Visión por Computador. Práctica 1.

Antonio Coín Castro

Curso 2019-20

Estructura del código

Se han desarrollado una serie de funciones genéricas de representación de imágenes, extráidas en su mayoría de la práctica 0 (aunque con ligeras modificaciones). Hay una serie de consideraciones a tener en cuenta:

- En general, todas las imágenes se convierten a números reales en el momento en que se leen, y solo se normalizan a $[0, 1]$ cuando se vayan a pintar.
- Las imágenes se pintan usando la librería `matplotlib`, empleando la técnica de *subplots* cuando sea necesario mostrar más de una imagen en la misma ventana.
- Tras mostrar una ventana de imagen, el programa quedará a la espera de una pulsación de tecla para continuar con su ejecución (incluso si se cierra manualmente la ventana del *plot*).
- Hay una serie de parámetros globales (editables) al inicio del programa para modificar ciertos comportamientos de algunas funciones.
- Se trabaja salvo excepciones con una única imagen de ejemplo en todo el programa, la cual debe estar en la ruta relativa `imagenes/`.
- Todas las funciones están comentadas y se explica el significado de los parámetros.

El programa desarrollado va ejecutando desde una función `main` los distintos apartados de la práctica uno por uno, llamando en cada caso a las funciones que sean necesarias para mostrar ejemplos de funcionamiento. Los parámetros de ejecución se fijan en estas funciones, llamadas `bonusX` ó `exXY`, donde `X` es el número de ejercicio e `Y` es el apartado.

Nota: Es posible que al ejecutar el programa aparezca el siguiente error: *Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers)*. Esto se debe a errores de redondeo en los cálculos, que hacen que las imágenes se salgan del rango $[0, 1]$ por valores despreciables del orden de 10^{18}.

Bonus 1

Comentamos primero el bonus 1 porque se utiliza en el resto de apartados de la práctica. Se ha implementado una función `convolution2D` que recibe una imagen y un *kernel* bidimensional y se encarga de:

a) Decidir si es separable en dos vectores unidimensionales, y
b) en caso afirmativo, encontrar dichos vectores y aplicar la convolución a la imagen con ellos.

Para la primera parte tenemos en cuenta el siguiente resultado de álgebra lineal:

Proposición. Sea A una matriz real $n \times m$. Entonces, $\text{rango}(A) = 1$ sí y solo sí existen vectores columna no nulos $u, v \in \mathbb{R}^m$ tales que $A = uv^T$.

Demostración. Recordamos que el rango de A puede verse como la dimensión de la imagen de la aplicación lineal asociada $x \mapsto Ax$, para $x \in \mathbb{R}^m$.

Para ver la primera implicación, basta observar que si $A = uv^T$, entonces $Ax = (uv^T)x = u(v^T x) = \langle v, x \rangle u$. Así, la imagen cae en una recta vectorial, luego tiene dimensión 1.

Recíprocamente, si A tiene rango 1 existe $u \in \mathbb{R}^m$ tal que $Ax = k(x)u$ para todo $x \in \mathbb{R}^m$, con $k(x) \in \mathbb{R}$. Aplicando esto a los vectores de la base usual de $\mathbb{R}^m$ obtenemos que cada columna de A es un múltiplo de u, digamos $A = u(k_1 \cdots k_m)$. Tomando $v = (k_1 \cdots k_m)^T$ tenemos que $A = uv^T$, como queríamos. □

Aprovecharemos también la descomposición en valores singulares de una matriz real [**1**]. Esta descomposición nos permite escribir $A = U\Sigma V^T$, donde U, V son matrices ortogonales y Σ es una matriz (rectangular) diagonal con valores no negativos. La propiedad fundamental que usaremos de esta descomposición es que el rango de A coincide con el número de elementos no nulos en la diagonal de Σ. En el caso en que A tenga rango 1, las primeras columnas de U y V (convenientemente ordenadas) serán los vectores que forman la descomposición de A como producto de dos vectores (el resto de columnas son nulas), salvo una constante multiplicativa. Si queremos recuperar exactamente la matriz A, notamos que

$$A = U \operatorname{diag}\{s_1, 0, \ldots, 0\} V^T \implies A = s_1 U_1 V_1^T,$$

donde U_1 y V_1 son las primeras columnas de U y V, respectivamente. Basta entonces tomar $u = \sqrt{s_1} U_1$ y $v = \sqrt{s_1} V_1^T$ para recuperar la matriz original como $A = uv^T$.

Utilizando esta descomposición, podemos a la vez comprobar que el *kernel* tiene rango 1 y encontrar los dos vectores por los que se factoriza. Empleamos la función `linalg.svd` de la librería `numpy` como sigue:

```
u, s, vt = np.linalg.svd(kernel)
rank = np.sum(s > EPSILON)
if rank == 1:
  vx = u[:,0] * sqrt(s[0])
  vy = vt[0] * sqrt(s[0])
```

Notamos que hemos considerado que los elementos de Σ son 0 si están por debajo de un cierto umbral, para subsanar posibles errores de redondeo a la hora de calcular la descomposición SVD.

Disponemos también una función `separable_convolution2D` que realiza la convolución con una imagen de dos vectores `vx` y `vy`, que representan las máscaras que se pasan por filas y por columnas (deben ser siempre de longitud impar). A la hora de implementarla, se realiza primero la convolución por filas y luego por columnas usando la función `convolve`. Esta función implementa la fórmula de la convolución de dos vectores `a` y `v`, y devuelve un vector convolucionado de la misma longitud que `a`:

$$(a * v)[n] = \sum_{i=-M}^{M} a[i] v[n-i].$$

Para intentar que sea eficiente, se construye una matriz `A` que se multiplica por el vector de máscara `v` recorrido al revés, de acuerdo con la fórmula anterior. Tenemos en cuenta que para 'encajar' los dos vectores y multiplicar debemos rellenar antes con 0 a ambos lados del vector `a`, tantos como indique la mitad de la longitud de la máscara (división entera). La construcción de la matriz `A` es como sigue:

```
for i in range(len(a)):
    rows.append(a[i:i + len(v)])
A = np.array(rows, dtype = np.double)
```

Para aplicar la convolución por filas y por columnas tenemos en cuenta que tenemos una imagen ampliada con bordes que luego descartaremos, por lo que tenemos que ajustar el recorrido de la matriz atendiendo al número de píxeles extra en los bordes de cada dimensión (valores `kx` y `ky`):

```
# Aplicamos la máscara por filas
for i in range(nrows + 2 * ky):
    im_res[i] = convolve(im_res[i], vx, kx)

# Aplicamos la máscara por columnas
for j in range(kx, ncols + kx):
    im_res[:,j] = convolve(im_res[:, j], vy, ky)
```

El último detalle que queda por mencionar es la estrategia seguida en los bordes de la imagen. Se proporciona una función `make_border` que devuelve una imagen ampliada con tantas filas y columnas de borde como sean necesarias, según la longitud de la máscara. Para ello se emplean las funciones `vstack` y `hstack` de `numpy`. En principio se han programado dos estrategias: rellenar los bordes con un valor constante (`BORDER_CONSTANT`) o replicar el borde a partir del último píxel (`BORDER_REPLICATE`):

```
BORDER_CONSTANT: iiiiii | abcdefgh | iiiiii
BORDER_REPLICATE: aaaaaa | abcdefgh | hhhhhh
```

En el caso de que la imagen sea una matriz tribanda, se realiza la convolución por separado en cada banda siguiendo el procedimiento mencionado, mediante la función `channel_separable_convolution2D`. Para separar en bandas empleamos las funciones `split` y `merge` de *OpenCV*:

```
channels = cv2.split(im)
im_res = cv2.merge([channel_separable_convolution2D(ch, vx, vy) for ch in channels])
```

Ejercicio 1

Apartado A

En este apartado mostramos el efecto de alisamiento con un *kernel* Gaussiano en una imagen. Como sabemos que la función Gaussiana es separable, basta considerar dos máscaras unidimensionales dadas por la función

$$G(x, \sigma) = exp\left\{\frac{-x^2}{2\sigma^2}\right\}.$$

Definimos entonces una función `gaussian_kernel1D` que nos devuelve una máscara Gaussiana unidimensional a partir de un parámetro σ (la desviación típica). Para ello, muestreamos los enteros del intervalo $[-3\sigma, 3\sigma]$, donde es conocido que se concentra casi toda la densidad de la función (el 99.73 %) [**2**]. Así, las máscaras resultantes tendrán un tamaño $2\lfloor 3\sigma \rfloor + 1$, que es siempre impar.

```
l = floor(3 * sigma)
gauss_ker = [gaussian(x, sigma) for x in range(-l, l + 1)]
gauss_ker = gauss_ker / np.sum(gauss_ker)
```

Es importante notar que devolvemos una máscara **normalizada**, en el sentido de que la suma de sus elementos es 1.

Mostramos ahora algunos ejemplos de convolución con máscaras Gaussianas (obtenidos a partir de la función `separable_convolution2D`), de forma que el alisado es isotrópico.

Vemos en la Figura 1 que, como era de esperar, a mayor valor de σ mayor emborronamiento se produce, pues se eliminan más altas frecuencias de la imagen perdiendo detalles en el proceso. Además, observamos que el tipo de borde empleado influye en el resultado final, pues en la alternativa de borde

Figura 1: Ejemplos de convolución con máscaras Gaussianas.

constante se aprecia un borde negro en la imagen resultante que no aparece en la alternativa con borde replicado. Los resultados son muy similares a los que se obtienen con una llamada a la función `GaussianBlur` de *OpenCV* con los mismos parámetros.

Pasamos ahora a realizar convoluciones con máscaras de derivada. Empleamos la aproximación por diferencias finitas para calcular las derivadas parciales de una imagen $f(x, y)$:

$$\frac{\partial f(x,y)}{\partial x} \approx f(x+1, y) - f(x, y).$$

Análogamente se puede aproximar la derivada con respecto a y, e iterando el proceso obtenemos derivadas de cualquier orden. Las máscaras de derivadas se obtienen mediante una llamada a la función de *OpenCV* `getDerivKernels`, que recibe como parámetros el orden de derivación en cada variable y el tamaño de las máscaras deseado. Internamente devuelve máscaras de Sobel para realizar la convolución [**3**], que estarán normalizadas (gracias al parámentro `normalize = True`).

Obtenemos la convolución con máscaras de derivadas mediante una llamada a la función `derivatives2D`. Un detalle a tener en cuenta es que se devuelve el valor absoluto de la imagen resultante, pues lo que nos interesa son los valores altos, tanto negativos como positivos (no queremos perder los valores negativos en la normalización posterior).

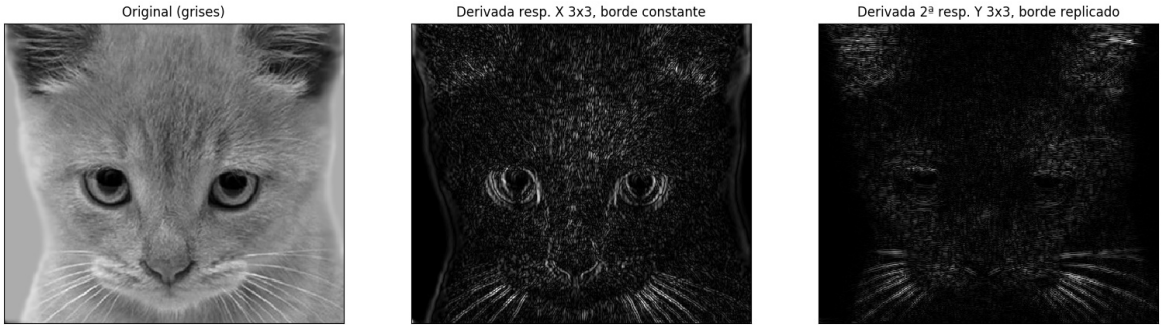

Figura 2: Ejemplos de convolución con máscaras de derivada.

Vemos que en la derivada respecto de x se aprecian los cambios en la dirección vertical, y que al aplicar la segunda derivada respecto de y se aprecian los cambios más bruscos de la imagen en la dirección horizontal. En este caso el efecto del borde apenas es visible sobre el fondo negro.

Apartado B

En este apartado perseguimos mostrar el resultado de aplicar un filtro Laplaciana-de-Gaussiana a una imagen. Este filtro consiste en aplicar primero un alisado Gaussiano para eliminar ruido y mejorar la visualización de la imagen, y después aplicar el operador Laplaciano. Si G denota el alisamiento Gaussiano, el resultado es:

$$\Delta(f) = G(f)_{xx} + G(f)_{yy}$$

La implementación dada esta fórmula es una consecuencia directa de lo que se vio en el apartado anterior:

```
im_smooth = gaussian_blur2D(im, sigma)
vxx, v = get_derivatives2D(2, 0, size)
u, vyy = get_derivatives2D(0, 2, size)
im1 = separable_convolution2D(im_smooth, vxx, v)
im2 = separable_convolution2D(im_smooth, u, vyy,)
laplacian = im1 + im2
```

Mediante este filtro podemos detectar algunos bordes, pues en los sitios donde haya un gran cambio de intensidad en una dirección, el operador será negativo a un lado y positivo al otro. De nuevo visualizamos el valor absoluto de la imagen resultante para no perder los valores negativos de cambio.

Notamos que este filtro ya está normalizado en escala, al estarlo el filtro Gaussiano que aplicamos inicialmente.

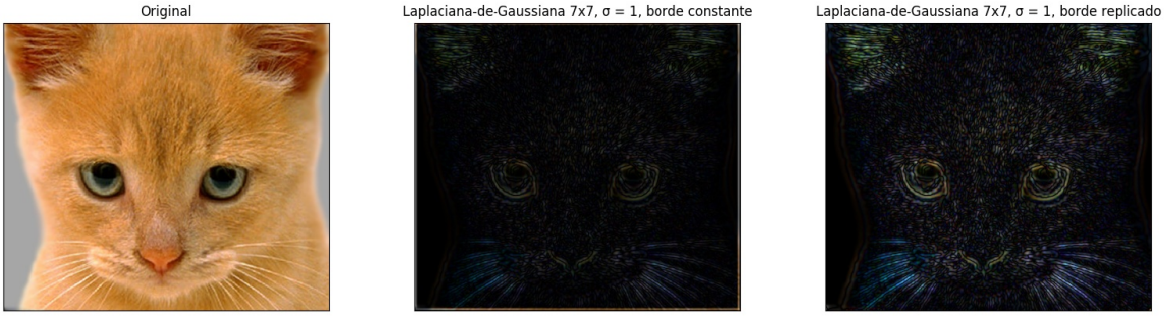

Figura 3: Ejemplos de filtrado con Laplaciana-de-Gaussiana.

En este caso observamos que el efecto del borde constante es bastante visible. Una cosa curiosa es que en este caso la imagen se ve más oscurecida, dando la sensación de que se extiende el efecto del borde al resto de la imagen. Este fenónemo puede parecer extraño, pero creo que puede deberse a que se detecta un cambio de gran intensidad en el borde (al ser constante) que eclipsa los cambios del resto de la imagen, y hacen que al normalizarla se vea más oscura. Si utilizamos la función `Laplacian` de *OpenCV* vemos que el resultado es similar.

Ejercicio 2

Apartado A

Queremos implementar ahora la visualización de la pirámide Gaussiana de una imagen, simulando cómo veríamos la imagen a distancia. Para implementarla, reducimos el tamaño de la imagen una octava cada vez hasta llegar al número de niveles deseado, alisando previamente con un filtro Gaussiano que elimine el ruido progresivamente.

Esta funcionalidad se implementa en la función `gaussian_pyramid`, que se apoya en la función `blur_and_downsample` para ir reduciendo el tamaño de la imagen. La reducción se consigue eliminando de la imagen las filas y las columnas pares:

```
nrows, ncols = im.shape[:2]
im_downsampled = np.array([im[i] for i in range(1, nrows, 2)])
im_downsampled = np.transpose([im_downsampled[:, j] for j in range(1, ncols, 2)])
```

Para la visualización de la pirámide se utiliza la función `format_pyramid`, inspirada en un ejemplo de la documentación de la librería `scikit-image` [**4**]. La estrategia consiste en construir una matriz que actúe como marco donde iremos incrustrando cada una de las imágenes que forman la pirámide. En principio se permite que el factor de reducción de una imagen a la siguiente sea un número real `k`, por lo que debemos adaptar el marco para que pueda almacenar todas las imágenes de la pirámide (añadiendo filas y columnas extra):

```
diff = np.sum([im.shape[0] for im in vim[1:]]) - nrows
extra_rows = diff if diff > 0 else 0
extra_cols = int(ncols / k)
```

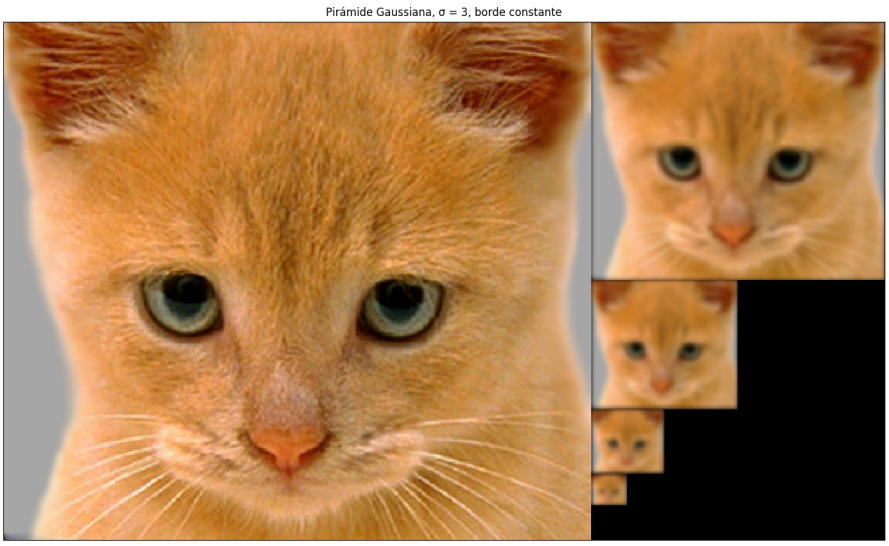

Figura 4: Ejemplo de pirámide Gaussiana de 4 niveles.

Como podemos apreciar, la primera imagen es la imagen original, y cada nueva imagen en la pirámide se ve más emborronada, reteniendo efectivamente las frecuencias bajas en cada paso como se esperaría. El efecto del borde constante también es visible, como ya lo era al aplicar un solo alisado Gaussiano a la imagen. En el último nivel ya apenas podemos apreciar detalles, debido también al tamaño tan pequeño que tiene la imagen.

Apartado B

Pasamos ahora a contruir una pirámide Laplaciana para una imagen. La construcción de esta pirámide se obtiene haciendo en cada paso la diferencia entre un nivel de la pirámide Gaussiana y el siguiente magnificado. Así, nos quedamos en cada nivel con frecuencias altas de la imagen, lo que nos permite ir detectando bordes en cada paso.

En el último nivel de la pirámide no hacemos la diferencia, sino que guardamos las frecuencias bajas de la imagen en dicho nivel. Esto nos permite reconstruir la imagen original a partir de la pirámide,

deshaciendo el proceso que hemos mencionado para su construcción. Es importante notar que el método usado para aumentar las imágenes en cada paso de la pirámide debe ser el mismo que se use para la reconstrucción. En nuestro caso se trata de una interpolación (bi)lineal obtenida mediante una llamada a `cv2.resize`, teniendo en cuenta que acepta como parámetro el nuevo tamaño de la imagen en el formato `(ancho, alto)`.

```
# x es la imagen en el paso anterior
im_upsampled = cv2.resize(im, (x.shape[1], x.shape[0]),
                          interpolation = cv2.INTER_LINEAR)
```

A la hora de la implementación podríamos haber reutilizado la función `gaussian_pyramid` del apartado anterior, pero no lo hemos hecho porque aquí permitimos que la reducción de tamaño de un nivel al siguiente sea un factor distinto de 2. El código para la construcción puede consultarse en la función `laplacian_pyramid`, y el de la reconstrucción en `reconstruct_im`. Ambos son autoexplicativos.

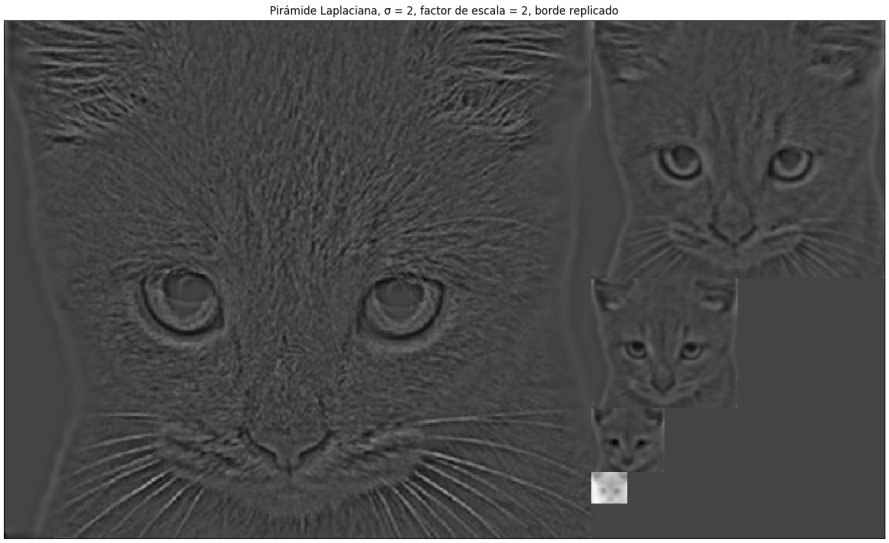

Figura 5: Ejemplo de pirámide Laplaciana de 4 niveles.

Observamos que efectivamente la pirámide está formada por las altas frecuencias de la imagen original (excepto el último nivel). En este ejemplo se ha usado borde replicado a la hora de convolucionar, por lo que el efecto en los bordes apenas es visible. Se puede comprobar que con borde constante se obtienen bordes negros alrededor de cada imagen, como cabría esperar visto lo que ocurría en apartados anteriores.

Vemos en la Figura 6 que al reconstruir la imagen la diferencia con la original no es exactamente 0, debido a fallos de redondeo en las operaciones:

Apartado C

En este apartado pretendemos mostrar una posible aplicación del filtro Laplaciana-de-Gaussiana para construir un espacio de escalas que posibilite la detección de regiones en imágenes **en escala de grises**. Para ello, consideramos un algoritmo que aplica en cada caso un filtro Laplaciana-de-Gaussiana a la imagen original cada vez con un alisamiento más agresivo, y guarda el cuadrado de la respuesta. El valor de σ que controla dicho alisamiento se ve multiplicado por una constante k = 1.2 en cada paso.

Una vez tenemos las distintas escalas de frecuencias altas, realizamos la estrategia de *supresión de no-máximos*. Esto consiste en mirar, para cada píxel en cada escala, el vecindario (cubo) formado por

Figura 6: Ejemplo de reconstrucción a partir de pirámide Laplaciana.

sus 9 vecinos adyacentes (contando a él mismo) en su propia escala, en la escala inferior y en la escala superior. Si el valor de la imagen en dicho píxel no es inferior al valor de la imagen de ninguno de sus 26 vecinos, guardamos la posición del píxel y su escala en una lista de índices. En la primera y la última escala solo se tienen en cuenta las escalas superior e inferior, respectivamente.

```
if im_scale[i, j] > THRESHOLD and np.max(neighbours) <= im_scale[i, j]:
  index_lst[p].append((i, j))
```

Un pequeño añadido es que solo consideramos aquellos píxeles que, además de ser máximos en su vecindario, superen un cierto umbral sobre la respuesta normalizada (`THRESHOLD` en el código), para quedarnos con las regiones más significativas. Notamos que al hacer el cuadrado de la respuesta en cada escala podemos encontrar regiones claras y oscuras buscando únicamente máximos.

Finalmente ordenamos los píxeles elegidos en cada escala de mayor a menor, y nos quedamos con una cantidad constante de ellos en cada escala (`NUM_MAX`, que por defecto es 1000). Una vez los tenemos, pintamos sobre la imagen original círculos con centro en los máximos elegidos y radio proporcional a la escala, mediante la función `cv2.circle`, a la que debemos pasarle el centro de los círculos al revés, es decir, `(j, i)` en vez de `(i, j)`.

```
blob = im.copy()
for p, lst in enumerate(index_lst):
  lst = sorted(lst, key = lambda x: scale_regions[p][x], reverse = True)
  for index in lst[:NUM_MAX]:
    radius = int(2 * scale_sigma[p])
    blob = cv2.circle(blob, index[::-1], radius, color)
```

Hemos elegido 6 colores distintos para las 6 primera escalas, que luego van ciclando en el caso de que haya más escalas. Todo este código se encuentra en la función `blob_detection`.

Observamos en la Figura 7 que la detección de regiones dista de ser perfecta, si bien parece que acierta al detectar los bordes más notables en las imágenes (por ejemplo, los bigotes, ojos y orejas del gato). Al aumentar el número de escalas se encuentran regiones que previamente no se detectaban. La elección inicial de σ también afecta a la cantidad de puntos detectada en las escalas, que disminuye más rápidamente conforme mayor sea σ. Esto tiene sentido, ya que se está realizando un alisado más agresivo que elimina los detalles de las imágenes.

Podríamos intentar ajustar mejor el valor del umbral (por defecto 0.05) y el número máximo de puntos que elegimos en cada escala, pues una pequeña variación en estos parámetros hace que las regiones detectadas sean completamente distintas.

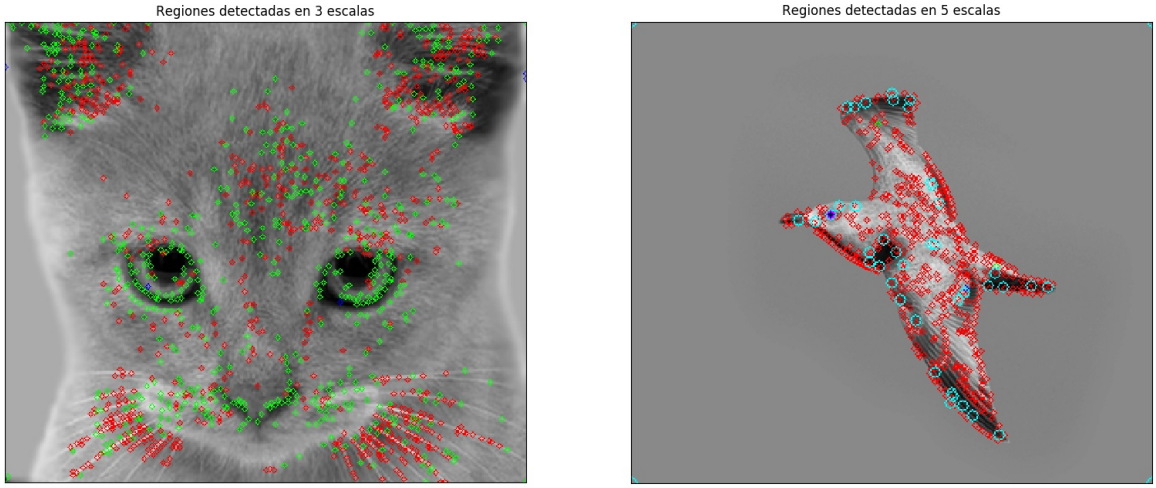

Figura 7: Ejemplo de detección de regiones partiendo de $\sigma = 1$.

Ejercicio 3

En este ejercicio la idea es experimentar con imágenes híbridas en blanco y negro obtenidas mezclando las frecuencias bajas de una imagen con las frecuencias altas de otra. Para extraer las frecuencias bajas utilizamos un filtro Gaussiano con parámetro σ_1, y para extraer las frecuencias altas restamos a la imagen original sus frecuencias bajas extraídas con un filtro Gaussiano de parámetro σ_2. La imagen final se obtiene sumando las frecuencias bajas y altas de cada imagen, como podemos ver en la función `hybrid_im`.

```
low_filter = gaussian_kernel1D(sigma1)
im1_low = separable_convolution2D(im1, low_filter, low_filter)
im2_high = im2 - separable_convolution2D(im2,low_filter, low_filter)
hybrid = im1_low + im2_high
```

La idea de una imagen híbrida es que permite distintas interpretaciones a distintas distancias, aprovechando la forma en la que nuestro cerebro percibe las imágenes. Desde cerca solo apreciamos las frecuencias más altas, pero conforme nos vamos alejando vamos perdiendo detalle y observando solo las frecuencias bajas.

Los parámetros σ_1 y σ_2 controlan con qué parte de frecuencias bajas y altas nos quedamos en cada imagen. A mayor valor de σ_1 más frecuencias altas eliminamos en la primera imagen, y a mayor valor de σ_2 más frecuencias bajas eliminamos en la segunda imagen. Es importante elegir estos parámetros por separado, y según las recomendaciones de **[5]**, se deben elegir de forma que las *frecuencias de corte* de los filtros tengan cierta separación y no se solapen. Se define la frecuencia de corte del filtro como el valor para el que se obtiene una respuesta de 0.5, que en nuestro caso se calcula como $f_c = \sigma\sqrt{2\log 2}$. En la ejecución del programa se imprimen las frecuencias de corte de cada filtro.

Para simular el efecto de alejarse, construimos una pirámide Gaussiana para las imágenes híbridas, en la que se debería apreciar únicamente la segunda imagen elegida (altas frecuencias) en el primer nivel, y únicamente la primera imagen elegida (bajas frecuencias) en el último nivel. Los parámetros para cada filtro se han elegido por experimentación.

Ejemplo 1: perro-gato

Para esta pareja de imágenes se ha decidido que la imagen del perro será la que conserve las frecuencias bajas, mientras que la del gato tendrá las frecuencias altas, pues tiene más detalles (los bigotes, los pelos, las orejas...). Elegimos los valores $\sigma_1 = 5$ y $\sigma_2 = 7$.

Figura 8: Imagen híbrida perro-gato, con $\sigma_1 = 5$ y $\sigma_2 = 7$.

Vemos como en la pirámide Gaussiana se aprecia lo que buscábamos: la primera imagen es prácticamente un gato, mientras que en el penúltimo nivel ya se observa la forma (un poco difusa) de un perro, sin haber rastro del gato.

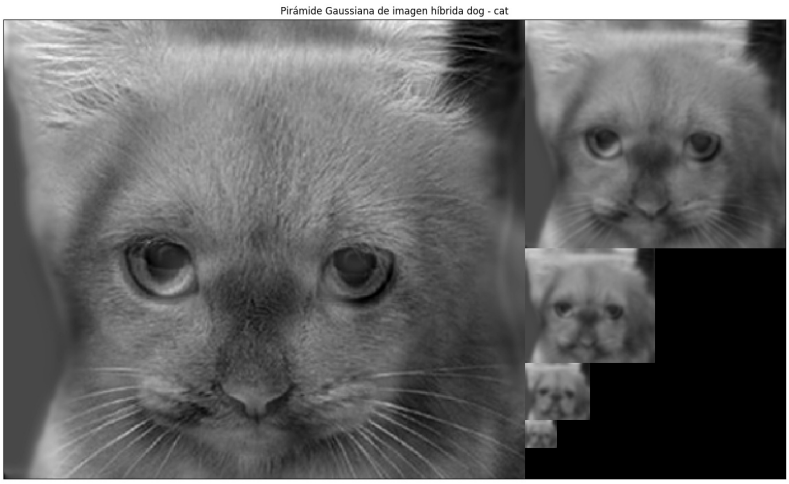

Figura 9: Pirámide Gaussiana para la imagen híbrida perro-gato.

Ejemplo 2: Marilyn-Einstein

En este caso mezclamos dos fotos de Marilyn Monroe y de Albert Einstein. Como la foto del segundo tiene más detalles, la elegimos como segunda imagen (frecuencias altas). Además, el contorno del pelo de Marilyn hace que sea una gran candidata a estar en el fondo, pues así se disimula mejor a corta distancia. En este caso elegimos $\sigma_1 = 3$ y $\sigma_2 = 7$.

Vemos como en la primera imagen se aprecian todos los rasgos de Einstein, y el pelo de Marilyn que sobresale se disimula como contorno de la cara y el cuello. Al aumentar un poco el nivel en la pirámide se eliminan los detalles de Einstein y apreciamos a Marilyn en su totalidad.

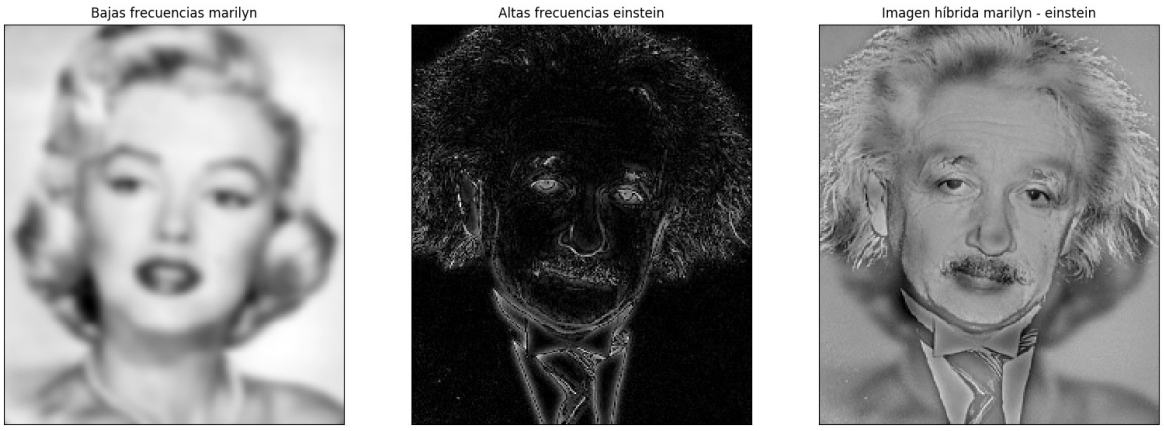

Figura 10: Imagen híbrida Marilyn-Einstein, con $\sigma_1 = 3$ y $\sigma_2 = 7$.

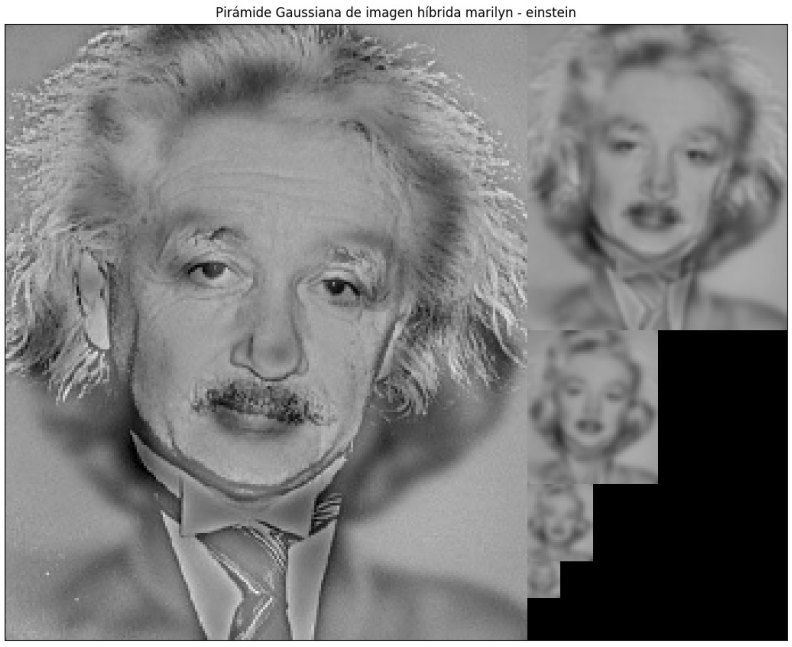

Figura 11: Pirámide Gaussiana para la imagen híbrida Marilyn-Einstein.

Ejemplo 3: submarino-pez

Elegimos el submarino como primera imagen (frecuencias bajas) ya que se camufla mejor con el fondo (es un objeto casi liso). Como la forma del pez coincide con la del submarino, este último hace las veces de contorno del pez, y no se nota demasiado de cerca. Elegimos $\sigma_1 = 3$ y $\sigma_2 = 7$.

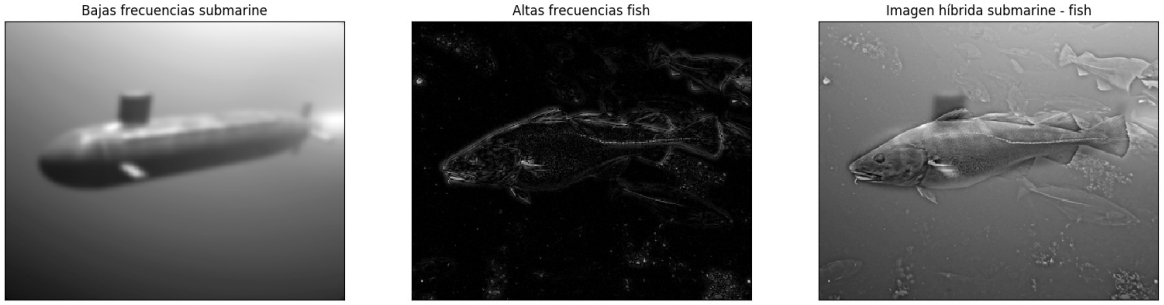

Figura 12: Imagen híbrida submarino-pez, con $\sigma_1 = 3$ y $\sigma_2 = 7$.

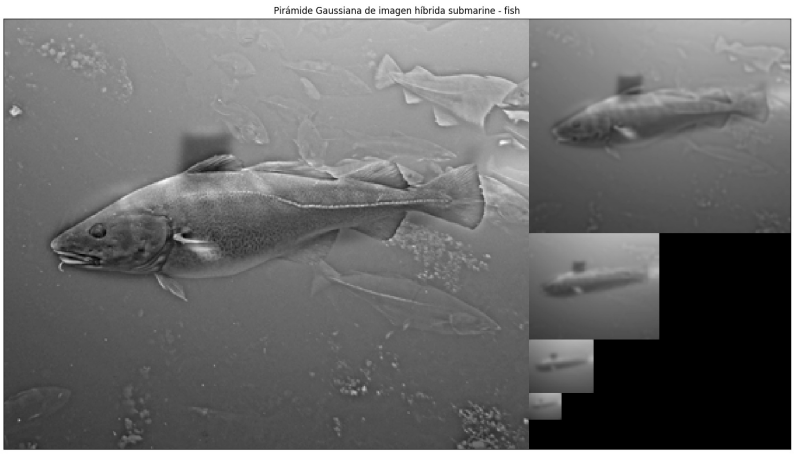

Figura 13: Pirámide Gaussiana para la imagen híbrida submarino-pez.

Bonus 2

Mostramos ahora todas las parejas de imágenes híbridas a color, excepto la pareja Marilyn-Einstein que ya estaba originalmente en blanco y negro.

Pareja perro-gato

En este caso volvemos a elegir la misma distribución de las imágenes, con la salvedad de que incrementamos ligeramente el alisado en la primera imagen, pues ahora el color del perro resalta más. Se elige $\sigma_1 = 6$ y $\sigma_2 = 7$.

Pareja submarino-pez

De nuevo elegimos la misma distribución de imágenes, y aumentamos también el alisado en la primera imagen, aunque los parámetros elegidos para la imagen en blanco y negro funcionan también bien. Se elige $\sigma_1 = 4$ y $\sigma_2 = 7$.

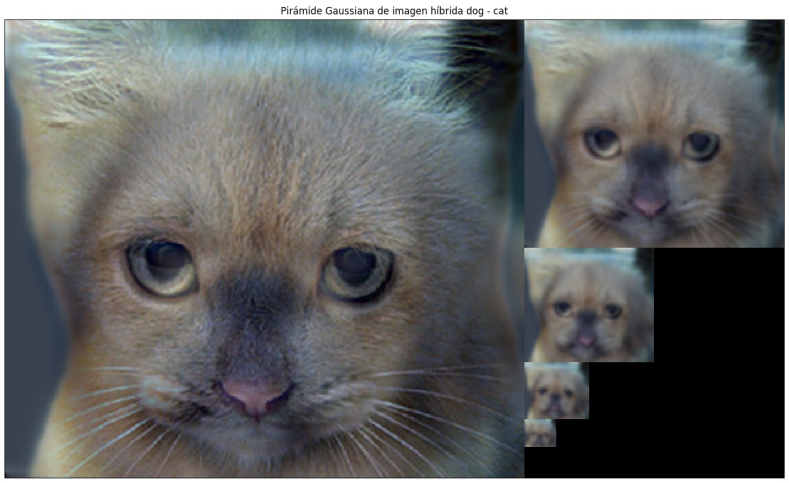

Figura 14: Pirámide Gaussiana de la imagen híbrida perro-gato.

Figura 15: Pirámide Gaussiana de la imagen híbrida submarino-pez.

Pareja avión-pájaro

En este caso hay que afinar un poco más, pues las dos siluetas no coinciden del todo. Intentamos arreglar esto emborronando mucho la imagen del avión para que de cerca apreciemos solo el pájaro. Parece que conseguimos más o menos lo que pretendemos con los valores $\sigma_1 = 9$ y $\sigma_2 = 7$.

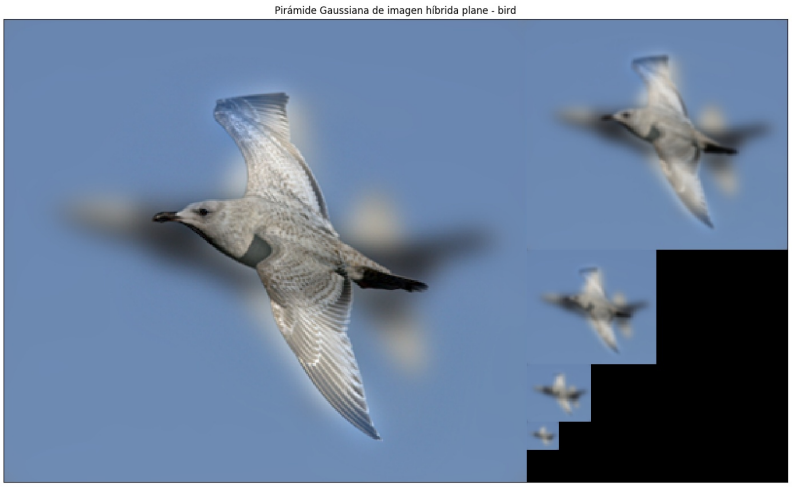

Figura 16: Pirámide Gaussiana de la imagen híbrida avión-pájaro.

Pareja moto-bici

Esta imagen ha sido la más difícil de hibridar. Probé primero poniendo la bici en el fondo, pero los resultados no fueron muy satisfactorios. Decidí poner la moto en el fondo porque era un poco más gruesa y al difuminarse se camuflaba mejor. También es un problema que ambas imágenes tengan colores muy vivos, ya que en la imagen híbrida se aprecia siempre una mezcla de ambos. Finalmente he elegido los valores $\sigma_1 = 7$ y $\sigma_2 = 9$.

Figura 17: Pirámide Gaussiana de la imagen híbrida moto-bici.

Bibliografía

[1] Singular value decomposition

[2] Normal distribution three-sigma rule

[3] Sobel filter

[4] Image pyramids with scikit-image

[5] Oliva, Torralba & Schyns. Hybrid images (2006)