

Visión por computador. Práctica 1.

José Carlos Martínez Velázquez

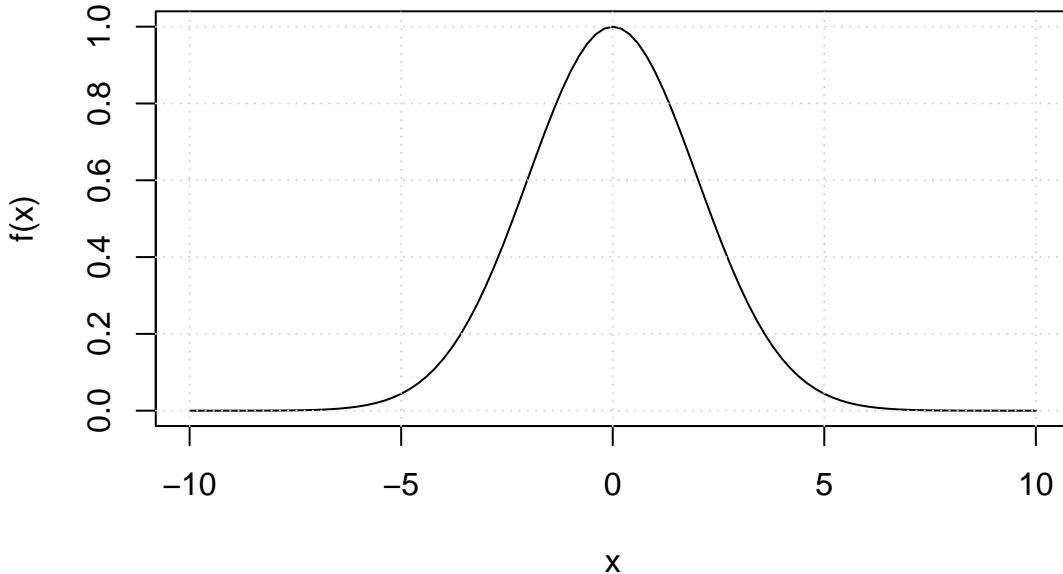
15 de octubre de 2016

A) Implementar una función de convolución my_imGaussConvol. debe ser capaz de calcular la convolución 2D de una imagen con una máscara. Ahora supondremos que la máscara es extraída del muestreo de una Gaussiana 2D simétrica. Para ello implementaremos las siguientes funciones auxiliares:

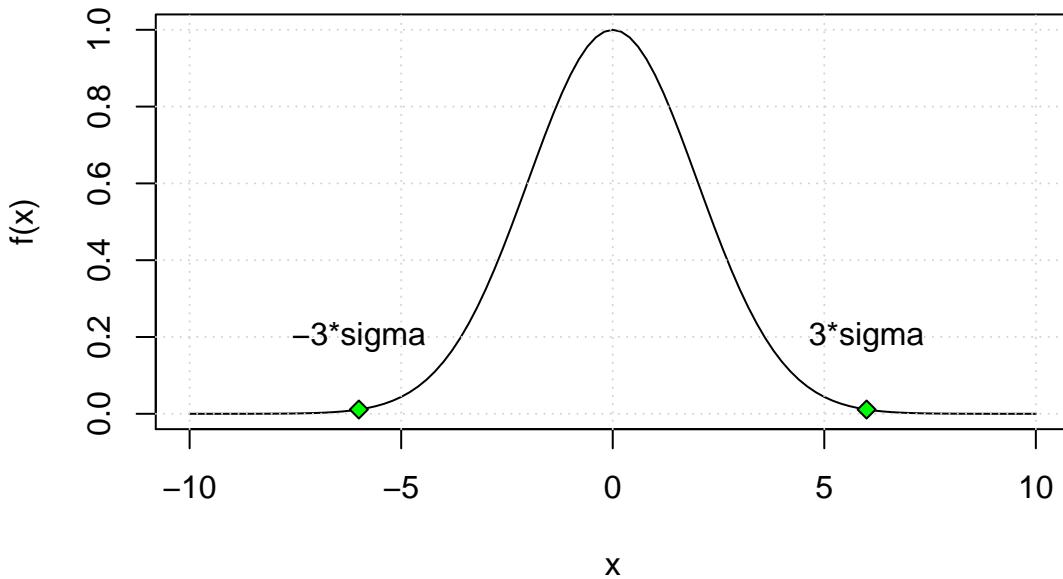
1).-Cálculo del vector máscara: Sea $f(x) = \exp(-0.5\frac{x^2}{\sigma^2})$ una función donde σ representa un parámetro en unidades píxel. Implementar una función que, tomando σ como parámetro de entrada devuelva una máscara de convolución representativa de dicha función. Justificar los pasos dados.

En primer lugar, debemos conocer la función que nos presentan. La función f es una Gaussiana con media 0 y sigma variable (la damos nosotros). Vamos a suponer un caso concreto, para entender bien cómo vamos a manipular la función en el caso general.

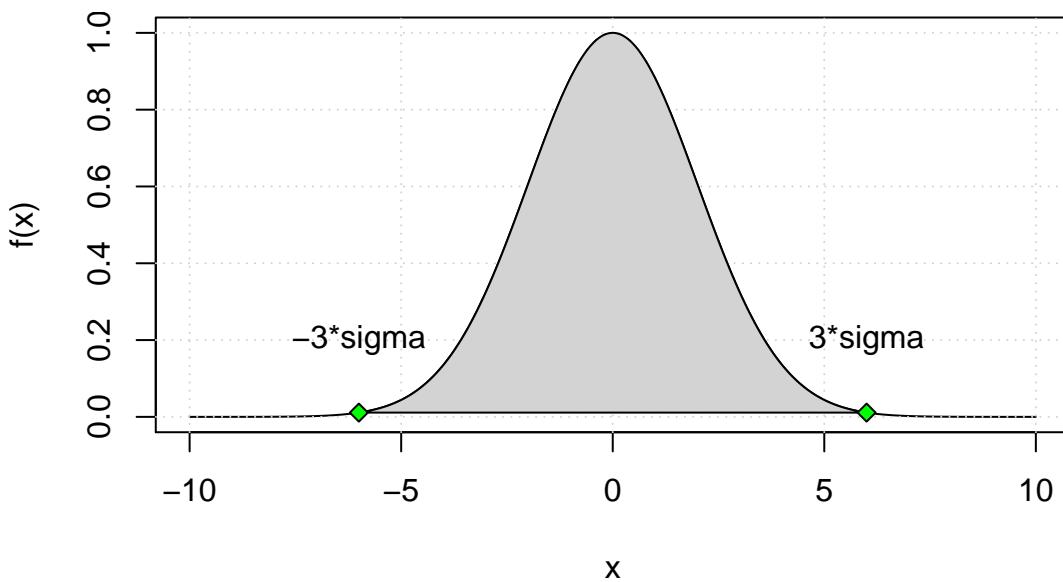
Supongamos $\sigma = 2$, entonces nuestra función es la siguiente:



El siguiente paso es definir qué área es interesante para nosotros, es decir, a partir de qué valores de x , $f(x)$ es considerada nula (0). Para nosotros este valor es constante y será, para todos los casos 3σ .



A partir de aquí, nuestro área de trabajo, es decir, lo que nos interesa de la función es el área sombreada. Podemos considerar que $\forall x < -3\sigma, f(x) = 0$. Del mismo modo, $\forall x > 3\sigma, f(x) = 0$.

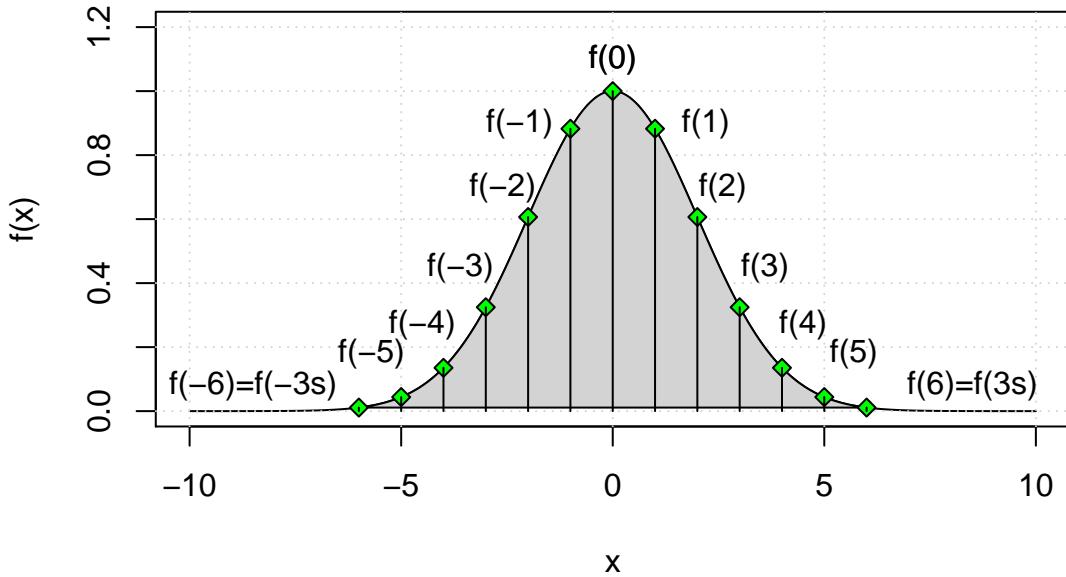


Llegados aquí, tenemos que discretizar la parte que nos interesa del área de seleccionada. Dado un valor de σ , nuestra máscara 1D contendrá $2(3 * \sigma) + 1$ valores, en nuestro caso de ejemplo, $2(3 * 2) + 1 = 13$. Recordemos que la máscara debe siempre ser siempre impar. El proceso es muy simple. Necesitamos tomar $n = 2(3 * \sigma) + 1$ muestras en un intervalo, pero ya conocemos una: $f(0)$, por lo que nos quedan que tomar $n - 1 = 2(3 * \sigma)$ muestras. La ventaja es que la función es simétrica, por lo que $f(a) = f(-a)$. Entonces, tenemos que tomar $\frac{n-1}{2} = \frac{2(3 * \sigma)}{2} = (3 * \sigma)$ muestras. Como tenemos que tomar $(3 * \sigma)$ muestras en el intervalo $[0, 3\sigma]$, la distancia entre cada valor muestreado en el eje x es de $\frac{3\sigma}{3*\sigma} = 1$

Todo esto justifica de forma analítica, que las muestras en el eje x deben tomarse de uno en uno, así, nuestra máscara estará formada por los siguientes valores:

$$f(-3\sigma), f(-3\sigma + 1), \dots, f(-1), f(0), f(1), \dots, f(3\sigma - 1), f(3\sigma)$$

Dado que las funciones Gaussianas son simétricas, entonces, $f(-a)=f(a)$. En lo que a cálculos se refiere, tenemos que hacer la mitad, luego sólo tenemos que calcular: $f(0), f(1), \dots, f(3\sigma - 1), f(3\sigma)$, con lo que obtendríamos la función discretizada como sigue:



Una vez obtenidos todos los valores que formarán parte de la máscara, hay que multiplicarlos por un factor de normalización `normFactor`. Este paso se hace para que los valores de la máscara sumen 1. La razón de hacer esto es para evitar deformaciones en la imagen. En todos los casos, el factor de normalización es igual a la inversa de la suma de los valores de la máscara, es decir: $\text{normFactor} = \frac{1}{\sum_{i=1}^k \text{mascara}_i}$.

Teniendo claros estos pasos, la implementación de la obtención de la máscara es sencilla. Sólo tenemos que codificar la función f , calcular el tamaño de la máscara, que será un vector de $2(3 * \sigma) + 1$ posiciones. La posición 0 estará justo en el medio de dicho vector, por lo que la posición cero será la parte entera de $p0 = \frac{2(3 * \sigma) + 1}{2}$. Entonces, `mascara1D[p0]=f(0,sigma)`. Ahora, hasta llegar a 3σ , no tenemos mas que, en un bucle, llenar los espacios que quedan: `mascara1D[p0+1]=mascara1D[p0-1]=f(1,sigma)`, ..., `mascara1D[p0+3*sigma]=mascara1D[p0-3*sigma]=f(3*sigma,sigma)`. Por último nos quedaría devolver `mascara1D` multiplicada por el factor de normalización `normFactor`. Ni que decir tiene que, con esta implementación la máscara forzosamente queda simétrica, como debe ser. Esta función se llamará con la sentencia `getMask(sigma)` y su código, en Python sería implementado como sigue:

"""

*La función `getMask` recibe como parámetro `sigma`, es decir el número de pixeles (número de elementos) que tendrá la máscara. Entenderemos como relevantes aquellos valores que se encuentran como máximo a $3 * \sigma$ de distancia de la media (0). `GetMask` devuelve un array con `sigma` elementos donde cada elemento contiene el valor de f evaluado en la discretización de la función.*

Un ejemplo. Supongamos f con $\sigma=5$. `GetMask` devolvería un vector de 5 elementos, que serían: $|f(-15,5)|/f(-7.5,5)|/f(0,5)|/f(7.5,5)|/f(15,5)|$.

Hay que multiplicar por un factor de normalización para que el valor de la suma de todos los elementos de la máscara sumen 1.

"""

```
def getMask(sigma):
```

```

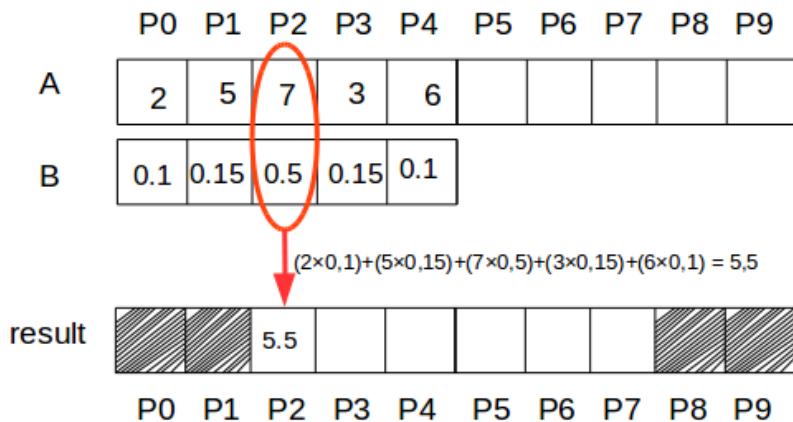
sigma=abs(sigma)
limit=3*sigma #DEFINIMOS EL LIMITE DE LA FUNCION F EN 3 SIGMAS
#La longitud de la mascara va a ser 2*LIMITE+1, asi forzamos que sea impar
mask1D=[0.0 for i in xrange((2*int(limit))+1)]
lenMask=len(mask1D)
positionzero=int(lenMask/2)
mask1D[positionzero]=f(0,sigma)
suma=mask1D[positionzero]
for i in xrange(positionzero+1,lenMask):
    resultF=f(i-positionzero,sigma)
    mask1D[i]=resultF
    mask1D[(lenMask-1)-i]=resultF
    suma+=2*resultF
normFactor=(1/suma)
finalMask=[x * normFactor for x in mask1D]
return finalMask

```

2).- Implementar una función que calcule la convolución de un vector señal 1D con un vector máscara 1D de longitud inferior al de la señal usando dos posibles tipos de condiciones de contorno (uniforme a ceros y reflejada). La salida será un vector de igual longitud que el vector señal de entrada.

2.1) Paso 1: Implementar una función que calcule la convolución 1D entre dos vectores, dando de salida sólo los valores donde ha sido posible el cálculo.

Para calcular la convolución 1D entre dos vectores necesitamos hacer una media ponderada por cada pixel en donde quepa la máscara. El algoritmo es muy sencillo. Supongamos dos vectores A y B, donde la longitud de B es menor que la de A. En cada paso del algoritmo de convolución consideramos el pixel de A que señala la posición central de B y calculamos la media ponderada del pixel. El resultado de esta operación será el valor de la convolución para ese pixel, tal y como muestra la figura.



Como vemos, si colocaramos la posición central de B en el primer o segundo elemento de A, el cálculo no podría ser realizado. Del mismo modo, ocurre por el final. Entonces la primera posición en la que se puede realizar el cálculo coincide con la posición del elemento central de B (máscara), en el caso del ejemplo, la posición 2 (o lo que es lo mismo, que en las dos primeras y en las dos últimas posiciones el cálculo no puede ser realizado) es la primera en la que el cálculo puede ser realizado. Del mismo modo ocurre por el final.

Entendiendo el problema, la implementación consiste en calcular la primera posición útil y la última: `startposition=parte_entera(longitud(B)/2)` y `finishposition=(longitud(A)-1)-startposition`.

A continuación colocamos el vector B con la posición central en `startposition` (basta con emparejar ambos vectores por el principio) y con un bucle la vamos moviendo de una en una posición hasta que la posición central de B esté emparejada con la última posición útil de A (que ambos vectores estén emparejados por el final), haciendo los cálculos pertinentes. Una vez el vector resultado ha sido calculado, basta con devolver lo que contiene el vector resultado desde la posición `startposition` hasta la posición `finishposition` y habremos conseguido lo que buscábamos. Esta función se llamará mediante la sentencia `convolution2Vectors(mask, vect)` y se implementaría en Python como sigue:

```
"""
La funcion convolution2Vectors
coge un vector y una mascara y calcula el filtrado del vector
con respecto de dicha mascara.

El tamaño de la mascara debe ser impar, pues el pixel central
es el que marca que pixel es para el que estamos calculando
los valores
"""

def convolution2Vectors(mask, vect):
    startPosition=len(mask)/2
    finishPosition=len(vect)-startPosition
    result=[sum([float(a*b) for a,b in zip(
                    mask,
                    vect[i-startPosition:i+startPosition+1]
                )])
        for i in xrange(startPosition,finishPosition)]
    return result
```

2.2) Definir un vector auxiliar de longitud mayor definida a partir de las dimensiones de los dos vectores de entrada. Por ejemplo, la señal tiene longitud N y el otro $2k+1$, entonces el vector auxiliar será $N+2k$. Copiar en su centro el vector señal y llenar los extremos usando el criterio de borde elegido.

Como hemos visto en el apartado 2.1, si queremos realizar el cálculo en toda la señal completa (vector A), no podíamos, pues había posiciones donde no se podía realizar el cálculo. Para solucionar esto, vamos a meter relleno por ambos lados del vector señal de forma que la posición central de la máscara (vector B) sea colocada en la posición 0 de la señal pudiendo así realizar los cálculos que faltaban.

Para introducir relleno, no nos hace falta mucho más de lo que tenemos. De hecho, calculando la variable `startposition` como en el apartado 2.1, sabemos cuántas posiciones de relleno hay que añadir por cada extremo del vector señal. Entonces, dado un vector señal y un vector máscara, calculamos un vector auxiliar de tantas posiciones como tenga el vector señal mas dos veces `startposition` (una vez al principio y otra al final). Es decir: `aux = [0.0] * (longitud(señal) + 2*startposition)`. (Este ejemplo inicializaría el vector auxiliar a 0). Ahora lo que queda es copiar el contenido del vector señal en las posiciones desde `startposition` hasta `finishposition` y devolver dicho vector auxiliar.

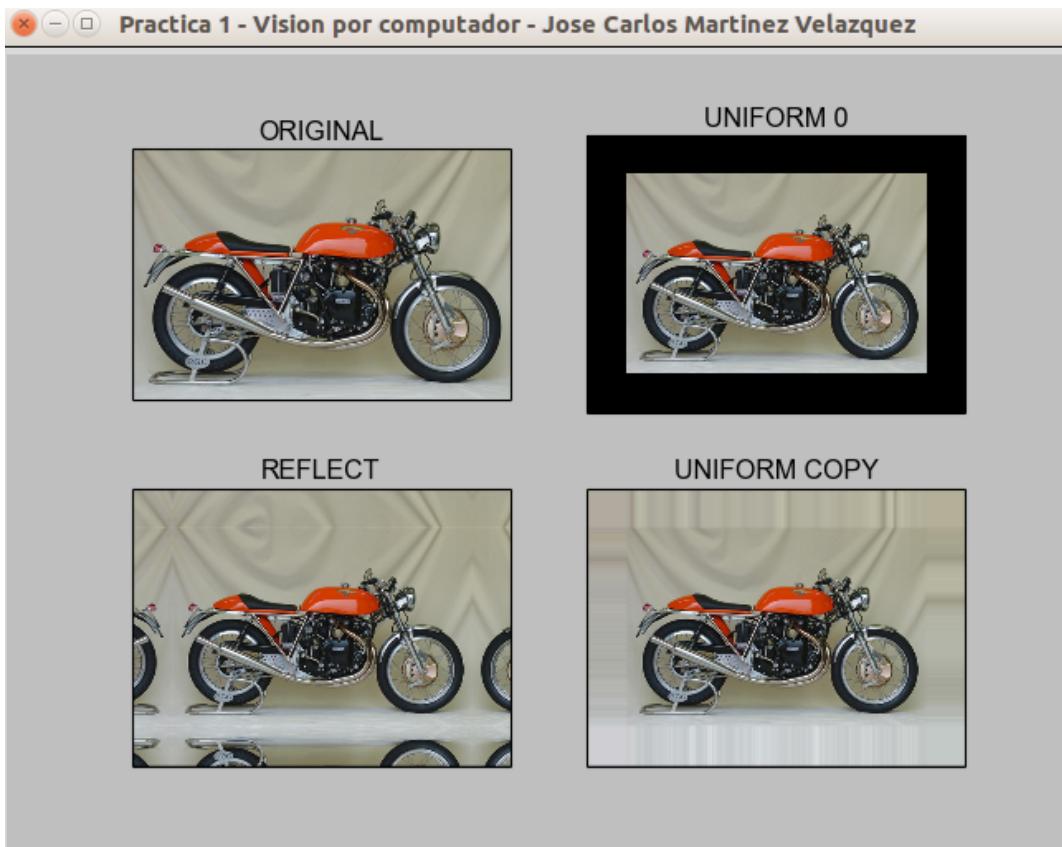
En el ejemplo concreto de antes, $N = 9$ y $2k + 1 = 5 \Rightarrow 2k = 4$. Calculando `startposition`, tenemos que $startposition = \text{parte_entera}(N/2) = 4 = 2k$. Lógicamente, el vector auxiliar tiene que tener una longitud de $N + 2k = N + 2 * startposition = 13$.

Una vez obtenido el vector auxiliar, las diferentes formas de llenar los espacios introducidos son:

- Uniforme a ceros: No hay que hacer nada, simplemente si se ha inicializado el vector auxiliar a ceros, basta con devolverlo teniendo copiado la señal en su centro como acabamos de indicar.

- Reflejada: Consiste en extraer el vector de píxeles que va desde `startposition` hasta $2 * startposition$, darle la vuelta y colocarlo desde la posición 0 hasta `startposition-1`. Del mismo modo, por el final, se extrae el vector de píxeles que va desde `finishposition-startposition` hasta `finishposition`, darle la vuelta y colocarlo desde `finishposition+1` hasta `longitud(señal)-1`.
- Copia del primer y último pixel (no se pide): Consiste el coger el valor de la posición `startposition` y copiarlo desde la posición 0 hasta `startposition-1`. Del mismo modo, se coge el valor de la posición `finishposition` y copiarlo desde `finishposition+1` hasta la posición `longitud(señal)-1`.

Para hacernos una idea de cómo quedan los diferentes tipos de bordes sobre una imagen, no hay mejor manera que verlo:



Esta función se llamará mediante la sentencia `createAuxVector(mask, vect, borderType)` y su código en Python sería el siguiente:

```
"""
La funcion createAuxVector corresponde al segundo paso del apartado 2).
De lo que se trata es de copiar los bordes ya sea constante a
cero o con reflejo o copia del ultimo pixel.
"""

def createAuxVector(mask, vect, borderType):
    result=np.array([0 for i in xrange(len(vect)+(len(mask)-1))])
    startPosition=len(mask)/2
    finishPosition=len(result)-startPosition
    result[startPosition:finishPosition]=vect
```

```

#if(borderType==0): #Borde a ceros -> esta comentado porque no se haria nada

if(borderType==1): #Borde reflejo
    result[0:startPosition]=result[2*startPosition:startPosition:-1]
    result[finishPosition:-1]=result[finishPosition-1:finishPosition-startPosition:-1]

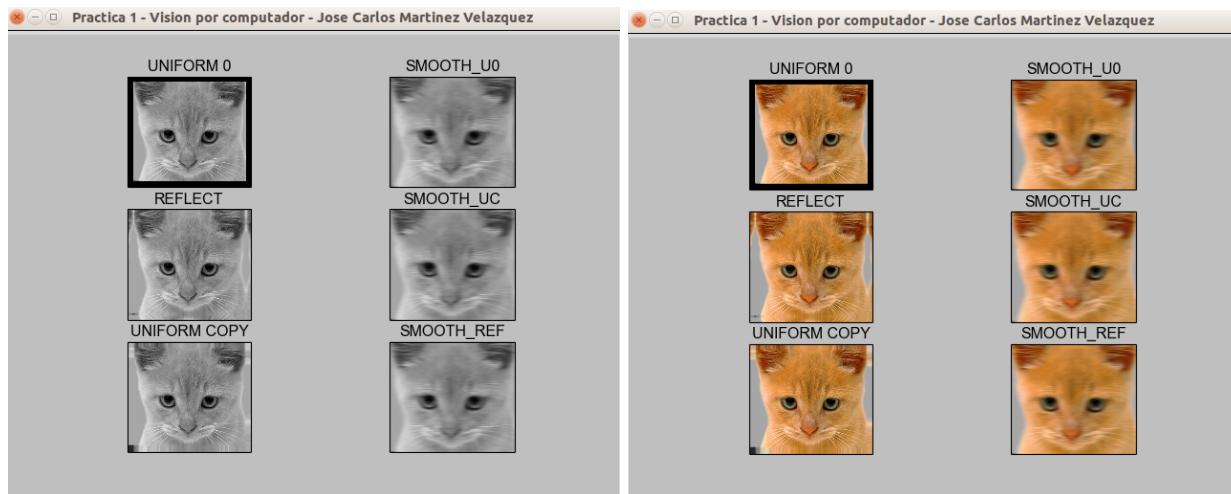
elif(borderType==2): #Borde copia
    result[0:startPosition]=result[startPosition]
    result[finishPosition:-1]=result[finishPosition-1]

return result

```

2.3) Ejecutar la función del paso 2.1 sobre el vector creado y la máscara. 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 a montar el vector de salida. Usar las funciones split() y merge() de opencv.

Para no preocuparme de la diferencia entre color y escala de grises e construir las funciones de forma universal, todas las implementaciones se basan en la idea de que una imagen en escala de grises no es más que una imagen en color donde las capas R,G,B (B,G,R en OpenCV) son todas iguales, es decir, cada pixel tiene el mismo valor en las tres capas. Vamos a suponer una imagen completa, representada por una matriz. Cada fila es un vector de señal, entonces, vamos a aplicar convolución 1D a cada fila para conseguir una convolución en horizontal de la imagen. Aplicamos el algoritmo para escala de grises y color, así como para cada tipo de borde:



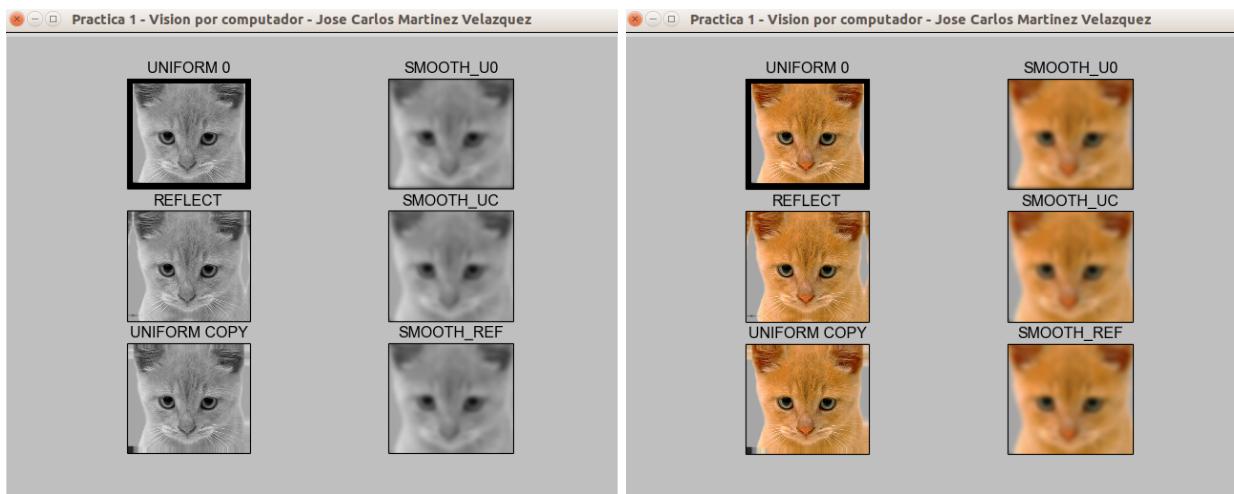
Es muy evidente que la convolución se ha aplicado sólo en horizontal, pues los rasgos sólo se notan en esta dirección. De hecho, en algunos casos se nota como una especie de “ensanchado” de la imagen. Hemos conseguido realizar una convolución 1D a la imagen.

3).- Implementar una función que tomando como entrada una imagen y el valor de sigma, calcule la convolución de dicha imagen con una máscara Gaussiana 2D. Usar las funciones implementadas en el punto anterior. Recordar que la Gaussiana es descomponible en convoluciones 1D por filas y columnas.

Esta función es un compendio de todo lo que hemos visto hasta ahora. En primer lugar, se obtiene la máscara con la llamada a la función `mask=getMask(sigma)`. Ahora separamos las capas con

`r,g,b=cv2.split(imagen)`. Por cada capa R,G,B, debe seguirse el siguiente proceso: convertimos los valores de la capa al tipo `CV_32FC3`, es decir, flotantes de 32 bits, pues todos nuestros cálculos serán con números en coma flotante. A continuación, para cada fila (`vect`), obtendremos el vector auxiliar, con la llamada a la función `auxvec = createAuxVector(mask, vect, borderType)` al que se le aplicará la convolución 1D con la llamada a la función `convolution2Vectors(mask, auxvec)`. Una vez hecha la convolución 1D en horizontal, hay que hacerle la convolución 1D en vertical al resultado. Para ello basta con trasponer las capas (son matrices) y volver a aplicar el proceso de convolución 1D horizontal a cada una. Una vez terminado se vuelve a trasponer, para devolver cada capa a su estado original, se regresa al tipo `CV_8UC3` (valores enteros) para convertirla en una imagen que se pueda “pintar” y se reagrupa la imagen completa con `imagen=cv2.merge([r,g,b])`. Ahora la imagen puede ser mostrada.

Vamos a ver un ejemplo de imágenes convolucionadas por completo:



A diferencia de cómo lo haciamos antes, ahora mantiene la forma de la imagen original aunque borrosa, esto es, hemos eliminado frecuencias altas.

Para ver la diferencia más evidente entre una imagen convolucionada sólo en horizontal y completamente veamos este ejemplo.



El código en Python para hacer una convolución es el siguiente:

```
"""
La funcion my_imGaussConvol realiza la convolucion de una imagen.
Los parametros que se usan son la imagen que se quiere convolucionar,
el sigma y el tipo de borde: 1 -> borde reflejo,
2-> borde copia, cualquier otro valor -> borde negro
"""
def my_imGaussConvol(image,sigma,bordertype):
    mask=getMask(sigma)
    r,g,b=cv2.split(image)
```

```

#Trabajamos en modo CV_32FC3 (CALCULOS EN COMA FLOTANTE)
r=np.float32(r)
g=np.float32(g)
b=np.float32(b)

for i in xrange(len(r)):
    r[i]=convolution2Vectors(mask,createAuxVector(mask,r[i],bordertype))
    g[i]=convolution2Vectors(mask,createAuxVector(mask,g[i],bordertype))
    b[i]=convolution2Vectors(mask,createAuxVector(mask,b[i],bordertype))

r=np.transpose(r)
g=np.transpose(g)
b=np.transpose(b)

for i in xrange(len(r)):
    r[i]=convolution2Vectors(mask,createAuxVector(mask,r[i],bordertype))
    g[i]=convolution2Vectors(mask,createAuxVector(mask,g[i],bordertype))
    b[i]=convolution2Vectors(mask,createAuxVector(mask,b[i],bordertype))

r=np.transpose(r)
g=np.transpose(g)
b=np.transpose(b)

#Regresamos al modo CV_8UC3 (ENTERO)
r=np.uint8(r)
g=np.uint8(g)
b=np.uint8(b)

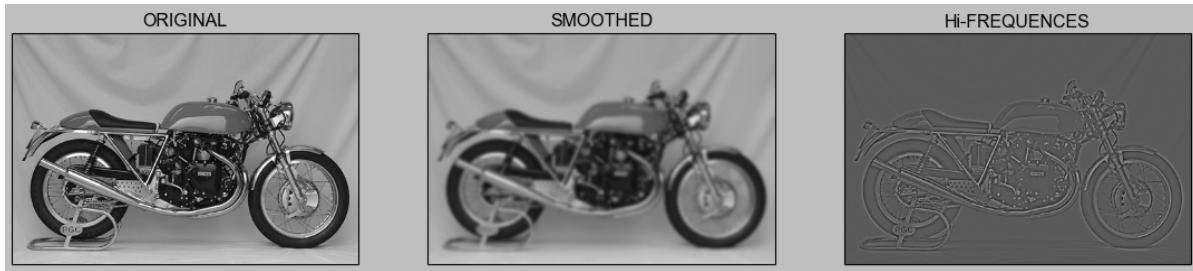
imgi=cv2.merge([r,g,b])
return imgi

```

B) Imágenes híbridas. Usar la función que hemos implementado en el apartado A para elegir los sigmas más adecuados para la selección de frecuencias en parejas de imágenes.

1) Implementar una función que genere las imágenes de baja y alta frecuencia.

La dificultad para el algoritmo de generar una imagen de baja frecuencia habiendo superado los apartados anteriores, radica única y exclusivamente en encontrar en un valor de σ adecuado para engañar al ojo humano al construir la ilusión, pues generar una imagen en baja frecuencia no es más que aplicarle convolución 2D a la imagen original con dicho valor de σ . Para generar una imagen en alta frecuencia, basta con restar a la imagen original, una imagen a bajas frecuencias y normalizar el resultado entre 0 y 255. El aspecto que tendría una imagen a altas frecuencias sería algo parecido a esto:



El código en Python para conseguir este tipo de imágenes sería el siguiente:

```
"""
La funcion getHighFrequencies toma dos veces la misma imagen,
la original y la convolucionada. Resta la convolucionada a
la imagen original y obtenemos los detalles (altas frecuencias).
Se puede aplicar un filtro Laplaciano, por lo que se pasa un coeficiente
Laplaciano.
"""
def getHighFrequencies(image,imageconv,hFfactor):
    r,g,b=cv2.split(image)

    #Trabajamos en modo CV_32FC3 (CALCULOS EN COMA FLOTANTE)
    r=np.float32(r)
    g=np.float32(g)
    b=np.float32(b)

    rc,gc,bc=cv2.split(imageconv)

    rc=np.float32(rc)
    gc=np.float32(gc)
    bc=np.float32(bc)

    r=hFfactor*r-rc
    g=hFfactor*g-gc
    b=hFfactor*b-bc

    r=normalize(r,0,255)
    g=normalize(g,0,255)
    b=normalize(b,0,255)

    #Regresamos al modo CV_8UC3 (ENTERO)
    r=np.uint8(r)
    g=np.uint8(g)
    b=np.uint8(b)

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

    return finalimage
```

La función de normalizado `normalize()` es el algoritmo de mapeo lineal y se hace por cada capa. Se obtendría el máximo y el mínimo de la capa (`max` y `min` actual) y, teniendo los nuevos máximo y mínimo al intervalo que queremos mapear, se aplica la siguiente expresión:

$$\text{valor_normalizado} = \frac{(\text{valor_actual} - \text{min_actual})(\text{max_nuevo} - \text{min_nuevo})}{(\text{max_actual} - \text{min_actual})} + \text{min_nuevo}$$

El código en Python del normalizado es el siguiente:

```
"""
La funcion normalize mapea un valor en un intervalo conocido al mismo valor
medido en un nuevo intervalo.
"""

def normalize(shape,min_v,max_v):
    if(min_v<max_v):
        maxvalue=256
        minvalue=0
        nshape=np.array([[0]*len(shape[0])]*len(shape))
        for i in xrange(len(shape)):
            for j in xrange(len(shape[0])):
                if(shape[i][j]>maxvalue):
                    maxvalue=shape[i][j]
                if(shape[i][j]<minvalue):
                    minvalue=shape[i][j]
        for i in xrange(len(shape)):
            for j in xrange(len(shape[0])):
                nshape[i][j]=((shape[i][j] - minvalue)*(max_v-min_v))
                / (maxvalue-minvalue) + min_v
        return nshape
    else:
        raise ValueError, "min value in range must be smaller than max value in range."
```

A continuación, para obtener la imagen híbrida basta con sumar los pixeles de cada imagen y obtener la media. El código en Python sería el siguiente:

```
"""
La funcion getHybridImage construye una imagen hibrida
a partir de dos imagenes. El primer parametro debe ser
una imagen correspondiente a las altas frecuencias
(extraidas previamente) y el segundo, una imagen alisada
convenientemente.
"""

def getHybridImage(imageHF,imageLF):
    #Trabajamos en coma flotante CV_32FC3
    imgfh=np.float32(imageHF)
    imglf=np.float32(imageLF)
    hybridImage=(imgfh+imglf)/2
    #Volvemos a entero CV_8UC3
    return np.uint8(hybridImage)
```

2) Escribir una función para mostrar las tres imágenes (alta, baja e híbrida) en una misma ventana.

El proceso es sencillo. Cargamos las dos imágenes que hacen falta en el color deseado. A continuación alisamos y obtenemos la imagen de alta frecuencia y la de baja. Hacemos la media de ambas imágenes y ya tendríamos el resultado deseado. El código es el siguiente.

```

"""
Muestra solo la imagen de altas frecuencias, la de bajas frecuencias y la imagen hibrida.
"""

def showConstructionHybridImage(rutaAltas,colorAltas,sigmaAltas,
                                  rutaBajas,colorBajas,sigmaBajas,
                                  factorLaplaciano,border):

    imagenAltas=loadImage(rutaAltas,colorAltas)
    imagenBajas=loadImage(rutaBajas,colorBajas)

    imconv=my_imGaussConvol(imagenAltas,sigmaAltas,border)
    imAltas=getHighFrequencies(imagenAltas,imconv,factorLaplaciano)

    imBajas=my_imGaussConvol(imagenBajas,sigmaBajas,border)

    hybridimage=getHybridImage(imAltas,imBajas)

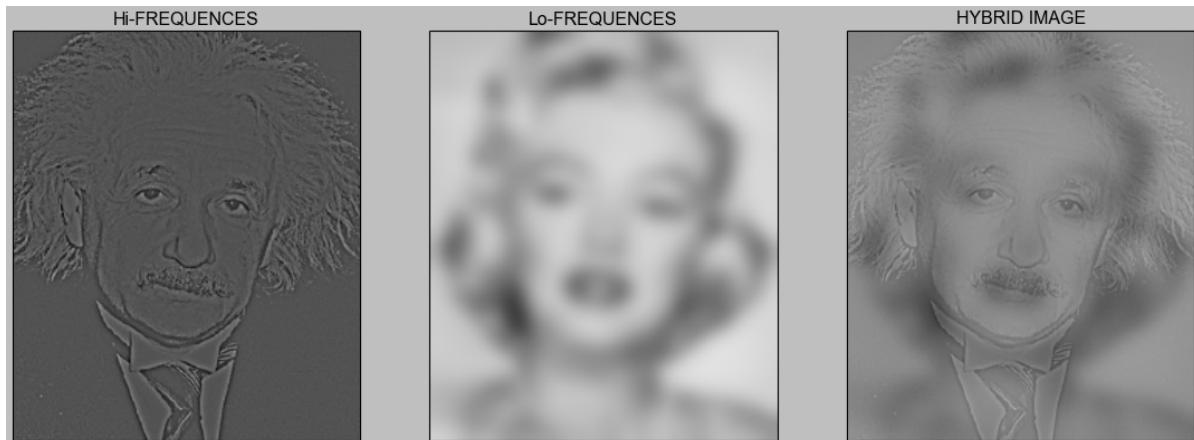
    print("Esta viendo el proceso de construccion de la imagen hibrida,
          cierre la ventana para continuar.")

    paintMatrixImages(
        [[imAltas,imBajas,hybridimage]],
        [["Hi-FREQUENCES","Lo-FREQUENCES","HYBRID IMAGE"]],
        "Practica 1 - Vision por computador"
    )

    return hybridimage

```

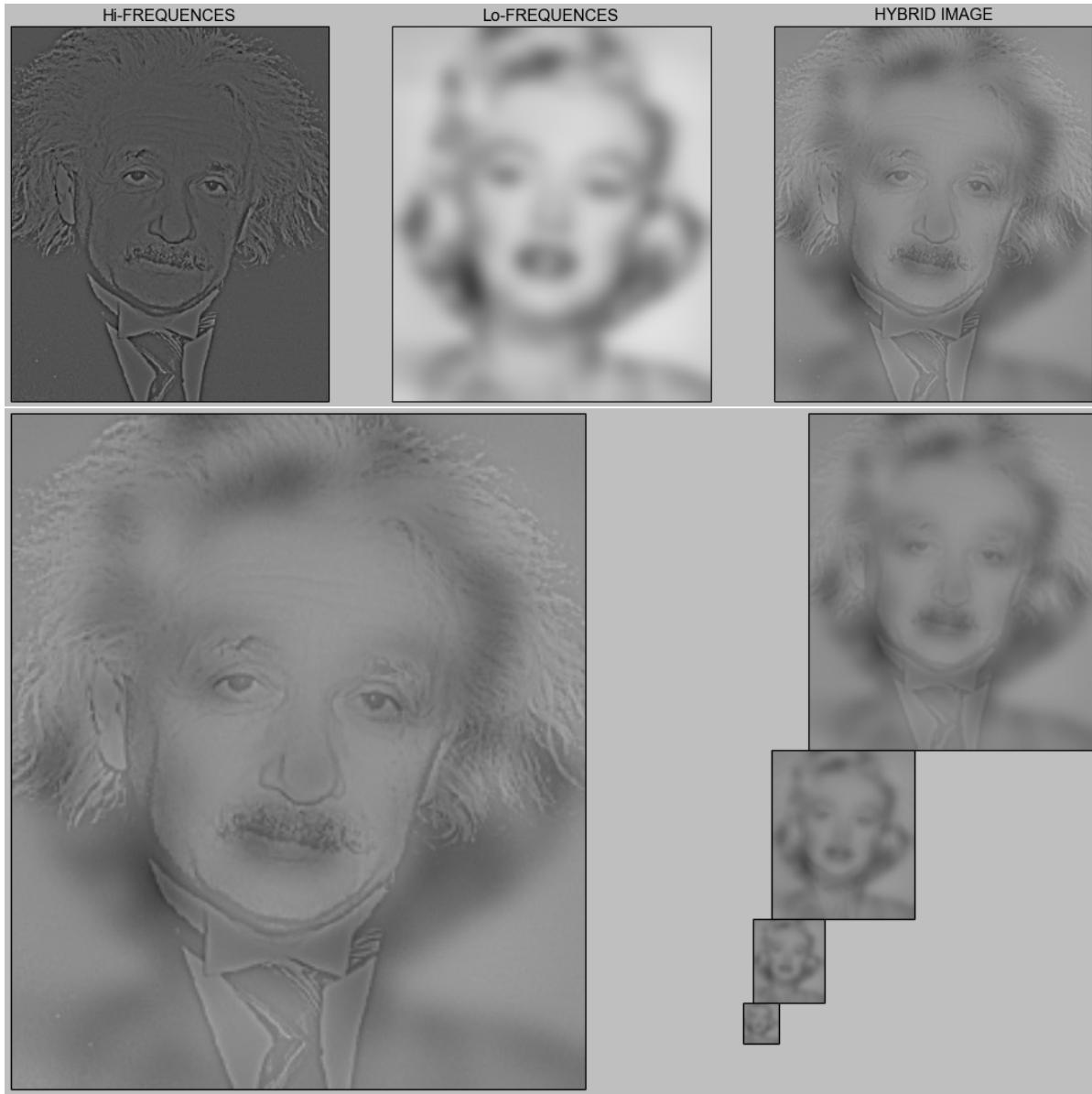
El resultado de la ejecución es el siguiente:



C) Construir una piramide 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.

En este apartado vamos a poner, para cada par de imágenes, el proceso de construcción de su imagen híbrida y su pirámide. Veremos procesos a escala de grises y a color para ver que la decisión tomada de triplicar las capas en el caso de escala de grises nos hace ganar en generalidad y que el mismo algoritmo funciona sea cual sea el tipo de color elegido.

Einstein-Marilyn

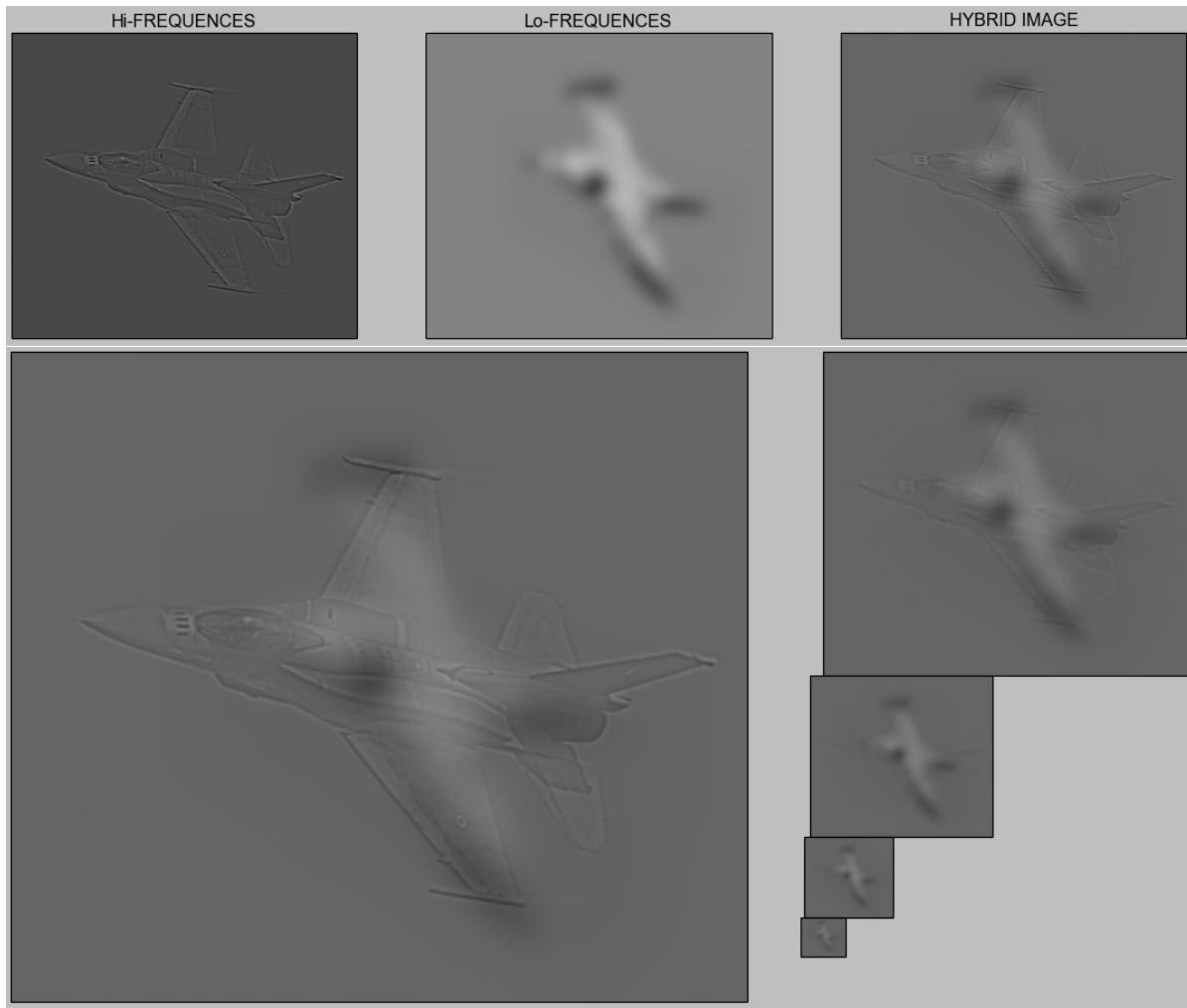


Vamos a comenzar con imágenes híbridas en escala de grises. En este caso, la calidad alcanzada es aceptable. En la escala real, no hay discusión posible y reconocemos sin duda alguna a Albert Einstein. A medida que la imagen disminuye, se intuye la silueta de Marilyn Monroe hasta que, a partir de la tercera escala se la reconoce claramente.

Los parámetros utilizados son:

- σ_{altas} : 1.8
- σ_{bajas} : 6
- Factor Laplaciano: 1 (sin cambios)
- Borde: 1 (reflejo)

Avión-Ave



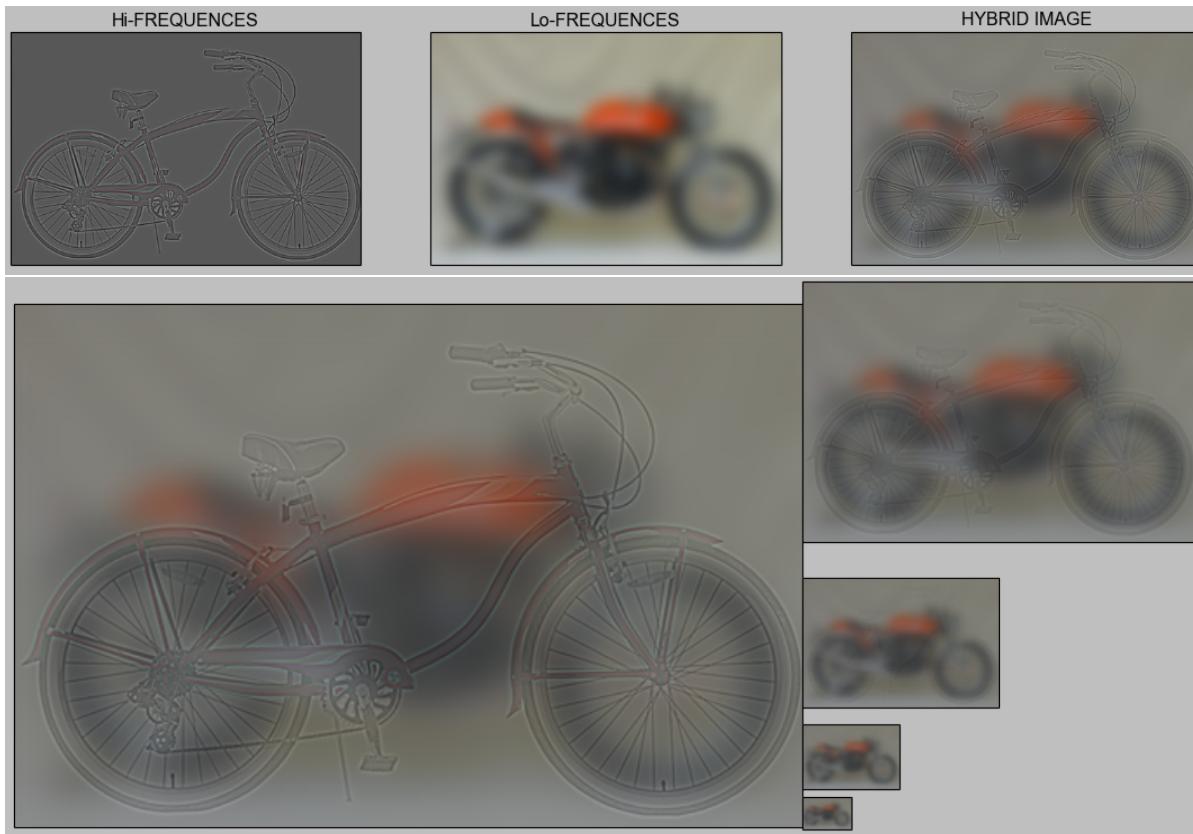
En esta segunda imagen a escala de gries nos enfrentamos a un cazabombardero superpuesto a un ave. Es difícil conseguir una imagen híbrida de calidad en este caso, pues los rasgos del ave sobre un fondo uniforme son difíciles de disolver.

En cualquier caso, se puede percibir el efecto. En la primera escala se ve el cazabombardero sin dudas, siendo difícil identificar un ave, aunque los rasgos son percibibles si se presta mucha atención. A partir de la tercera escala, el cazabombardero desaparece casi por completo y se aprecia un ave cuyo cuello y bordes de las alas son negros.

Los parámetros utilizados son:

- σ_{altas} : 1.7
- σ_{bajas} : 8
- Factor Laplaciano: 1 (sin cambios)
- Borde: 1 (reflejo)

Bici-Moto



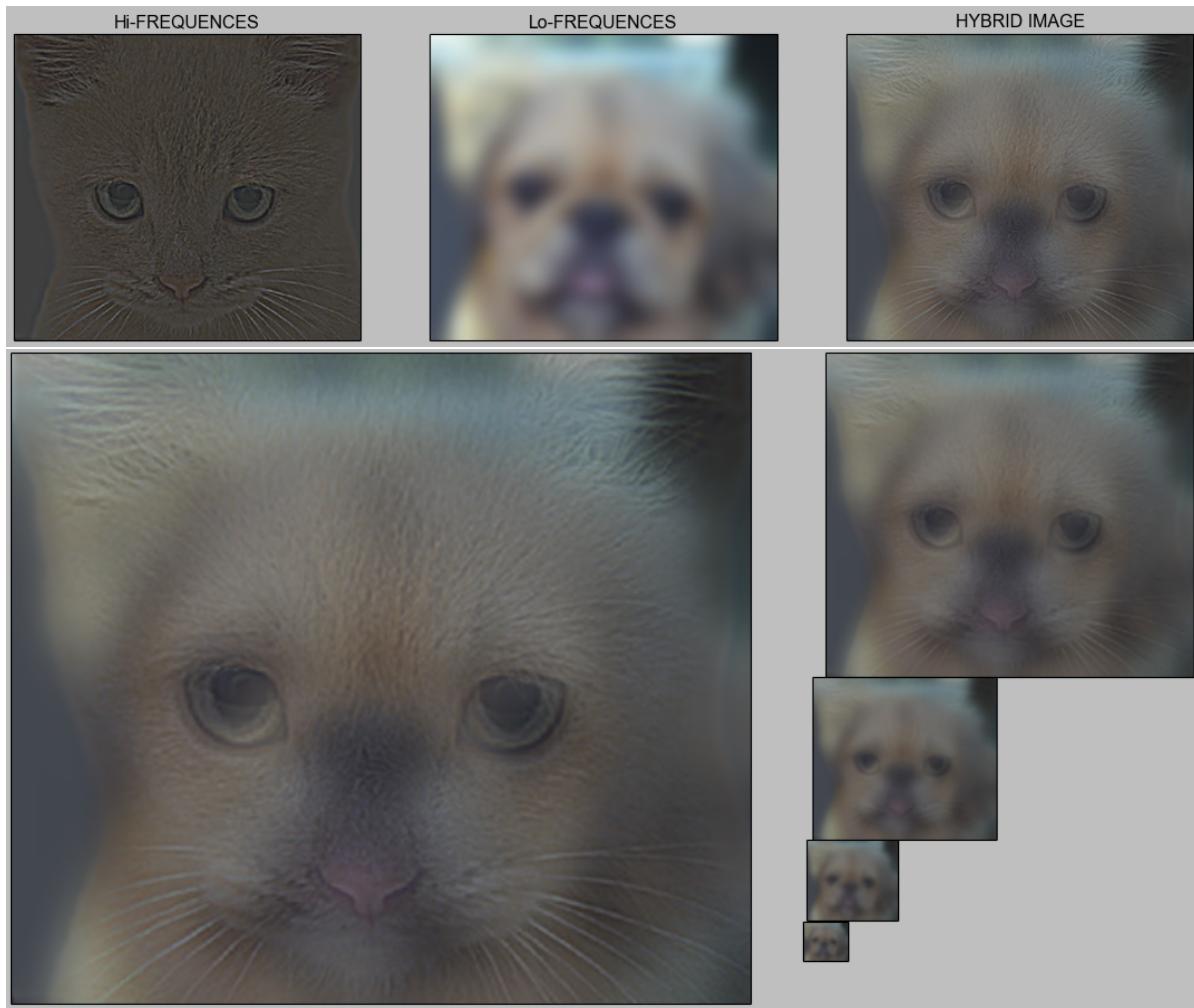
Para comprobar que los mismos algoritmos pueden ser empleados en imágenes a color que en imágenes en escala de grises, a partir de ahora, vamos a construir todas las imágenes híbridas con imágenes a color.

Las formas afiladas y bien definidas de la bicicleta permiten sacar sus detalles (altas frecuencias) sin que haya mucha diferencia entre la imagen original y la convolucionada de las que se obtienen estos, es por eso que se podrá usar un valor de σ_{altas} no muy elevado. Del mismo modo, las formas características de la moto requerirán un σ_{bajas} elevado para que no se aprecien. Hecho así, podemos comprobar cómo la bicicleta se mantiene reconocible hasta la segunda escala, siendo esta imperceptible en la tercera, donde aparece ya claramente la *Egli Vincent Sport GT*.

Los parámetros utilizados son:

- $\sigma_{altas} : 1.2$
- $\sigma_{bajas} : 10$
- Factor Laplaciano: 1 (sin cambios)
- Borde: 1 (reflejo)

Gato-Perro



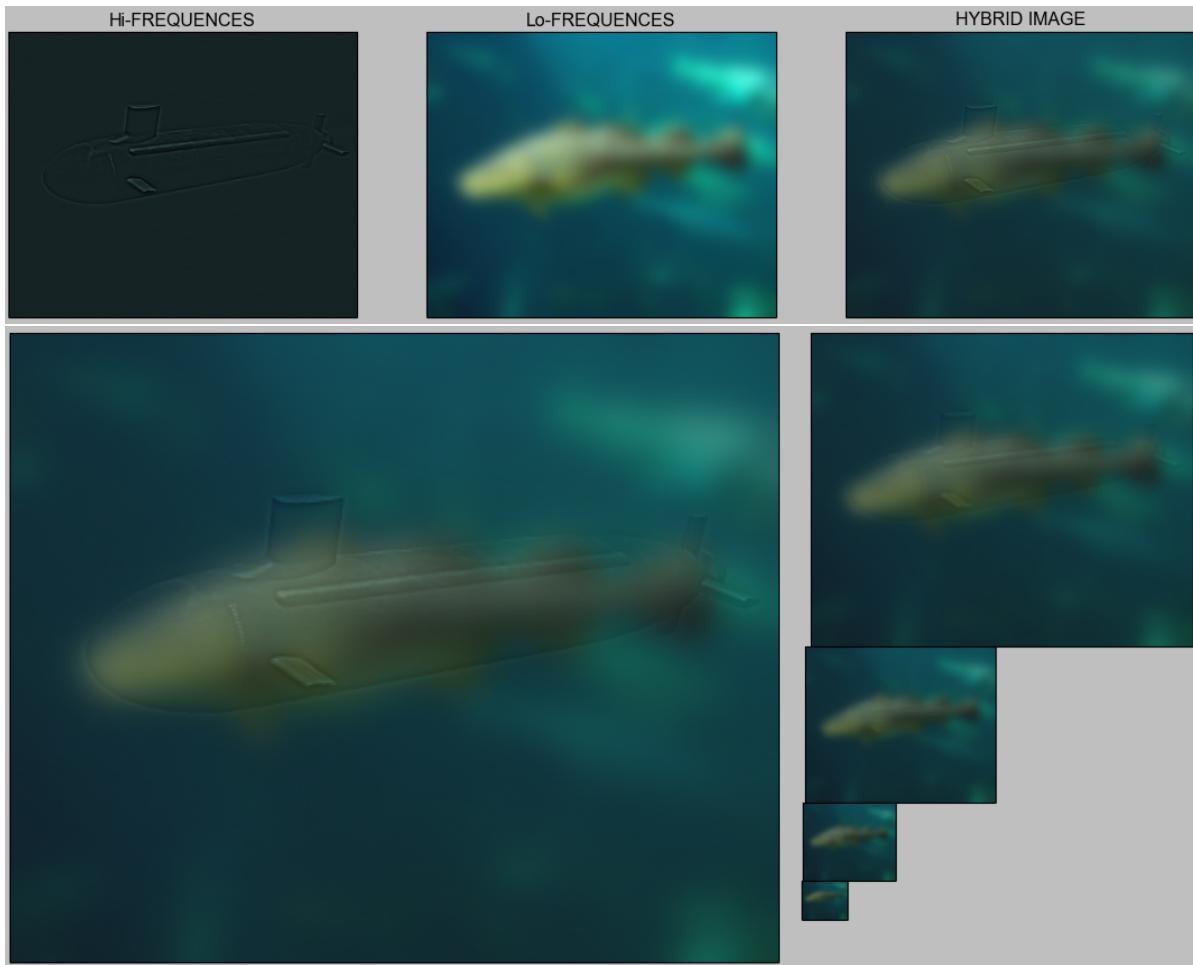
Esta imagen es complicada de interpretar. La imagen en alta frecuencia del gato devuelve demasiados detalles en un espacio muy plano, por lo que fue necesario “exagerar” los cambios con un factor Laplaciano de 1.1. Esto permite restaltar los saltos un 10% más, lo cual parece suficiente. La imagen del perro tiene muchos detalles que se pueden intuir aún a muy baja frecuencia, ojos y nariz grandes, la silueta de la cabeza, etc. Lo cual necesita un σ_{bajas} elevado para que no se perciban desde la cercanía.

La imagen construida realza al gato en las dos primeras escalas, pero a partir de la tercera ya se puede intuir el perro sin problemas, lo que permite que el efecto conseguido sea de una calidad aceptable.

Los parámetros utilizados son:

- $\sigma_{altas} : 2.5$
- $\sigma_{bajas} : 10$
- Factor Laplaciano: 1.1 (Exagera los saltos en el gato un 10%)
- Borde: 1 (reflejo)

Submarino-Pez



Por último, construimos la imagen de un submarino sobre un pez. Como ocurría en el caso de la bicicleta y la moto, los límites bien definidos del submarino permiten extraer las altas frecuencias sin problemas y la construcción de la imagen híbrida es sencilla. Como vemos, a partir de la tercera escala deja de ser perceptible el submarino y se percibe un pez, aunque con pocos detalles. De nuevo, podríamos decir que la calidad de la ilusión generada es aceptable.

Los parámetros utilizados son:

- σ_{altas} : 2
- σ_{bajas} : 8
- Factor Laplaciano: 1 (sin cambios)
- Borde: 1 (reflejo)

El código para escalar hacia abajo (achicar) la imagen, alisa primero con un determinado σ y posteriormente quita filas y columnas alternas para construir una imagen la mitad de grande que la anterior, manteniendo las proporciones. El código de esta función es el siguiente:

```
"""
La funcion scaleDownImage se encarga de escalar a mas pequena
una imagen, alisando la imagen y quitando filas y columnas
alternativamente
"""

def scaleDownImage(image,sigma,borderType):
    img=my_imGaussConvol(image,sigma,borderType)
    r,g,b=cv2.split(img)
    nrow=len(r)/2
    ncol=len(r[0])/2
    for i in xrange(0,nrow):
        r=np.delete(r,i,0) #Quitando arrays en el eje x (quitar filas)
        g=np.delete(g,i,0)
        b=np.delete(b,i,0)
    for i in xrange(0,ncol):
        r=np.delete(r,i,1) #Quitando arrays en el eje y (quitar columnas)
        g=np.delete(g,i,1)
        b=np.delete(b,i,1)
    return cv2.merge([r,g,b])
```

Por otro lado, la función que muestra la pirámide Gaussiana ha sido implementada a mano, esto es, sin utilizar funciones de OpenCV. La idea es crear una ventana, dividida en filas y columnas (las divisiones no son perceptibles). El numero de filas viene dado por el número de filas de la imagen original y el número de columnas por el número de columnas de la imagen original mas la mitad de estas, es decir, la anchura de la imagen original mas la anchura de la siguiente escala. Para dibujar, basta con indicar lo siguiente:

- La imagen original comienza en (0,0) y se extiende en su número de filas (`rowspan`) y su número de columnas (`colspan`) hacia abajo y a la derecha respectivamente.
- La segunda escala comienza en (0,colspan) y se extiende en `rowspan/2` y `colspan/2`
- La tercera escala comienza en (`rowspan/2`) y se extiende en `rowspan/4` y `colspan/4`

Así sucesivamente.

Lo primero es cerciorarse de que la imagen puede ser dividida tantas veces como indique la variable `level` para lo cual calculamos el logaritmo en base 2 del menor tamaño (numero de filas o de columnas). La imagen original ocupará todas las columnas de la parte izquierda (`ncol`) por lo que se inserará manualmente. La primera imagen de la escala será el principio de la recursividad de la columna derecha, empezará en la posición (0,ncol) y es la excepción. A partir de aquí, habrá que ir calculando en qué fila empieza la siguiente imagen de forma recursiva. El código en Python que realiza esta acción es el siguiente:

```
"""
Calcula y pinta la piramide gaussiana de una imagen a level niveles
"""

def showPyramid(imagen,windowtitle,sigma,border,level):
    nrow=len(imagen)
    ncol=len(imagen[0])

    min_size=min(nrow,ncol)

    if(level<math.log(min_size,2)):
```

```

fig = plt.figure()
fig.canvas.set_window_title(windowtitle)

ax1=plt.subplot2grid((nrow,ncol+ncol/2),
(0,0), rowspan=nrow,colspan=ncol)
plt.xticks([]),plt.yticks([])
plt.imshow(imagen)

i=0
rowini=i
row_span=nrow/2***(i+1)
col_span=ncol/2***(i+1)

img1=scaleDownImage(imagen,sigma,border)
ax2 = plt.subplot2grid((nrow,ncol+ncol/2),
(rowini,ncol), rowspan=row_span,colspan=col_span)
plt.xticks([]),plt.yticks([])
plt.imshow(img1)

for i in xrange(1,level-1):
    rowini+=row_span
    row_span=nrow/2***(i+1)
    col_span=ncol/2***(i+1)

    imgbucle=scaleDownImage(img1,sigma,border)
    plt.subplot2grid((nrow,ncol+ncol/2),
    (rowini, ncol), rowspan=row_span, colspan=col_span)
    plt.xticks([]),plt.yticks([])
    plt.imshow(imgbucle)

    img1=imgbucle
print("Esta viendo la piramide gaussiana, cierre la ventana para continuar.")
plt.show()
else:
    raise ValueError,
    "Image cannot be scaled down more than "+math.log(min_size,2)+" times."

```

Cuestiones técnicas

El proyecto ha sido implementado y probado en Ubuntu 14.04 LTS, con la versión Python 2.7.6. No se ha utilizado ninguna biblioteca no estándar de Python, excepto OpenCV, que hay que tener correctamente instalado y configurado en las variables de entorno. Para ejecutar correctamente el proyecto, hay que ejecutar el comando `python P1.py` (o `python2 P1.py` en caso de no ser la 2.7 la versión de Python por defecto) en una terminal desde el directorio donde se encuentra dicho archivo. Las imágenes que se usan deben estar dentro de una carpeta llamada `imagenes` y esta carpeta debe estar en el mismo directorio que el archivo `P1.py`. Los nombres de las imágenes deben ser los siguientes:

- bicycle.bmp
- bird.bmp
- cat.bmp
- dog.bmp
- einstein.bmp

- fish.bmp
- marilyn.bmp
- motorcycle.bmp
- plane.bmp
- submarine.bmp