Panoramas lineales con proyección en superficies cilíndricas o esféricas usando el algoritmo de Burt-Adelson.

ASIGNATURA: VISIÓN POR COMPUTADORES

GRUPO: A

ALUMNOS: ALFREDO CARRIÓN CASTEJÓN

CARLOS MORALES AGUILERA

DNI: 15521994-F

75925767-F

ENTREGA: PROYECTO FINAL

ÍNDICE

NOTA: Para la ejecución del proyecto es necesario descargar el archivo de *https://consigna.ugr.es/f/xXDQS93eLEuBjPwx/imagenes.zip*

descomprimir en la misma carpeta donde se encuentra el archivo.py.

1. Motivación del proyecto.

Como sabemos, una de las tareas a estudiar en la rama informática de la Visión por Computadores, es la de creación de mosaicos o panoramas.

Este problema consiste en la formación de imágenes (denominadas panorámicas), las cuales son formadas a partir de un conjunto de imágenes de un mismo lugar, con el objetivo de formar una única imagen, que se acopla correctamente y contiene la información de las distintas imágenes siguiendo una continuidad adecuada.

El problema surge a la hora del acople de imágenes, ya que como hemos estudiado en esta asignatura, un simple acople de imágenes con el uso de transformaciones no producirá un buen resultado en la mayoría de casos (luminosidad, distintos instantes a la hora de sacar las imágenes, objetos aparentes en una imagen pero no en otra), formando cortes innecesarios entre las imágenes, lo cual es la motivación de este proyecto.

2. **Solución propuesta**: Algoritmo de mezcla mediante pirámides Laplacianas de Burt-Adelson con el uso de proyecciones cilíndricas o esféricas en las imágenes.

Como sabemos, hasta el momento tendríamos un mosaico que se forma adecuadamente debido al uso de correspondencias con descriptores SIFT (Scale-Invariant Feature Transform), el cual hemos estudiado en la asignatura, pero una vez explicado el problema en el apartado anterior, surgen dos ideas para solucionar dicho problema mencionado anteriormente.

La primera consiste en trasladar el problema de panoramas lineales a proyecciones cilíndricas o esféricas, con el objetivo de obtener unas imágenes en una proyección distinta, suavizando los distintos cortes que obtengamos entre imágenes y facilitando un mayor acople entre imágenes.

Por otro lado, presentamos el algoritmo de Burt-Adelson, que busca principalmente suavizar las zonas comunes entre imágenes con el objetivo de no visualizar cortes entre imágenes (se describe posteriormente).

3. Esquema del proyecto.

Una vez mencionado el objetivo y solución, resumimos brevemente los pasos a realizar en el proyecto:

- Proyecciones cilíndricas y esféricas en las imágenes (4).
- Obtención de correspondencias (5).
- Composición de mosaico mediante homografías (6).
- Algoritmo de Burt-Adelson (7).

4. Proyecciones.

Como hemos comentado, el primer objetivo es trasladar nuestro problema sobre panoramas lineales a proyecciones esféricas, buscando un mejor acople de las imágenes y mejorando las correspondencias a realizar.

4.1. Cilindrícas.

Como sabemos una proyección cilíndrica consiste en situar una imagen con su información en una determinada proyección lineal, a una proyección cilíndrica, la cual posee una deformación curva y se pierde algo de información.

La transformación a realizar depende principalmente de dos elementos: distancia focal *f* la cual consiste en la distancia que hay desde el centro óptico que vamos a situar en el centro de la imagen y el factor de escalado *s* que definirá cuando se ha de escalar la imagen para situarla en la imagen original que sustituiremos.

Se propone la siguiente fórmula para las transformaciones de las componentes x e y de las coordenadas:

$$x_{cilindrica} = s * arctan\left(\frac{x - centro_x}{f}\right) + centro_x$$

$$y_{cilindrica} = s * \frac{y - centro_y}{\sqrt{(x + centro_x)^2 + f^2}} + centro_y$$

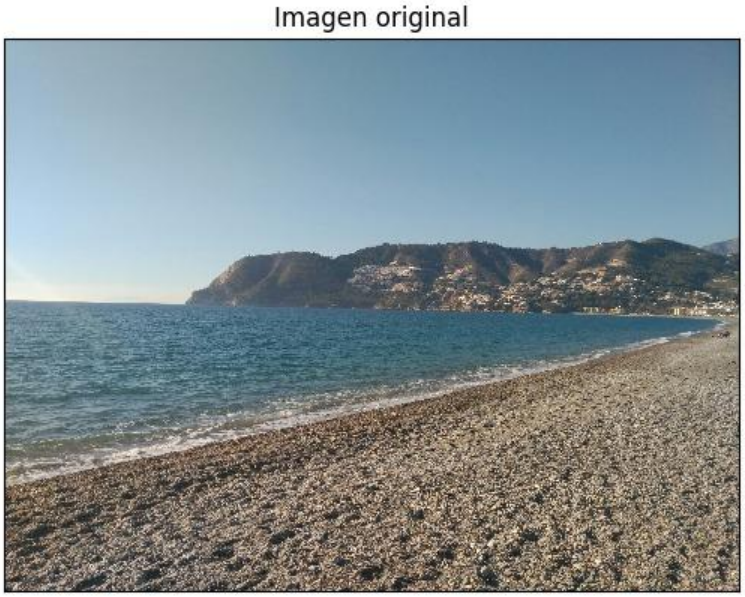
En nuestro planteamiento, como queremos conservar la mayor precisión posible en la información hemos decidido separar las imágenes en sus canales RGB (Red, Green, Blue), y aplicar dicha transformación en cada canal, uniéndolos posteriormente, evitando que en cualquier cálculo pudiéramos perder precisión o información.

Una vez explicado el procedimiento pasamos a la implementación del mismo, como se pueden observar tenemos dos casos: imagen a color e imagen en escala de grises, en el primer caso la descomponemos en canales, operamos sobre ellos como escala de grises y los mezclamos en una imagen a color con las funciones propias de OpenCV **cv2.split()** y **cv2.merge()**. En el caso contrario, como sabemos, nuestro origen en OpenCV se sitúa en la esquina superior izquierda, por lo que tendremos que trasladar las coordenadas al centro, aplicar la proyección cilíndrica y volver a trasladarlas respecto al origen en OpenCV.

```python
# Funcion que implementa una proyeccion cilindrica sobre una imagen,
# dada una distancia focal f y un factor de escalado s
def ProyeccionCilindrica(imagen, f, s):
    # Si tenemos una imagen en color, separamos en canales RGB y realizamos
    # el proceso de proyeccion canal a canal
    if len(imagen.shape) == 3:
        # Separamos en los distintos canales
        canales = cv2.split(imagen)
        canales_proyecciones = []
        for n in range(len(canales)):
            # Realizamos cada proyección cilindrica
            canales_proyecciones.append(ProyeccionCilindrica(canales[n],f,s))
        # Mezclamos los canales obteniendo las proyeccion para la imagen a color
        imagen_proyectada=cv2.merge(canales_proyecciones)

    else:
        # Creamos una nueva imagen del tamaño de la original
        imagen_proyectada = np.zeros(imagen.shape)

        # Recorremos la imagen y vamos aplicando la proyeccion, como sabemos
        # nuestras coordenadas empiezan en la esquina superior izquierda de
        # nuestra imagen
        centro_anchura = imagen.shape[1]/2
        centro_altura = imagen.shape[0]/2

        for i in range(imagen.shape[0]):
            for j in range(imagen.shape[1]):
                # Sacamos los indices x' e y'
                i_proyectada = floor(s*((i-centro_altura)/np.sqrt((j-centro_anchura)*(j-centro_anchura)+f*f)) + centro_altura)
                j_proyectada = floor(s*np.arctan((j-centro_anchura)/f) + centro_anchura)
                imagen_proyectada[i_proyectada][j_proyectada] = imagen[i][j]
        # Normalizamos los datos al tipo uint8
        imagen_proyectada = cv2.normalize(imagen_proyectada,imagen_proyectada,0,255, cv2.NORM_MINMAX,cv2.CV_8U)

    return imagen_proyectada
```

Para comprobar el perfecto funcionamiento de nuestra proyección cilindrica mostraremos varias ejecuciones de nuestro algoritmo.

- Imgen Original:



- Imagen con proyección cilíndrica con factor de escala y distancia focal de 300:

- Imagen con proyección cilíndrica con factor de escala y distancia focal de 500:

- Imagen con proyección cilíndrica con factor de escala y distancia focal de 700:

- Imagen con proyección cilíndrica con factor de escala y distancia focal de 900:

Como podemos observar cuanto mayor es la distancia focal y el factor de escala, la imagen estará menos proyectada cilíndricamente, es decir, más se parecerá a nuestra imagen original.

4.2. Esféricas.

Para la proyección esférica la idea es la misma, solo que empleando su fórmula correspondiente, el procedimiento e implementación es idéntico, únicamente variaremos el cómputo de la misma debido a que presenta una transformación distinta:

$$x_{esferica} = s * arctan\left(\frac{x - centro_x}{f}\right) + centro_x$$

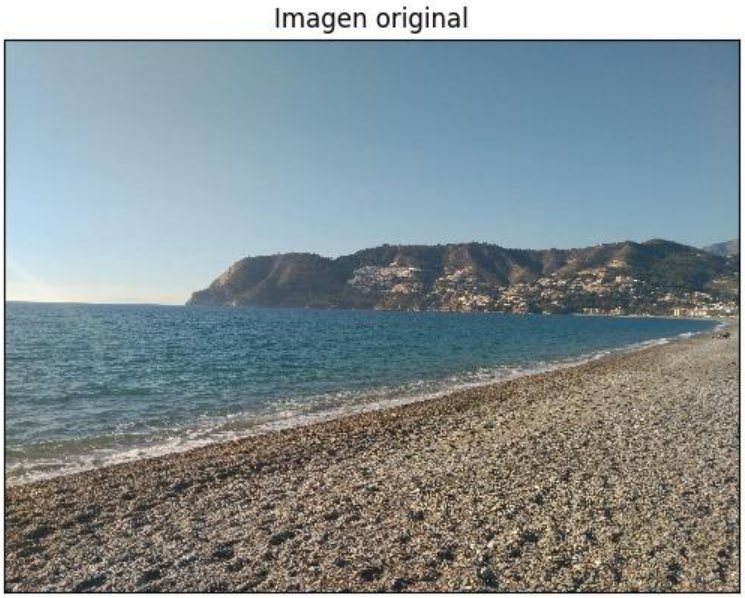
$$y_{esferica} = s * arctan\left(\frac{y - centro_y}{\sqrt{(x + centro_x)^2 + f^2}}\right) + centro_y$$

La implementación como podemos comprobar, sigue el mismo procedimiento que la proyección cilíndrica, sustituyendo donde corresponde por la proyección esférica:

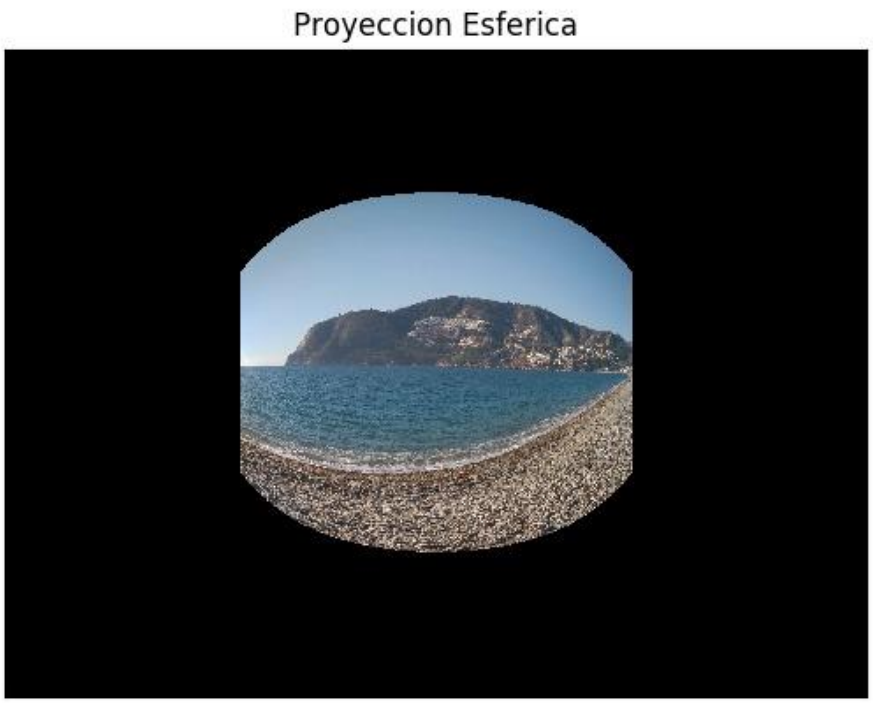
```python
# Funcion que implementa una proyeccion cilindrica sobre una imagen,
# dada una distancia focal f y un factor de escalado s
def ProyeccionEsferica(imagen, f, s):
    # Si tenemos una imagen en color, separamos en canales RGB y realizamos
    # el proceso de proyeccion canal a canal
    if len(imagen.shape) == 3:
        # Separamos en los distintos canales
        canales = cv2.split(imagen)
        canales_proyecciones = []
        for n in range(len(canales)):
            # Realizamos cada proyección cilindrica
            canales_proyecciones.append(ProyeccionEsferica(canales[n],f,s))
        # Mezclamos los canales obteniendo las proyeccion para la imagen a color
        imagen_proyectada=cv2.merge(canales_proyecciones)

    else:
        # Creamos una nueva imagen del tamaño de la original
        imagen_proyectada = np.zeros(imagen.shape)

        # Recorremos la imagen y vamos aplicando la proyeccion, como sabemos
        # nuestras coordenadas empiezan en la esquina superior izquierda de
        # nuestra imagen
        centro_anchura = imagen.shape[1]/2
        centro_altura = imagen.shape[0]/2

        for i in range(imagen.shape[0]):
            for j in range(imagen.shape[1]):
                # Sacamos los indices x' e y'
                i_proyectada = floor(s*np.arctan((i-centro_altura)/np.sqrt((j-centro_anchura)*(j-centro_anchura)+f*f)) + centro_al
                j_proyectada = floor(s*np.arctan((j-centro_anchura)/f) + centro_anchura)
                imagen_proyectada[i_proyectada][j_proyectada] = imagen[i][j]
        # Normalizamos los datos al tipo uint8
        imagen_proyectada = cv2.normalize(imagen_proyectada,imagen_proyectada,0,255, cv2.NORM_MINMAX,cv2.CV_8U)

    return imagen_proyectada
```

Para comprobar el perfecto funcionamiento de nuestra proyección esférica mostraremos varias ejecuciones de nuestro algoritmo.

- Imagen Original:



- Imagen con proyección esférica con factor de escala y distancia focal de 300:



- Imagen con proyección esférica con factor de escala y distancia focal de 500:

- Imagen con proyección esférica con factor de escala y distancia focal de 700:

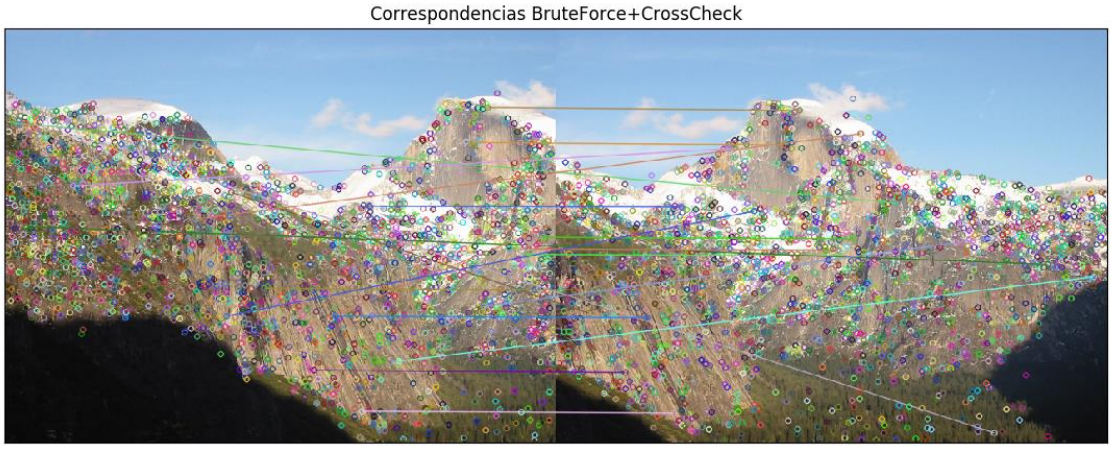
- Imagen con proyección esférica con factor de escala y distancia focal de 900:

Como podemos observar cuanto mayor es la distancia focal y el factor de escala, la imagen estará menos proyectada esféricamente, es decir, más se parecerá a nuestra imagen original.

5. Obtención de correspondencias.

Como ya hemos visto a lo largo de esta asignatura, uno de los temas importantes de la Visión por computador es la del uso de descriptores de imagen los cuales aportan información relevante de la misma, para este apartado hemos utilizado los descriptores *SIFT* (Scale-Invariant Feature Transform) vistos en las prácticas de la asignatura.

El objetivo ha sido el mismo que el de la práctica 2 de la asignatura, componer un mosaico, y para realizar las homografías correspondientes es necesario el uso de correspondencias entre puntos de dos imágenes a fusionar en un mosaico. Como ya sabemos SIFT posee información de posición y orientación de los descriptores de la imagen, y además obtenemos los keypoints o "puntos clave" los cuales ofrecen puntos con alta relevancia en la imagen. Con este conjunto de información hemos utilizado un objeto *BruteForce* para realizar las correspondencias de descriptores entre ambas imágenes, aplicando también la técnica de *CrossCheck* o validación cruzada, consistente en que asegura que una correspondencia existe sí y solo sí ambos descriptores son la mejor correspondencia del otro (mutualidad). Con esto obtendremos las correspondencias necesarias para poder realizar las homografías y conformar nuestro mosaico.



La implementación empleada para este propósito es la misma utilizada en la práctica 2 de dicha asignatura (del alumno Carlos).

```python
def correspondencias(imagen1, imagen2, criterio, elementos):
    # Creamos el detector-descriptor SIFT
    sift = cv2.xfeatures2d.SIFT_create()

    # Obtenemos los keypoints y los descriptores de las dos imagenes
    keypoints1, descriptores1 = sift.detectAndCompute(imagen1, None)
    keypoints2, descriptores2 = sift.detectAndCompute(imagen2, None)

    # Si hemos escogido el criterio 1, utilizamos BruteForce+CrossCheck
    if criterio == 1:
        # Creamos el objeto BFMatcher con CrossCheck
        bf = cv2.BFMatcher(normType=cv2.NORM_L2, crossCheck=True)

        # Calculamos las correspondencias con los descriptores de las imagenes
        correspondencias = bf.match(descriptores1, descriptores2)

        # De todas las correspondencias obtenidas, escogemos n elementos
        # aleatorios para mostrarlos
        elem_correspondencias = sample(range(len(correspondencias)), elementos)

        seleccionados = []
        for i in range(len(elem_correspondencias)):
            seleccionados.append(correspondencias[i])

        # Mostramos los n elementos aleatorios
        imagen_final = cv2.drawMatches(img1=imagen1, keypoints1=keypoints1,
                                       img2=imagen2, keypoints2=keypoints2,
                                       matches1to2=seleccionados, outImg=None)
```

6. Composición del mosaico a partir de homografías.

Una vez obtenidas las correspondencias, el siguiente paso sería conformar un mosaico, el cual se realizaría con sus homografías a partir de las correspondencias y dependiendo de donde queramos situar una imagen en el canvas creado. En nuestro proyecto, sin embargo, no colocaremos las imágenes como tal con sus correspondientes homografías, es justo el problema que vamos a intentar corregir con el algoritmo de Burt-Adelson. Definiremos las homografías, colocaremos las imágenes y aplicaremos Burt-Adelson. El resultado de Burt-Adelson es una imagen que mezcla dos imágenes combinadas correctamente y con suavizado entre ellas, como ya anteriormente hemos aplicado las homografías, se colocaran correctamente en su sitio.

La implementación es la siguiente:

```python
def mosaico_nBA(lista_imagenes):
    # Sacamos el centro de la imagen
    centro = len(lista_imagenes)//2

    # Empezamos la parte derecha desde el centro
    mosaico_der = mosaicoBA(lista_imagenes[centro], lista_imagenes[centro+1])
    # Empezamos la parte izquierda desde el centro
    mosaico_izq = mosaicoBA(lista_imagenes[centro-1], lista_imagenes[centro])

    # Realizamos el mosaico iterativamente hacia la derecha
    for n in range(centro, len(lista_imagenes)):
        mosaico_der = mosaicoBA(mosaico_der, lista_imagenes[n])
    # Realizamos el mosaico iterativamente hacia la izquierda
    for n in range(centro, 0, -1):
        mosaico_izq = mosaicoBA(mosaico_izq, lista_imagenes[n])

    # Unimos ambas partes en un único mosaico
    mosaico_final = mosaicoBA(mosaico_izq, mosaico_der)

    return mosaico_final
```

Como podemos observar simplemente vamos construyendo un mosaico desde el centro a los lados para finalmente unir los mosaicos obtenidos en un mosaico final que contendrá a ambos. Procedemos a ver la formación de un mosaico a partir de únicamente dos imágenes:

```python
def mosaicoBA(imagen1, imagen2):
    # Calculamos la homografía que sítua a la imagen 2 en el centro del canvas
    homografia=np.matrix([[1,0,0],[0,1,0],[0,0,1]],
                         dtype=float)

    # Calculamos las correspondencias y keypoints de la imagen 2 (derecha) a la imagen 1 (izquierda)
    correspondencias_x, keypoints1, keypoints2 = correspondencias(imagen1, imagen2, 1, 20)[0:3]

    # Obtenemos los puntos de fuente y destino de los objetos DMatch y con un reshape
    # tal y como se muestra en https://docs.opencv.org/3.3.1/d1/de0/tutorial_py_feature_homography.html
    puntos_dst = np.float32([keypoints1[m.queryIdx].pt for m in correspondencias_x]).reshape(-1, 1, 2)
    puntos_src = np.float32([keypoints2[m.trainIdx].pt for m in correspondencias_x]).reshape(-1, 1, 2)

    # Calculamos la homografía de la imagen 2 a la imagen 1
    homografia_x = cv2.findHomography(puntos_src, puntos_dst, cv2.RANSAC, 1)[0]

    # Tamaño del canvas
    size = (imagen1.shape[1] + imagen2.shape[1], imagen1.shape[0])

    # Aplicamos la homografia que situa a la imagen 2 en el centro del canvas
    area1 = cv2.warpPerspective(imagen1, homografia, size)
    # Aplicamos la homografia que situa a la imagen 1 con respecto a la imagen 2
    area2 = cv2.warpPerspective(imagen2, homografia*homografia_x, dsize=size)

    area1, area2 = limpiarImagen(area1, area2)

    # Aplicamos Burt-Adelson y obtenemos un mosaico
    mosaico = BurtAdelson(area1,area2)

    return mosaico
```

Realizamos una homografía simple que nos coloque en el canvas formado la primera imagen, y, a continuación, en otro canvas del mismo tamaño, la otra imagen con la homografía adecuada, con esto obtenemos dos imágenes con sus correspondientes homografías por separadas, que son las que combinaremos con Burt-Adelson.

7. Algoritmo de Burt-Adelson.

Para comenzar vemos una breve descripción antes de entrar en detalle:

Este algoritmo propone una solución al problema de mezclar imágenes, consistente en el uso de combinar pirámides laplacianas y restauración de la misma. El objetivo es obtener una mezcla suave y difuminada de dos imágenes sin que se note el cambio de contraste en las mismas.

Como se explica en http://www.cs.princeton.edu/courses/archive/fall05/cos429/papers/burt_adelson.pdf, consiste en un algoritmo que escoge una franja común entre imágenes, y la mezcla con una convolución con el objetivo de tener un difuminado suave entre las mismas imágenes. El problema ahora viene al obtener nuevamente malos resultados debido a que se producirá una línea borrosa que en casos muy aislados servirá como solución, debido a que no evitamos los cortes entre imágenes que queríamos eliminar, sino que únicamente los difuminamos.

Es por este motivo por lo que algoritmo de Burt-Adelson propone realizar dicha operación en diferentes bandas de frecuencias con el uso de pirámides Laplacianas. Esto se debe a que al realizar dicha mezcla nivel por nivel, y posteriormente reconstruir las imágenes, obtenemos una mezcla suave entre dichas imágenes.

Procedemos a continuación a la explicación del algoritmo:

- Construcción de las pirámides Laplacianas de las imágenes a mezclar.
  Para la construcción de las pirámides Laplacianas, hemos utilizado primero el mismo algoritmo propuesto en la primera práctica de la asignatura para conformar la pirámide Gaussiana, utilizando para ello la función de OpenCV **cv2.pyrDown()**.

```
def PiramideGaussiana(imagen, niveles=8):
    # Guardamos la imagen original
    piramide = [imagen]
    actual = imagen
    # Recorremos las imágenes
    for i in range(niveles):
        # Establecemos el formato adecuado para OpenCV
        actual = np.uint8(actual)
        # Realizamos la convolucion y submuestreo
        actual = cv2.pyrDown(actual)
        # Recuperamos el formato
        actual = np.uint32(actual)
        # Guardamos la imagen
        piramide.append(actual)
    return piramide
```

Sin embargo, a la hora de realizar el procedimiento de la pirámide Laplaciana nos hemos encontrado con un problema con la función de OpenCV **cv2.pyrUp()** empleado en la primera práctica de la asignatura (se perdía más información de la deseada a la hora de realizar la pirámide Gaussiana, provocando demasiada luminosidad y deformación), por lo que hemos optado a realizar un sobremuestreo manual y aplicar una convolución con nuestra función de convolución de núcleo separable, que dado un kernel, aplica con dicho kernel una convolución en una dimensión con la función propia de OpenCV **cv2.filter2D()** (función de convolución de la primera práctica editada para el sobremuestreo).

```
def convolucionSeparable(img, tam, kernel_gaussiano):
    # Aumentamos el tamaño de la imagen al deseado
    im_expanded = np.zeros(tam.shape, img.dtype)
    im_expanded[::2, ::2, ...] = img

    # Generamos el kernel a utilizar
    if kernel_gaussiano:
        mascara = Gaussiana(1)
    else:
        mascara = 1.0 / 10 * np.array([1, 5, 8, 5, 1])

    copia = im_expanded # Realizo una copia de la imagen sobre la que trabajare

    for c in range(2): # Una iteracion para filas, otra para columnas
        aux = copia.copy() # Realizo una copia auxiliar
        for i in range(copia.shape[0]): # Recorro las filas
            # Aplico el filtro en las filas de la imagen, almacenandolo en
            # la imagen auxiliar, con la mascara calculada anteriormente
            # y el borde indicado
            aux[i,:] = cv2.filter2D(src=copia[i,:],dst=aux[i,:],ddepth=cv2.CV_32F,kernel=mascara,borderType=cv2
        # Realizo la traspuesta para poder actuar en filas y columnas como si
        # ambas fueran filas
        copia = cv2.transpose(aux)

    return copia
```

Donde Gaussiana es la función del bonus 1 de la primera práctica de la asignatura, el cual consiste en la creación de un kernel Gaussiano a partir de un determinado sigma, hemos probado con distintos kernels y hemos podido comprobar que con un kernel arbitrario [0.1, 0.5, 0.8, 0.5, 0.1] obtenemos resultados más suavizados, por lo que hemos deducido que el tipo de kernel influye enormemente, y una mayor diferencia en el kernel produce un mejor difuminado, aunque quepa pensar que un kernel gaussiano ofrezca mejor suavizado.

```
def Mascara_Gaussiana(x,sigma):
    # Mascara gaussiana
    return exp(-((x*x)/(2*(sigma*sigma))))

def Gaussiana(sigma):

    # Definimos un array de valores desde -3sigma hasta 3sigma+1
    mascara = np.arange(-floor(3*sigma), floor(3*sigma+1))

    # Calculamos la funcion para cada valor del vector
    vector = []
    for i in mascara:
        vector.append([Mascara_Gaussiana(i,sigma)])

    # Lo transformamos en un array
    mascara = np.array(vector)

    # Dividimos cada elemento de la mascara por la media de todos ellos
    mascara = np.divide(mascara, np.sum(mascara))
    return mascara
```

A continuación mostramos el código empleado para la realización de la pirámide gaussiana:

```
def PiramideLaplaciana(gaussiana):
    piramide = []
    for i in range(len(gaussiana) - 1):
        # Cogemos los operandos
        actual = gaussiana[i]
        siguiente = gaussiana[i + 1]
        # Transformamos al formato adecuado para el upsample
        siguiente = np.float32(siguiente)
        # Realizamos la convolucion y aumento
        aumento = convolucionSeparable(siguiente, actual)
        # Recuperamos el formato
        siguiente = np.uint32(siguiente)
        # Realizamos la diferencia para obtener el nivel de la laplaciana
        laplacian = actual - aumento
        piramide.append(laplacian)

    # Guardamos el ultimo nivel de la gaussiana que es el primer nivel
    # de la laplaciana
    piramide.append(gaussiana[-1])
    return piramide
```

- Mezcla de las pirámides en una nueva pirámide Laplaciana (utilizando una máscara). Una vez construidas ambas pirámides a mezclar, procedemos a la combinación de las mismas, tras probar con distintas máscaras a aplicar, con la que hemos obtenido mejores resultados dada nuestra implementación, es con la máscara que establece la mitad de cada imagen en la Laplaciana a cada una de las imágenes y promedia aquellos píxeles que se encuentran en la franja común de ambas imágenes.

Este proceso podría realizarse con una máscara personalizada por cada par de imágenes, en nuestro caso, hemos decidido tomar como zona común toda aquella zona que compartan ambas imágenes.

```
def Mezcla(Laplaciana1, Laplaciana2):
    Laplaciana_final = []
    for i in range(len(Laplaciana1)):
        #left = Laplaciana1[i]
        #right = Laplaciana2[i]
        nivel = np.zeros(Laplaciana1[i].shape, Laplaciana1[i].dtype)
        mitad = Laplaciana1[i].shape[1] // 2
        nivel[:, :mitad, ...] = Laplaciana1[i][:, :mitad, ...]
        nivel[:, -mitad:, ...] = Laplaciana2[i][:, -mitad:, ...]
        if Laplaciana1[i].shape[1] % 2 == 1:
            # Numero impar de columnas
            nivel[:, mitad, ...] = (Laplaciana1[i][:, mitad, ...] + Laplaciana2[i][:, mitad, ...])/2

        Laplaciana_final.append(nivel)
    return Laplaciana_final
```

- Restauración de la imagen mezclada a partir de la pirámide Laplaciana.
  Como sabemos, la construcción de una pirámide Laplaciana consiste en la extracción de frecuencias altas de una imagen a partir de sobremuestreos y convoluciones que parten del último nivel de la pirámide Gaussiana. Por lo tanto, el procedimiento a seguir es precisamente el contrario, la adición de los distintos niveles de la pirámide Laplaciana, realizando sobremuestreos añadiéndolos al siguiente nivel, recuperando así la imagen, para lo cual se propone la siguiente implementación:

```
def RestaurarLaplaciana(piramide):
    # Cogemos el ultimo nivel de la laplaciana
    recuperacion = piramide[-1]
    # Recorremos todos los niveles
    for i in range(len(piramide) - 1):
        # Cogemos el siguiente
        siguiente = piramide[-2 - i]
        # Transformamos al formato adecuado para el upsample
        recuperacion = np.float32(recuperacion)
        # Realizamos la convolucion y aumento
        aumento = convolucionSeparable(recuperacion, siguiente)
        # Recuperamos el formato
        recuperacion = np.uint32(recuperacion)
        # Realizamos la suma y recuperamos la imagen
        recuperacion = aumento + siguiente
    # Guardamos la imagen en el formato uint32
    recuperacion = np.uint32(recuperacion)
    return recuperacion
```

8. Resultados obtenidos y valoraciones de los mismos.

Como primer ejemplo hemos decidido tomar la cordillera de Yosemite, aplicando para ello 7 niveles de pirámide Laplaciana.

Como podemos observar en el resultado, con las proyecciones bien acopladas el resultado es realmente bueno, no vemos ningún tipo de corte o transición entre las imágenes, la imagen parece tomada de una única fotografía, lo cual era el objetivo del proyecto.

- Como segundo ejemplo hemos tomado una fotografía realizada por nosotros de una playa, para demostrar la diferencia entre aplicar más o menos niveles en la pirámide Laplaciana.

2 niveles de Laplaciana.

4 niveles de Laplaciana.

7 niveles de Laplaciana.

Una vez visualizadas con 3 números diferentes de niveles podemos observar que el número de niveles de la pirámide Laplaciana influye enormemente en el resultado, ya que un mayor número de niveles en la pirámide realizará como es lógico un mayor suavizado, por lo que el resultado se verá más o menos suavizados, dando una sensación más natural o no.

9. Conclusiones.

Dentro de que hemos obtenido resultados más o menos satisfactorios, podemos sacar una seria de ventajas y desventajas como conclusiones las cuáles enumeraremos:

- **Ventajas.**
  - Resultados buenos, podemos observar que con una buena elección de las proyecciones, máscara y niveles de la Laplaciana se pueden obtener resultados que se integran excelentemente, pese a la simple complejidad del algoritmo.
  - Computacionalmente, no es un algoritmo pesado, el cómputo dependerá del tamaño y calidad de las imágenes como es lógico, podemos comprobar que en el ejemplo de Yosemite al ejecutar, es bastante rápido, aunque en el ejemplo de la playa, al tomar mayor calidad en las fotografías, puede llegar a ser un proceso tedioso. Sin embargo, el algoritmo es de cómputo sencillo.
  - Personalización, con esto nos referimos al hecho de poder establecer una determinada máscara u otra, ya que cogiendo dos imágenes, Burt-Adelson permite fusionar dos imágenes dadas esas máscaras escogiendo la parte interesada en cada parte.
- **Desventajas.**
  - Elección de parámetros compleja, como podemos ver, un insuficiente nivel de Laplacianas produce deformaciones en la imagen, es cierto que perdemos las líneas que separaban, pero en lugar de ello las cambiamos por cortes de

contraste, lo cual no se puede considerar una mejora. Frente a ello nos encontramos con un exceso de niveles, lo cual difumina la imagen y perdemos información.

- Elección de proyección, como se puede observar en el ejemplo, es complejo escoger una que se adecue a nuestras necesidades, un factor muy alto no producirá proyección alguna, y una muy bajo deformará tanto la imagen que será difícil unirlas para crear un mosaico, además de formar bordes negros en mitad de la imagen.

10. Posibles mejoras.

El algoritmo propuesto para el proyecto es útil, pero con la inclusión de algunas mejoras, como una máscara que en términos generales se adecuara mejor al problema mejoraría bastante, además de la inclusión de algún algoritmo que adecuara el nivel de Laplaciana a cada imagen, para mejorar el resultado.

Por otro lado, es realmente relevante el orden de las imágenes, por lo que algún algoritmo que identificara donde se sitúa cada imagen sería interesante, quizás a partir de detección de regiones o características.