$v_i = \sum_{1}^{i} c_i$	Frecuencia o cuenta acumulada
'probability'	$v_i = {}^{c_i}/_{N}$	Probabilidad relativa
'pdf'	$v_i = {}^{c_i}/_{N \cdot w_i}$	Función de densidad de probabilidad
'cdf'	$v_i = \sum_{1}^{i} \frac{c_i}{N}$	Función de densidad de probabilidad acumulada

Tabla 1 – Tipos de normalización del histograma

Ejercicio 1 – Histogramas

1. Tome unas imágenes en color cualesquiera de las utilizadas en las prácticas anteriores, o de las disponibles en PoliformaT .

2. Escriba el código anterior y pruebe el resultado.

3. Añada las instrucciones necesarias para obtener el histograma de cada uno de los tres canales de color de la imagen RGB – hr, hg, hb - y mostrarlos sobre la misma gráfica, cada uno en su color, añadiendo:

```
axs[1].stairs(hg, edges_g, label='Green histogram', ec='g', hatch='||') …….
```

4. Modifique alguna de las propiedades de los histogramas, como por ejemplo el número de bins a valor 16 o la 'density = True, y observe el resultado.

Por otra parte, se pueden usar las funciones similares de **OpenCV**. La función principal para calcular el histograma de una imagen es:

- hist = cv2.calcHist(image, channels, mask, histsize, ranges)

- Parámetros:
  - image: Imagen fuente del tipo uint8 o float32, representada como una lista[img].
  - channels: índice de los cañales para los que se calcula el histograma [0,1,2] .
  - mask: Imagen de máscaras (0 o 255) 0 'None' .
  - hitsize: Número de bins [256].
  - ranges: Intervalos de valores de salida [0,256].
- Devuelve:
  - hist: Array con el histograma (cuentas en cada bin).

```
import numpy as np
import cv2
from matplotlib import pyplot as plt
img = cv2.imread('home.jpg')
color = ('b','g','r')
for i,col in enumerate(color):
histr = cv.calcHist([img],[i],None,[256],[0,256])
plt.plot(histr,color = col)
plt.xlim([0,256])
plt.show()
```

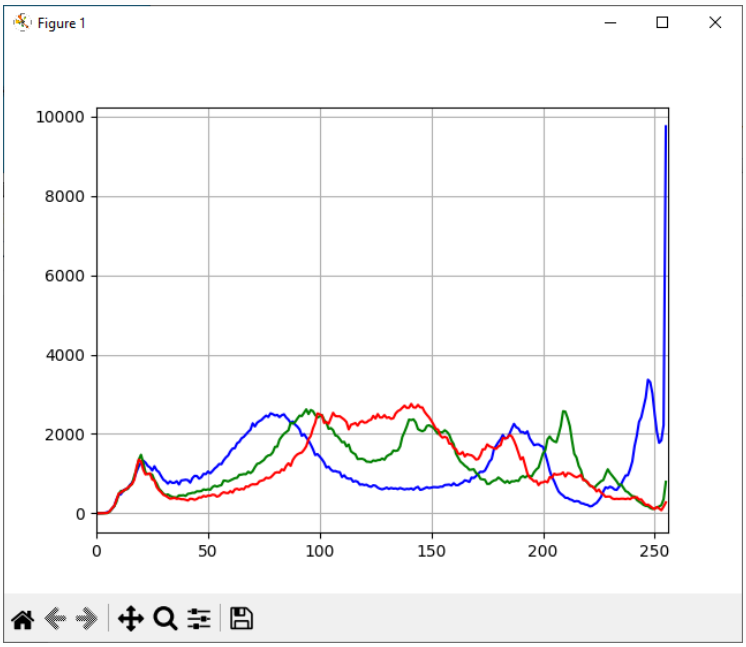

Figura 4 – Histogramas para R, G y B.

3. Operaciones basadas en el histograma

Otras funciones relacionadas con el histograma son cv2.equalizeHist(), cv2.compareHist(), cv2.calcBackProject() y cv2.createCLAHE().

La primera función se utiliza para ecualizar el histograma de una imagen, que ajusta los niveles de gris (o bandas de cada color) de forma que el resultado es una nueva imagen donde la distribución de probabilidad de los píxeles – o sea el histograma - es cuasi-constante. Esto permite ajustar el contraste de la imagen, pero sólo tiene buenos resultados con imágenes que tienen todo muy oscuro o muy claro. La función para esto es:

- img_dst = cv2.equalizeHist(img_src)

Esta función ecualiza la imagen completa, teniendo en cuenta el contrate global de todos los píxeles. Esto provoca que aquellas zonas que son muy brillantes, o muy oscuras, van a predominar sobre el resto. Por ello, en ocasiones es mejor ecualizar en pequeñas porciones de la imagen de manera que cada equalización se ajuste a esa porción. Para ello se utiliza la clase CLAHE (Contrat limited Adaptive Histogram Equalization).

- CLAHE clahe = cv2.createCLAHE(clipLimit, tileGridSize)
- img_dst = clahe.apply(img_src)

Esta clase se utiliza para dividir la imagen en pequeños cuadrados (tiles) del tamaño indicado (default=8) y en cada cuadrado se hace la ecualización. Para evitar que ese cuadrado tenga solo ruido se emplea una limitación del contraste, de forma que si un bin supera ese límite (defautl=40) los píxeles correspondientes son distribuidos uniformemente sobre los otros bins. Finalmente se eliminan los artefactos en los bordes de los cuadrados mediante interpolación bilineal.

► *Ejercicio 2 – Ecualización del histograma*

1. Tome una imagen de niveles de gris, o convierta una de color a gris, y muestre su histograma. Después realice la ecualización del histograma y muestre la imagen resultante y su histograma. Compruebe si, efectivamente, ese histograma resultante es más uniforme.

2. Pruebe la ecualización con la imagen 'clahe_1.jpg' y compruebe si el resultado es satisfactorio. Luego pruebe con CLAHE y compare el resultado con el anterior.

4. Ajuste de contraste

Las operaciones de ajuste de contraste (dilatación, recortado, etc...) se realizan mediante una función que llamaremos imadjust que ajusta los niveles de gris en el rango [low_in high_in] al rango de salida [low_out high_out]. Por simplicidad los valores de salida serán 0 y 255, con lo que el ajuste es el indicado por la figura 5.

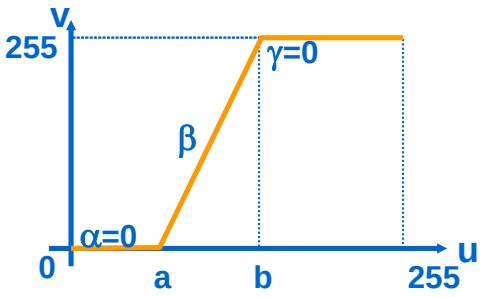

Figura 5 – Ajuste de contraste (u: input; v: output).

con valores entre 0 y 1. Si no se indica nada, la función recorta el 1% por abajo y el 1% por arriba del intervalo de valores de gris de la imagen. La función de salida es básicamente la siguiente:

$$\text{Output} = (\text{Input} - \text{low\_in})/(\text{high\_in} - \text{low\_in}) * 255$$

```
def imadjust (img,  low_in,  high_in):
    min = low_in
    max = high_in
    img_out = np.round( 255.0 * (img - min)/( (max-min+1)  ).astype(np.uint8)
    img_out[img < min] = 0
    img_out[img > max] = 255
    return img_out
```

Otra forma es empleando tablas LUT (LookUp Table) que representan la conversión indicada por la gráfica de la figura 5, como sigue:

```
def imadjustLUT (img,  low_in,  high_in, *args):
    if (type(args is np.ndarray):
         LUT = args
    else:
         min = low_in
         max = high_in
         # Make a LUT (Look-Up Table) to translate image values
         LUT = np.zeros(256,dtype=np.uint8)
         LUT[max+1:] = 255
         LUT[min:max+1] = np.linspace(start=0,stop=255,num=(max-min)+1,endpoint=True, dtype=
np.uint8)
    # Apply LUT and save resulting image
    img_out = LUT[img]
    return img_out
```

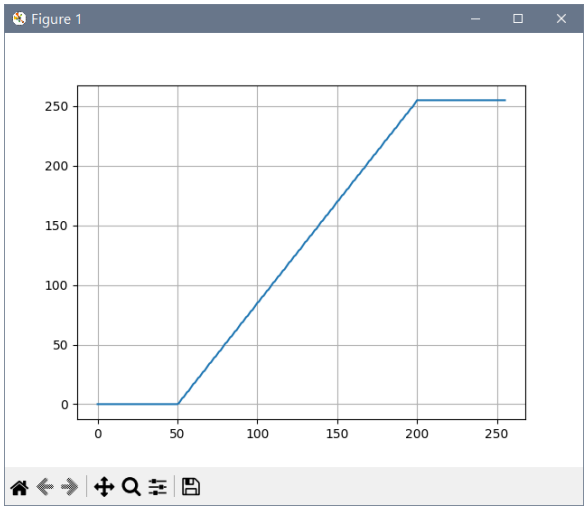

Figura 6 – LUT: Función de conversión de la imagen.

```
out = imadjustLUT(img, 50, 200)
plt.plot(LUT)
plt.grid()
plt.show()
```

Si dibujamos la LUT por defecto obtendremos la figura 6 Podemos crear otras LUTs y pasarlas a la función como cuarto parámetro.

▶ *Ejercicio 3 – Ajuste de contraste de una imagen*

1. Tome una imagen cualquiera e implemente un programa para ajustar el contraste mediante la función que se muestra en la figura 3. Utilice min=100 máx.=200, o cualesquiera de valores que desee. Muestre en una ventana la imagen original y la resultante.

2. Repita lo anterior, pero con las LUT indicadas en la figura 7.

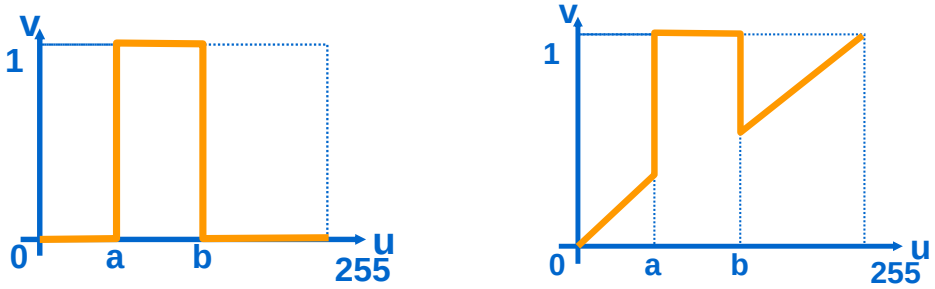

Figura 7 - Tipos de LUTs para ajuste de contraste.

5. Filtrado de imagen

Para las operaciones de filtrado de imagen mediante máscaras de convolución (kernels), **OpenCV** dispone de la función filter2D():

- dst = filter2D(src,ddepth,kernel [, dst[,anchor[, delta[, borderType]]]])
- Parámetros:
  - src: Imagen fuente del tipo uint8 o float32
  - ddepth: Profundidad deseada de la imagen resultado. ddepth=-1 idem que src.
  - kernel: Matriz cuadrada con la máscara de convolución. De tipo float.
  - anchor: Punto de origen de la máscara (-1,-1) es el centro del kernel.
  - delta: Valor opcionala sumar a la imagen de destino.
  - borderType: Tipo de borde cv2.BORDER_CONTANT, cv2.BORDER_REPLICATE, cv2.BORDER_REFELECT, cv2.BODER_DEFAULT,…
- Devuelve:
  - dst: Array con la imagen resultante.

Básicamente, esta función hace la convolución de la imagen '*src*' con el '*kernel*' y el resultado lo deja en la imagen destino '*dst*'. El *kernel* es una matriz cuadrada, de dimensión impar (3x3, 5x5, 7x7, ..), con los pesos a aplicar a cada pixel. Dependiendo de los valores de este *kernel* se tienen distintos tipos de filtros, como se muestra a continuación.

```
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image
import cv2

plt.figure(1)
src = np.array (Image.open('building4.jpg'))
plt.imshow(src)
H = np.array([
    [0,-1,-1],
    [1, 0, -1],
    [1, 1, 0] ])
plt.figure(2)
dst = cv2.filter2D(src, -1, H)
plt.imshow(dst)
plt.gca().set_title("Emboss Filter")
plt.show()
```

Tipo	kernel		Tipo	kernel
Realce (Emboss)	$\begin{bmatrix}0 & -1 & -1\\1 & 0 & -1\\1 & 1 & 0\end{bmatrix}$		Suavizado o promedio (Blurr Filter)	$\begin{bmatrix}1 & 1 & 1\\1 & 1 & 1\\1 & 1 & 1\end{bmatrix}/9$
Realce 2 (Emboss)	$\begin{bmatrix}-2 & -1 & 0\\-1 & 1 & 1\\0 & 1 & 2\end{bmatrix}$		Paso alto (Edge detection)	$\begin{bmatrix}-1 & -1 & -1\\-1 & 8 & -1\\-1 & -1 & -1\end{bmatrix}$
Agudizado (Sharpennig)	$\begin{bmatrix}0 & -1 & 0\\-1 & 5 & -1\\0 & -1 & 0\end{bmatrix}$		Sepia	$\begin{bmatrix}0.272 & 0.534 & 0.1311\\0.349 & 0.686 & 0.168\\0.393 & 0.769 & 0.189\end{bmatrix}$

Tabla 2 - Tipos de máscaras de convolución o *kernels* para distintos tipos de operaciones.

► *Ejercicio 4 – Filtrado de imágenes*

3. Tome una imagen cualquiera e implemente el código anterior para aplicar un filtrado. Muestre en una ventana la imagen original y la resultante.

4. Repita lo anterior para los distintos tipos de filtros indicados en la tabla 2.

Para filtrado de imágenes se pueden emplear también otros tipos de filtros, como el filtro Gausiano, para el que **OpenCV** dispone de la siguiente función:

- img_dst = cv2.GaussianBlurr(src,ksize, sigmaX [, dst[, sigmaY[, borderType]]])
- Parámetros:
  - src: Imagen fuente del tipo uint8 o float32
  - ksize: tamaño del kernal (positivo e impar)
  - sigmaX: Parámetro sigma 'σ_X' del gausiano en la dirección X.
  - sigmaY: Parámetro sigma 'σ_Y' del gausiano en la dirección Y. Si el cero es igual a 'σ_X'.
  - borderType: Tipo de borde cv2.BORDER_CONTANT, cv2.BORDER_REPLICATE, cv2.BORDER_REFELECT, cv2.BODER_DEFAULT,…
- Devuelve:
  - dst: Array con la imagen resultante.

También dispone de funciones específicas para algunos filtros comúnes, incluyendo algunos de los indicados anteriormente, como son:

- cv2.bilateralFilter(src,…) -- Filtro bilateral
- cv2.blur(src,…) -- Filtro paso bajo
- cv2.boxFilter(src,…) -- Filtro de promedio
- cv2.buildPyramid(src,…) -- Pirámide de filtros gausianos
- cv2.medianBlur(src,…) -- Filtro de mediana
- cv2.sqrBoxFilter(src,..) -- Filtro de suma de cuadrados
- cv2.erode(src,…) -- Erosión morfológica
- cv2.dilate(src,..) -- Dilatación morfológica
- cv2.morphologyEx(src,..) -- Operaciones morfológicas en general.

También se pueden obtener los kernels de las distintas operaciones:

- cv2.getGaussiankernel(ksize,…) -- Coeficientes de un filtro Gausiano
- cv2.getGaussiankernel(ksize,…) -- Coeficientes de un filtro de Gabor
- cv2.getStructuringElementl(ksize,…) -- Coeficientes de un elemento estructurante
- etc…

► *Ejercicio 5 – Filtrado Gaussiano y de mediana*

1. Tome una imagen cualquiera e introduzca un ruido *Salt&Papper* o un ruido *Gaussiano*. Para esto abra el archivo 'noise.py' que se encuentra en PoliformaT y estudie las funciones indicadas.

2. Añada un ruido gaussiano y aplique a la imagen con ruido un filtrado Gaussiano con la función anterior. Muestre en una ventana la imagen original y la resultante.

3. Repita lo anterior para el filtro de mediana y compare los resultados.

6. Detección de bordes

Como ya sabemos, la detección de bordes se realiza mediante la convolución de la imagen con operadores espaciales (máscaras o '*kernels*') que representan:

- Modelos sencillos de discontinuidades: como puntos, líneas o esquinas.
- Mediante operadores de derivadas de primer o segundo orden: como los operadores de Sobel, Prewitt o LoG

Algunos de estos operadores de derivada se pueden obtener mediante las siguientes funciones de **OpenCV**:

- `dst = cv2.Sobel(src, ddepth, ksize. …)`
- `dst = cv2.Scharr(src, ddepth, ksize. …)`
- `h = cv2.spatialGradient(src, ddepth, ksize. …)`
- `h = cv2.Laplacian(src, ddepth, ksize. …)`

Y las máscaras que se aplican se muestran en la tabla 3 adjunta.

Tipo	kernel	Tipo	kernel
Sobel_X	$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$	Sobel_Y	$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$
Scharr	$\begin{bmatrix} -3 & 0 & 3 \\ -10 & 0 & 10 \\ -3 & 0 & 3 \end{bmatrix}$	Laplacian	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$

Tabla 3 - Tipos de máscaras de convolución o *kernels* para detección de bordes.

► *Ejercicio 6 – Detección de bordes*

1. Tome una imagen cualquiera y pruebe las funciones de Sobel para detección de bordes horizontales y verticales.
2. Combine el Laplaciano y el Gaussiano para crear un LoG.
3. Muestre los distintos resultados en ventanas diferentes.

7. Entrega de resultados

Para entregar los resultados de la práctica, haga un documento (Word o similar) con los distintos ejercicios incluyendo, en cada uno, lo siguiente:

- Enunciado del ejercicio
- Script de PYTHON (copy – paste)
- Imágenes de los resultados (menú *Edit->Copy figure* en la ventana y paste en el Word)
- Comentarios personales sobre el ejercicio y los resultados (si procede)

Incluya, en la página inicial, el título de la práctica y el nombre del/los alumnos implicados. Convierta el archivo a pdf y súbalo a PoliformaT al espacio compartido de cada alumno implicado.