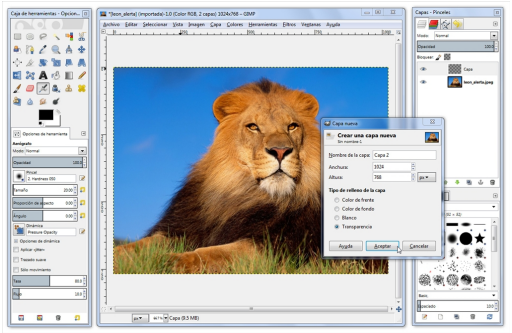
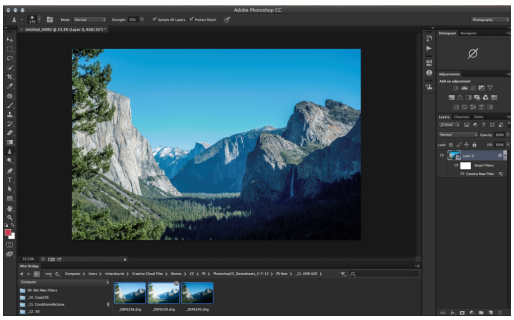
Informática - Curso 2020/2021

Práctica 3: filtros de imágenes.

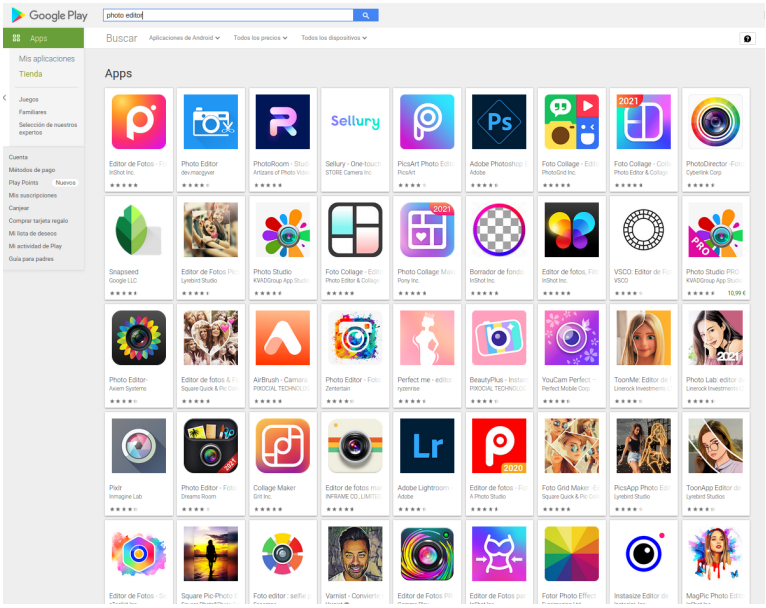
Motivación

Seguro que has utilizado alguna vez en tu ordenador programas para crear, editar, transformar y manipular imágenes. Por ejemplo, GIMP, software libre y el favorito en los sistemas GNU/linux, y Photoshop están entre los más famosos.





También hay decenas de apps para teléfonos móviles con la misma finalidad. ¿Cuál te gusta más?

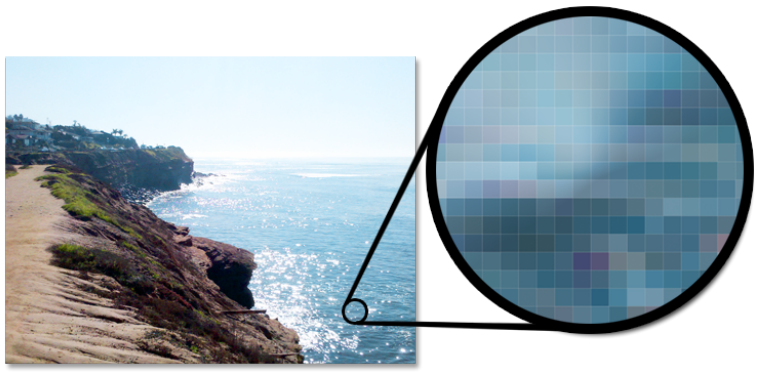


Una funcionalidad básica de la mayoría de estos programas de edición de imágenes es la aplicación de filtros. Hay filtros de todo tipo: para difuminar, reducir ruido, desenfocar, acentuar contrastes, resaltar líneas... puede que te suenen más los nombres en inglés *blur, smooth, enhance, sharpen....* Quizás te sorpenda que algunos de estos filtros sean también conocidos por el nombre propio de un matemático, por ejemplo gaussiano (http://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm) o laplaciano (http://homepages.inf.ed.ac.uk/rbf/HIPR2/log.htm).

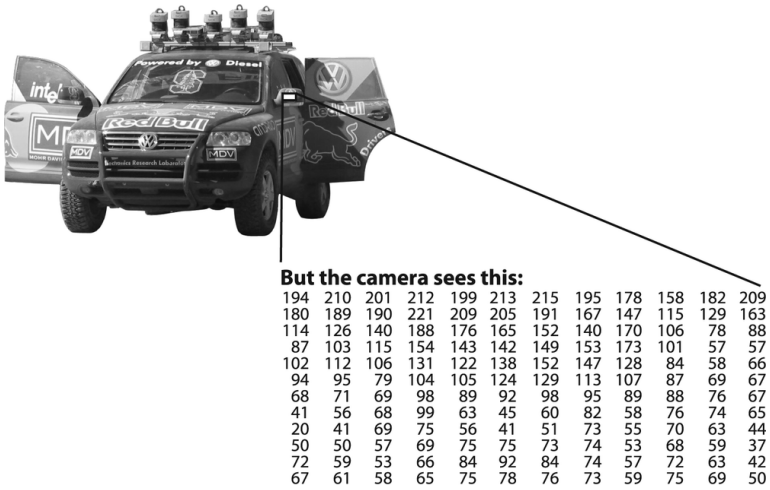
¿Cómo funcionan los filtros?

Pero.. ¿cómo se hacen estas transformaciones en las imágenes?

Las imágenes están compuestas de píxeles y cada píxel almacena un valor numérico que codifica el color. En la siguiente imagen puedes ver una zona aumentada de una fotografía y los píxeles que se encuentran en esa zona.



Para imágenes en blanco y negro este valor es un único número, pero para imágenes en color cada píxel contiene una terna de valores. **Vamos a suponer, de momento, que estamos trabajando con imágenes en blanco y negro y que el rango de valores de cada píxel es de 0 a 255**. La imagen siguiente muestra los valores de la matriz de valores asociada a la zona de la imagen destacada con un pequeño recuadro blanco en el retrovisor.



Entonces, si una imagen es esencialmente una matriz de números, la aplicación de filtros a las imágenes tiene que ser algún tipo de operación sobre matrices.

Los filtros que hemos comentado se basan en un proceso que matemáticamente se conoce como **convolución** (https://en.wikipedia.org/wiki/Convolution). Aunque el concepto matemático genérico de convolución puede ser sofisticado, a nivel práctico, trabajando con imágenes, que son discretas, la **convolución de una imagen** (https://en.wikipedia.org/wiki/Kernel_(image_processing)) es un proceso bastante sencillo. Se parte de una matriz, la **matriz de convolución**, que habitualmente es pequeña. Esta matriz indica cómo tienen que realizarse una serie de sumas y productos que afectan a los píxeles de la imagen. Con un ejemplo se ve muy claro:

Para reducir ruido o suavizar (*smooth*) una imagen se puede aplicar una matriz de convolución muy sencilla, por ejemplo

1	1	1

1	1	1
1	1	1

Supongamos que queremos calcular el valor del pixel (i,j) de la imagen resultante. Lo que tenemos que hacer es superponer el centro del kernel (son matrices, usualmente cuadradas, siempre con dimensión impar) en el pixel (i,j) de la imagen original. Calculamos el valor de multiplicar los valores del kernel por los de la imagen en cada uno de los píxeles afectados. La suma de estos productos parciales es el valor del pixel (i,j) en la imagen resultante.

Supongamos que tenemos una imagen (muy pequeña) y que los valores de su píxeles son los siguientes:

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

El resultado de aplicar la convolución *smoooth* al píxel (2,2), cuyo valor es 15, es
$8 \times 1 + 9 \times 1 + 10 \times 1 + 14 \times 1 + 15 \times 1 + 16 \times 1 + 20 \times 1 + 21 \times 1 + 22 \times 1 = 135$.

Si el kernel tuviera otros valores los multiplicadores variarían. El resultado de aplicar el kernel anterior a toda la matriz es:

1	2	3	4	5	6
7	72	81	90	99	12
13	126	135	144	153	18
19	180	189	198	207	24
25	234	243	252	261	30
31	32	33	34	35	36

Observa que solo aplicamos el kernel a los píxeles que permiten que el kernel entre completamente en la imagen. En el ejemplo anterior, como el kernel tiene dimensión 3, los píxeles del borde no son transformados y simplemente conservan su valor.

En los siguientes enlaces puedes ver el proceso funcionando con varios ejemplos
http://beej.us/blog/data/convolution-image-processing/ (http://beej.us/blog/data/convolution-image-processing/)

http://micro.magnet.fsu.edu/primer/java/digitalimaging/processing/kernelmaskoperation/ (http://micro.magnet.fsu.edu/primer/java/digitalimaging/processing/kernelmaskoperation/)

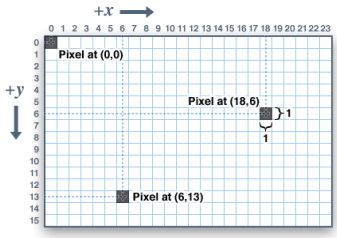
http://songho.ca/dsp/convolution/convolution2d_example.html (http://songho.ca/dsp/convolution/convolution2d_example.html)

En el Gimp también se puede definir manualmente un kernel y ver su resultado: Filtros->Genérico->Matriz de Convolución (Matriz fija de 5x5).

Índices en imágenes y matrices

Un detalle en el que conviene fijarse y entender bien es la diferente forma en la que los índices se utilizan en imágenes y matrices.

Por un lado, en las imágenes, se utiliza un sistema de coordenadas derivado de la geometría en el plano.



Por otro lado, en las matrices, se indexan los elementos por fila y columna, respectivamente:

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

Cuando decimos que, para hacer una convolución multiplicamos elemento a elemento los píxeles de una imagen y una matriz, por ejemplo

la imagen $\begin{matrix} i_{0,0} & i_{1,0} & i_{2,0} \\ i_{0,1} & i_{1,1} & i_{2,1} \\ i_{0,2} & i_{1,2} & i_{2,2} \end{matrix}$ y la matriz $\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$

Estamos diciendo que queremos calcular:

```
i.getpixel((0,0)) * a
i.getpixel((0,1)) * d
i.getpixel((0,2)) * g
i.getpixel((1,0)) * b
i.getpixel((1,1)) * e
i.getpixel((1,2)) * h
i.getpixel((2,0)) * c
i.getpixel((2,1)) * f
i.getpixel((2,2)) * i
```

Pero el índice, en la matriz de `d` es (1,0) y no (0,1) como su píxel correspondiente, el de `g` es (2,0) y no (0,2) como el de su píxel correspondiente...

Pensando ya en programar el algoritmo, para que el manejo de los índices fuera lo más sencillo posible, lo mejor sería que los elementos del kernel tuvieran el mismo índice que los elementos de la imagen, es decir, si los valores de la matriz estuvieran en una lista de listas, `k`, nos gustaría que la forma de acceder a los elementos fuese algo así:

```
k[0][0] = a
k[0][1] = d
k[0][2] = g
k[1][0] = b
k[1][1] = e
k[1][2] = h
```

```
k[2][0] = c
k[2][1] = f
k[2][2] = i
```

Una forma fácil de conseguir esto es que el kernel que nos dan esté traspuesto. Otra forma de decir esto es que la lista de listas que representa al kernel sea una lista de las columnas de la matriz. Por tanto, sin pérdida de generalidad, podemos asumir y así lo haremos para esta práctica que la matriz que nos dan está ya transpuesta y podemos aplicar la correspondencia *natural* entre índices.

Ejemplo:

Si queremos aplicar el filtro descrito por la matriz $\begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{pmatrix}$

la forma de describirlo como una lista de listas será:

```
[[1,0,-1],[0,0,0],[1,0,-1]]
```

Valores fuera de rango. Imágenes en modo `I`.

Ya te habrás dado cuenta de que la aplicación de un kernel a una imagen puede hacer que los pixeles tengan valores por encima de 255. (En el ejemplo anterior esto pasa exáctamente en dos pixeles). Además algún kernel interesante tiene valores negativos haciendo que algunos pixeles del resultado tomen valores negativos. Para manejar estas situaciones disponemos del modo de imagen `I` que guarda en cada pixel un entero con signo en el rango -2147483648...2147483647 (más que suficiente para los casos que vamos a considerar).

Conversión. Del modo `I` al modo `L`.

Al mostrar una imagen en modo 'I' se debe hacer algo con los valores *fuera de rango*. El visor de imágenes (que la muestra cuando ejecutamos `img.show()`) considera los valores negativos como 0, y los mayores de 255 como 255. Sin embargo, al mostrar las imágenes en el terminal, Spyder *desecha* los valores *fuera de rango*, considerando todos como 0.

Nosotros convertiremos nuestras imágenes en modo `I` en imágenes en modo `L` aplicando una *normalización* como abajo.

He aquí un posible código de conversión:

In [1]:

```
def min_max(img):
    w,h = img.size
    minimo,maximo = img.getpixel((0,0)), img.getpixel((0,0))
    for x in range(w):
        for y in range(h):
            minimo = min(img.getpixel((x,y)),minimo)
            maximo = max(img.getpixel((x,y)),maximo)
    return minimo, maximo

def convert_to_L(img):
    """
    Devuelve img, de modo I, convertida a modo L.
    El valor mínimo de img pasa a ser 0, el máximo 255 y los valores intermedios s
e interpolan linealmente.
    """
    w,h = img.size
    result = Image.new('L',img.size)
    minimo,maximo = min_max(img)
    interval = float(maximo-minimo)
    for x in range(w):
        for y in range(h):
            result.putpixel((x,y),round(((img.getpixel((x,y))-minimo)/interval)*25
5))
    return result
```

Tarea a realizar

La práctica consiste en aplicar un filtro a una imagen dada. Por simplificar, consideraremos imágenes en blanco y negro.

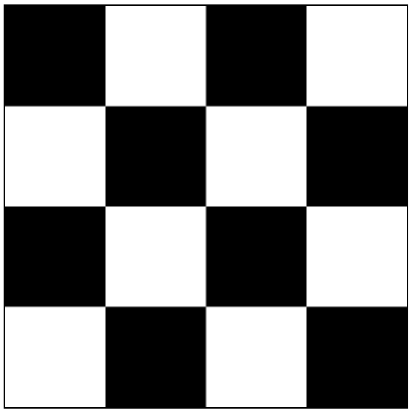
Escribir una función `filter_kernel(img, kernel)` que dada una imagen `img` en modo `L` y una matriz de convolución (filtro) `kernel`, devuelva el resultado de aplicar el filtro a la imagen, sustituyendo cada píxel por el resultado de aplicar la convolución.

Supondremos que el kernel es una matriz cuadrada de dimensión arbitraria pero siempre impar. Como se ha explicado antes, la imagen intermedia tiene que tener tipo 'I' pero la imagen resultado tiene que tener modo 'L'. Se puede (y es lo más recomendable) utilizar la función `convert_to_L` definida arriba para realizar este último paso. Para evitar problemas, la convolución sólo se aplicará en los píxeles en los que efectivamente se pueda aplicar, evitando los demasiado próximos a los bordes. El resto de los píxeles de la imagen que no se ven afectados por el kernel, por no poder aplicarse, deben mantenerse igual en la imagen resultante.

Ejemplos gráficos

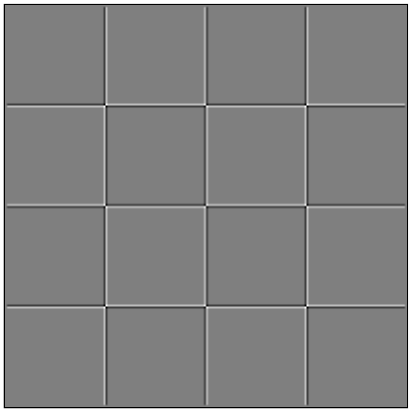
Veamos algunos ejemplos de los resultados de la función `filter_kernel` con algunas imágenes.

Comenzamos con una imagen muy sencilla. Tiene unas dimensiones de 200x200 píxeles y unos escaques o casillas de 50x50 píxeles.

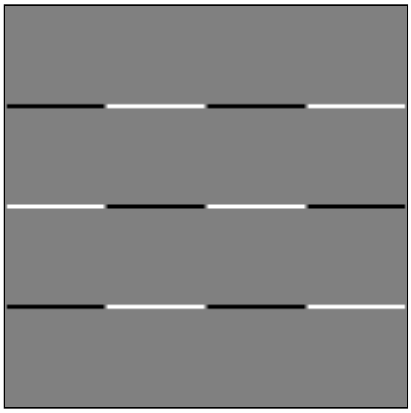


Supongamos que `img` es una variable que tiene la imagen anterior. Veamos el resultado de aplicar las siguientes funciones

- `filter_kernel(img, [[0,-1,0],[-1,4,-1],[0,-1,0]])`



- `filter_kernel(img, [[1,0,-1],[1,0,-1],[1,0,-1]])`



Terminamos con una imagen "natural".

El resultado de aplicar `filter_kernel(img, [[1, 1, 1],[1, 1, 1],[1, 1, 1]])` es

Y el de aplicar `filter_kernel(img, [[1,1,1],[0,0,0],[-1,-1,-1]])` es