Trabajo 2

Detección de puntos relevantes y Construcción de panoramas

21 de Noviembre de 2017

Gema Correa Fernández 75572158T

Visión por Computador (2017-2018)
Grado en Ingeniería Informática
Universidad de Granada

Ejercicio 1

Detección de puntos Harris multiescala. Por cada región detectada necesitaremos guardar la siguiente información de cada punto: (coordenada x, coordenada y, escala, orientación). Usar para ello un vector de estructuras KeyPoint de OpenCV. Presentar los resultados con las imágenes Yosemite.rar.

Apartado a

Escribir una función que extraiga la lista potencial de puntos Harris a distintas escalas de una imagen de nivel de gris. Para ello construiremos una Pirámide Gaussiana usando escalas definidas por sigma= 1,2,3,4,5. Sobre cada nivel de la pirámide usar la función OpenCV cornerEigenValsAndVecs para extraer la información de autovalores y autovectores de la matriz Harris en cada píxel (fijar valores de blockSize y ksize equivalentes al uso de máscaras gaussianas de $sigma_I$=1.5 y $sigma_D$=1 respectivamente). Usar uno de los criterios de selección estudiados a partir de los autovalores y crear una matriz con el valor del criterio selección asociado a cada píxel (para el criterio Harris usar k=0.04). Implementar la fase de supresión de valores no-máximos sobre dicha matriz (ver descripción al final). Ordenar de mayor-a-menor (ver sortIdx()) los puntos resultantes de acuerdo a su valor. Seleccionar al menos los 500 puntos de mayor valor. Mostrar el resultado dibujando sobre la imagen original un círculo centrado en cada punto y de radio proporcional al valor del sigma usado para su detección (ver circle()).

***Ayuda para la Supresión de No-máximos*: Esta fase del algoritmo elimina como candidatos aquellos píxeles que teniendo un valor alto de criterio Harris no son máximos locales de su entorno para un tamaño de entorno prefijado (parámetro de entrada). En consecuencia solo estamos interesados en píxeles cuyo valor Harris represente un máximo local. La selección máximos locales puede implementarse de distintas maneras, pero una que es razonablemente eficiente es la siguiente: a) escribir una función que tomando como entrada los valores de una ventana nos diga si el valor del centro es máximo local; b) escribir una función que sobre una imagen binaria inicializada a 255, sea capaz de modificar a 0 todos los píxeles de un rectángulo dado; c) Fijar un tamaño de entorno/ventana y recorrer la matriz binaria ya creada preguntando, en cada posición de valor 255, si el valor del pixel correspondiente de la matriz Harris es máximo local o no; d) en caso negativo no hacer nada; e) en caso afirmativo, poner a cero en la imagen binaria todos los píxeles de la ventana y copiar las coordenadas del punto central a la lista de salida junto con su escala (nivel de la pirámide o tamaño equivalente de ventana).**

Antes de proceder a realizar la Pirámide Gaussiana, debemos comprobar que la imagen es partible entre las divisiones que vayamos realizar. Para eso, debemos ampliar la imagen de cada escala rellenando con ceros las

columnas y filas necesarias. Esto será así para que cuando realicemos una división no perdamos información o desperdiciemos píxeles y así podamos recuperar la escala de un punto relevante. Si estamos usando una imagen cuyas dimensiones son pares, no hará falta realizar este proceso. En este caso, las imágenes a usar no se verán afectadas por dicha transformación.

Una vez hecho esto, ya podemos crear nuestra Pirámide Gaussiana haciendo uso de la función **`piramide_gaussiana(imagen, levels, borde)`** de nuestra *práctica 1*, la cual nos crea una pirámide con un número determinado de niveles a partir de una imagen, también le podemos añadir un borde. En este caso, usaremos la imagen de Yosemite y nos crearemos 6 niveles, donde el primer nivel corresponde a la escala original y los 5 restantes a las sucesivas escalas. Hecha nuestra pirámide, ya podemos proceder a calcular los puntos Harris de las distintas escalas. Para ello, hacemos uso de varias funciones que iremos explicando.

`obtenerPuntosHarris`: con esta función se obtienen los puntos Harris de una imagen. Recibiremos por entrada:

- Un *vector de imágenes* con imagen escalada de la Pirámide Gaussiana. Siendo la imagen original más las 5 imágenes reescaladas, que serán calculadas con el reescalado a mitad de la anterior imagen.
- El tamaño del bloque *block_size*, que será el tamaño del vencindario del punto.
- El *k_size*, que será el sigma de la derivada.
- Y un *umbral*, variable basada en un criterio con el que restringiremos los puntos Harris, para así quedarnos con los más relevantes.

Para cada nivel o escala de la Pirámide Gaussiana calculamos los valores propios de cada píxel. Usando para ello la función `cornerEigenValsAndVecs` [1] de OpenCV la cual devuelve una matriz del tamaño de la imagen de entrada con 6 canales (λ1, λ2, x1, y1, x2, y2) donde:

- λ1, λ2 son los autovalores no ordenados de la matriz M.
- x1, y1 son los autovectores de λ1.
- x2, y2 son los autovectores de λ2.

De todos estos valores, solo nos interesan los dos primeros, es decir, los autovalores de la matriz, ya que en el primer canal se encuentra el valor propio de cada posición asociado al píxel de la imagen de entrada y en el segundo canal se encuentra el segundo valor propio respecto a dicha posición. Una vez obtenidos los autovalores, calculamos los valores del criterio de selección de Harris [2] escogido y a partir de ellos nos creamos una matriz con el valor del criterio asociado. Para eso hemos usamos el *operador de Harris*:

$$f = determinant(M) - k \cdot trace(M)^2 = (\lambda_1 \lambda_2) - k \cdot (\lambda_1 + \lambda_2)^2$$

Siendo k = 0.04. Por tanto, solo necesitamos hacer esa operación, es decir, multiplicar los dos primeros canales y restarle la suma de los dos primeros canales al cuadrado multiplicados por 0.04. Con ello se obtiene una matriz con los valores de los puntos Harris. En la función creada devolveremos esa

matriz y la matriz binaria que se nos dice en el guión de prácticas, la cual tendrá todos los valores a cero, excepto los puntos Harris que superen un determinado umbral.

```
def obtenerPuntosHarris(lista_images, block_size, k_size, umbral):

    # Resultado de aplicar el criterio de selección por cada pixel
    matriz_criterio_seleccion = []

    # Por cada escala realizamos la deteccion de puntos Harris
    for nivel in lista_images:

        # Obtenemos los autovalores y autovectores propios asociados
        # a la matriz Harris en cada pixel
        informacion_pixel = cv2.cornerEigenValsAndVecs(src=nivel,
                            blockSize=block_size, ksize=k_size)

        # Separamos la informacion obtenida
        lambd = cv2.split(informacion_pixel)

        # Calculamos los valores del criterio de seleccion
        # (λ1·λ2) − 0.04 [(λ1+λ2)·(λ1+λ2)]
        k = 0.04
        matriz_criterio_seleccion.append(lambd[0]*lambd[1] - k *
                            ((lambd[0]+lambd[1])*(lambd[0]+lambd[1])))

    # Creamos matriz binaria de los puntos de Harris
    imagen_binaria = [((nivel >= umbral)*255)
                        for nivel in matriz_criterio_seleccion]

    # Devolvemos la matriz Harris y su respectiva imagen binaria
    return matriz_criterio_seleccion, imagen_binaria
```

supresionNoMaximos: con esta función se implementa la fase de supresión de valores no máximos sobre la matriz de puntos Harris de la anterior función. Esta función nos servirá para eliminar los falsos máximos y así quedarnos con los puntos que son máximos locales de su vecindario. Recibiremos por entrada:

- Los *puntos Harris* obtenidos en la anterior función.
- La *imagen binaria* obtenida en la anterior función.
- Y el *tamaño de la ventana*.

Partimos de la imagen binaria y le preguntamos a cada posición por el valor 255, las que respondan *sí* las guardamos en una nueva lista. Obteniendo así, una lista con tales posiciones. En un bucle recorremos esas posiciones comprobando si son máximos locales. Para ello usaremos varias funciones. La función **maximoLocalCentro**, nos dice si el valor del centro de una ventana de tamaño impar es máximo local, devolviendo *True* si lo es y *False* en caso contrario. A continuación, debemos comparar el valor máximo de la ventana con el valor donde la posición vale 255. Si nos devuelve que es un máximo local pasamos a usar la función **modificarPixelesEntorno**, la cual modifica a 0 todos los píxeles de un rectángulo dado, es decir, pone a negro todos los píxeles de un entorno menos el central. Si nos devuelve que no es un máximo local, no haremos nada, seguirá estando el píxel a 0. Con esto conseguimos

una nueva matriz binaria en donde si la posición vale 255 es un máximo local y si la posición es un 0, no lo es. Nuestra función devolverá esta matriz binaria.

```
def supresionNoMaximos(puntos_harris, imagen_binaria, tam_ventana):

    '''
    Nos dice si el valor del centro de una ventana es máximo local
    '''
    def maximoLocalCentro (tam_ventana):

        # Obtenemos el número de filas y de columnas de la ventana
        centro_filas = np.shape(tam_ventana)[0]
        centro_col = np.shape(tam_ventana)[1]

        # Calculamos el valor máximo de la ventana
        maximo = np.argmax(tam_ventana)

        # Calculamos el valor del centro de nuestra ventana
        centro = (centro_filas * centro_col)//2

        # Si es maximo local devolvemos True, sino devolvemos False
        if (maximo == centro): return True
        else: return False

    '''
    Función que sobre una imagen binaria inicializada a 255, sea
    capaz de modificar a 0 todos los pixeles de un rectángulo dado,
    es decir, pone en negro todos los píxeles de un entorno menos el
    central
    '''
    def modificarPixelesEntorno(imagen_binaria, rectangulo, i , j):

        # Calculamos el centro de la ventana
        centro_filas = rectangulo/2
        centro_col = rectangulo/2

        # Si no estoy en el centro de la ventana,
        # modificar a 0 todos los pixeles de un rectángulo dado
        if not (i == centro_filas) and not (j == centro_col):
            imagen_binaria[rectangulo,rectangulo] = 0

    '''
    Fijar un tamaño de entorno/ventana, recorrer la matriz binaria
    creada preguntando en cada posición de valor 255 si el valor
    del pixel correspondiente de la matriz Harris es máximo local o no
    '''
    for nivel in range(tam_ventana):

        # Recorrer la matriz binaria, preguntando por el valor 255
        harris = (imagen_binaria[nivel])
        harris_255 = (np.where(harris == 255))

        # Obtenemos el tamaño de la lista con las posiciones a 255
        len_harris_255 = len(harris_255[1])

        # Comprobamos si el valor del pixel correspondiente a esa
        # nueva lista es máximo local o no
        for i in range(len_harris_255):

            # Calculamos el valor de las filas y las columnas
            fila = harris_255[0][i]; columna = harris_255[1][i]
```

```
            # Calculamos las cuatro esquinas del rectángulo
            izquierda = fila - tam_ventana
            derecha = fila + tam_ventana + 1
            abajo = columna - tam_ventana
            arriba = columna + tam_ventana + 1

            # Si es máximo local ponemos a negro a todos
            # los píxeles de su entorno
            if maximoLocalCentro(puntos_harris[nivel][izquierda:derecha,
                                                    abajo:arriba]):
                modificarPixelesEntorno(imagen_binaria[nivel],
                                        tam_ventana, fila, columna)

            # Si no es máximo local lo ponemos a 0
            else: imagen_binaria[nivel][fila, columna] = 0

    return imagen_binaria
```

Una vez, tenemos hechas estas funciones ya podemos visualizar los puntos Harris de las imágenes con y sin supresión de máximos locales. Visualicémoslo para comprobar que vamos por buen camino.

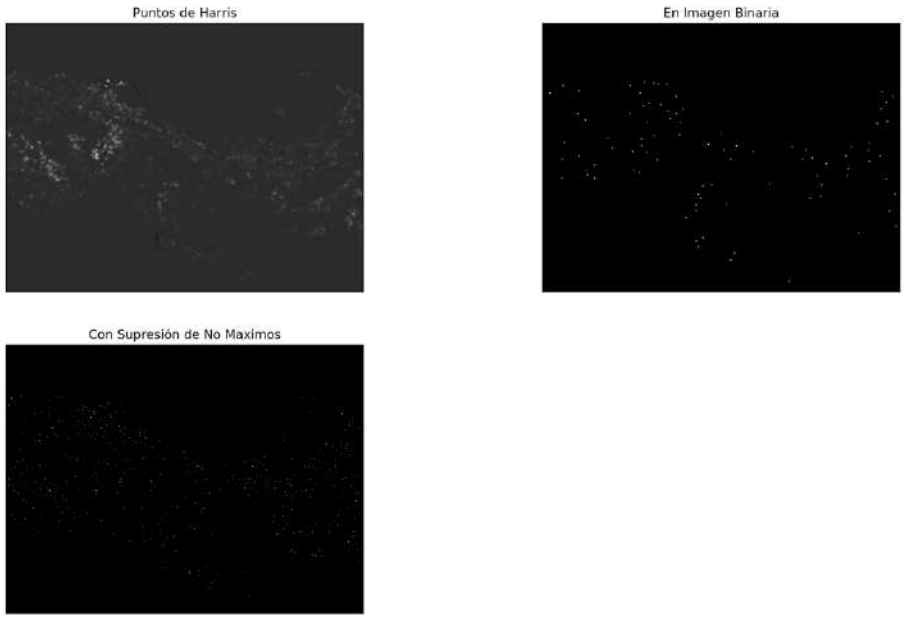

Figura 1: Visualización de Puntos Harris en la imagen de Yosemite

Para este ejemplo hemos usado los parámetros siguientes:

```
block_size=5; k_size=5; tam_ventana=3; umbral=0.012
```

Debemos tener en cuenta que dependiendo de los criterios que pongamos obtendremos una salida u otra ya que dejaremos que pasen más puntos Harris o no. Después de probar con varias opciones, pongo el resultado que veo más representativo. Como podemos ver en las anteriores imágenes, se cogen los puntos de los bordes de manera aceptable. Pero para saber, si la salida es correcta, probemos la detección de puntos Harris mediante la imagen del Tablero, ya que en esa imagen se distinguen a simple vista muy bien los bordes.

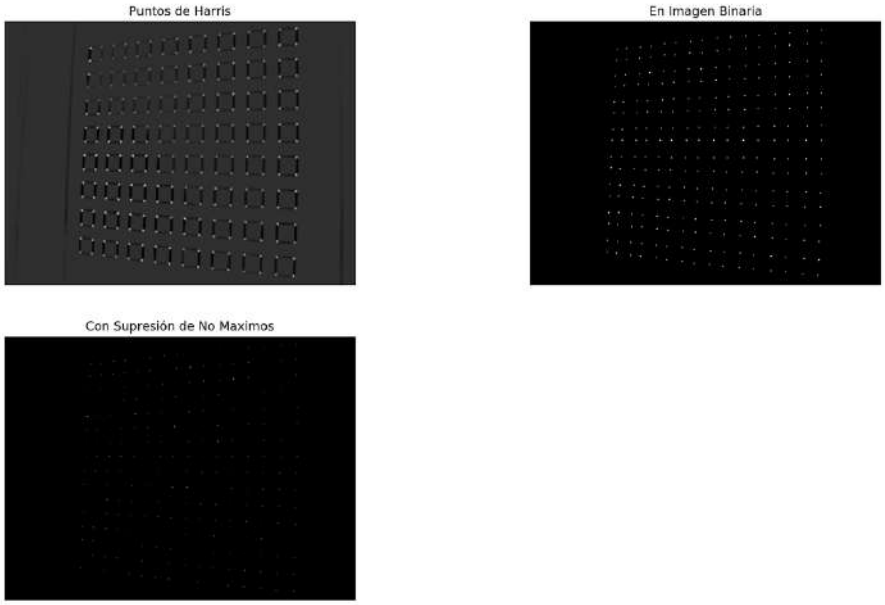

Figura 2: Visualización de Puntos Harris en la imagen de Yosemite

Hemos usado los parámetros siguientes:

```
block_size=5; k_size=5; tam_ventana=3; umbral=0.2
```

Si comparamos las imágenes binarias entre sí, se ve que hemos eliminado bastantes puntos entre las dos imágenes binarias, puesto que se han ido los que eran falsos máximos locales. Una vez hecha la supresión de valores no máximos procedemos a quedarnos con los puntos que tengan un mayor valor.

seleccionar500PuntosHarris: esta función ordena de mayor a menor los puntos resultantes de acuerdo a su valor Harris y selecciona al menos los 500 puntos mejores. Recibe como entrada los *puntos Harris* y la *imagen binaria* de la anterior función. Recorreremos cada escala de la pirámide y nos quedaremos con las posiciones donde la imagen binaria tenga de valor el 255, guardaremos las coordenadas de ese punto y nos creamos una tupla con *(cx, cy, nivel, valor_harris)* para cada punto. Una vez hecho, ordenamos por valor de Harris de mayor a menor valor y seleccionamos los 500 primeros.

```
def seleccionar500MejoresPuntosHarris(puntos_harris, imagen_binaria):

    # Tenemos los puntos que no se han ido en la fase de supresión de
    # valores no-máximos, los ordenamos por su valor mayor
    # Recorremos la lista de imágenes
    for nivel in range(6):

        # Guardamos los índices que son puntos de Harris (valen 255)
        harris = (imagen_binaria[nivel])
        indices_harris_255 = (np.where(harris == 255))

        # Nos quedamos con el valor de Harris de esa posición
        puntos_harris_255 = puntos_harris[nivel][indices_harris_255]

        # Cogemos las coordenadas x,y de los puntos y las juntamos
        x = indices_harris_255[0]; y = indices_harris_255[1]
```

```
        # Los metemos en lista (x, y, nivel, valor_harris)
        nueva = [(x[i],y[i], nivel, puntos_harris_255[i])
                            for i in range(len(indices_harris_255[0]))]

        # Vamos metiendo los valores
        mejores_puntos_harris = mejores_puntos_harris + nueva

    # Los ordenamos de mayor a menor y nos quedamos con los 500 primeros
    valor = sorted(mejores_puntos_harris[0:500],
                   key = lambda tup:tup[3])[::-1]

    return valor
```

Ahora solo nos queda pintar los puntos, para ello nos hemos creado una función que nos dibuja los círculos centrados en cada punto y de radio proporcional a la escala, también representaremos el ángulo de la orientación de cada punto. Como aún no he explicado como hago la orientación, cuando haga el apartado C explicaré esta función. Ahora simplemente veamos su salida para la imagen de Yosemite.

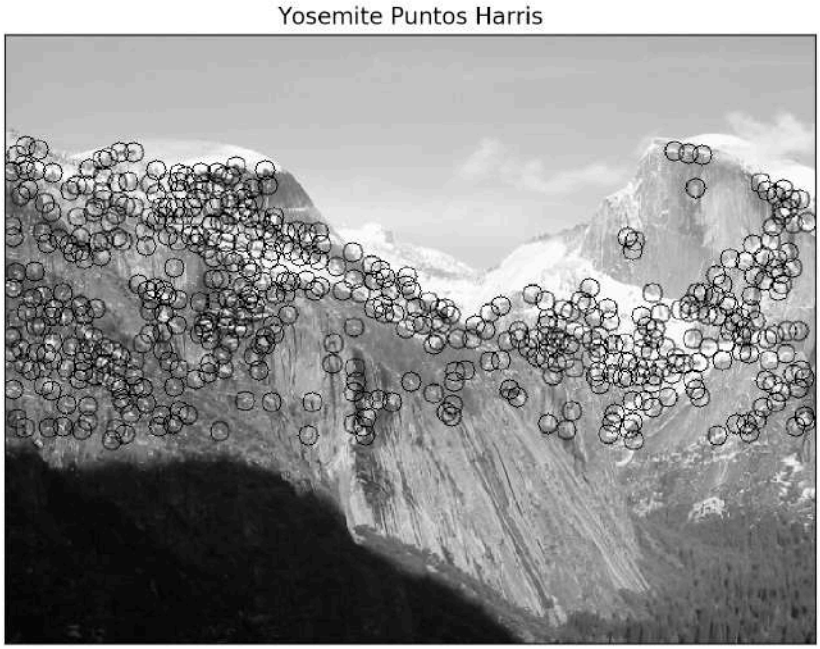

Figura 3: Visualización de Puntos Harris en la imagen de Yosemite

En la imagen observamos que solo se sacan puntos Harris de la escala original y viendo la salida de la función vemos que es correcta esa deducción. Mostremos como guardamos los valores en nuestra tupla *(x, y, escala, valor)*:

$$[(234, 162, 0, 0.50370437), (103, 182, 0, 0.45097011), (106, 205, 0, 0.34004107), \ldots$$
$$\ldots, (262, 506, 0, 0.01240451), (176, 572, 0, 0.012256866), (282, 338, 0, 0.012148639)]$$

Probando con diferentes umbrales y tamaño de ventana, elegimos el que dio mejor resultado:

```
block_size=5; k_size=5; tam_ventana=3; umbral=0.012
```

Pero si escogemos otros parámetros como por ejemplo, si aumentamos el umbral, estamos haciendo que pasen menos puntos, como se puede ver en la imagen siguiente:

```
block_size=5; k_size=5; tam_ventana=3; umbral=0.092
```

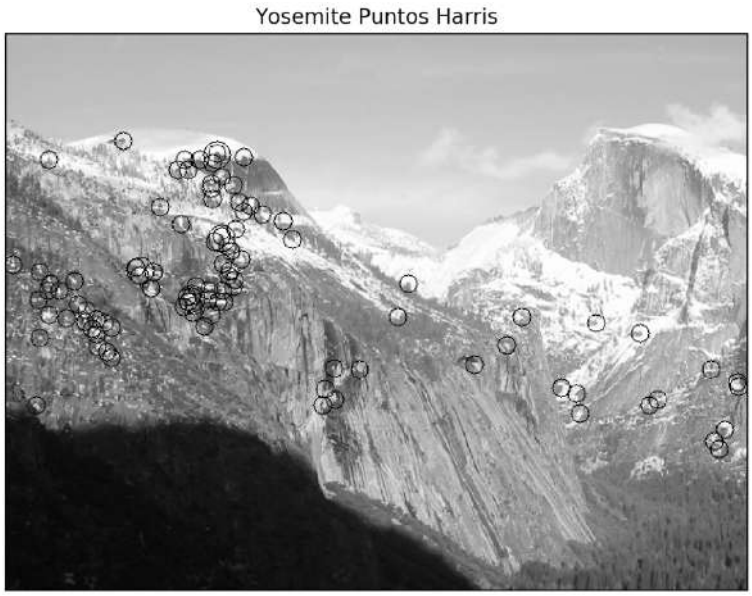

Figura 4: Visualización de Puntos Harris en la imagen de Yosemite

Sin embargo, vemos que aparecen puntos de otra escala:

$$[(234, 162, 0, 0.50370437), (103, 182, 0, 0.45097011), ...(52, 91, 1, 0.28365108),$$
$$.... (268, 432, 0, 0.092723481), (244, 445, 0, 0.092108577)]$$

Depende del corte y de los parámetros que escojamos dejaremos pasar unos puntos Harris u otros. Así que para para elegir el mejor resultado, lo he hecho a prueba y error.

Apartado b

Extraer los valores (cx, cy, escala) de cada uno de los puntos resultantes en el apartado anterior y refinar su posición espacial a nivel sub-píxel usando la función OpenCV `cornerSubPix()`) con la imagen del nivel de pirámide correspondiente. Actualizar los datos (cx, cy ,escala) de cada uno de los puntos encontrados.

Ya hemos extraídos los valores (cx, cy, escala) del apartado anterior, aunque en nuestro caso tenemos *(cx, cy, escala, valor_harris)* para cada uno de los puntos resultantes. A partir de esos puntos, tenemos que refinarlos a nivel de subpíxel. Para ello, nos hemos creado una función.

`refinarPuntosHarris`: con esta se refinan los puntos obtenidos en el apartado anterior. Recibiremos por entrada:

- El *vector de imágenes* de nuestra pirámide gaussiana, siendo la imagen correspondiente a su escala correspondiente.
- Y los *puntos Harris* obtenidos de la anterior función, los cuales refinaremos.

Por tanto, una vez obtenidos los mejores puntos de cada nivel aplicando los distintos criterios de selección que hemos usado en el anterior apartado, pasamos a refinar esos puntos a nivel de subpíxel. Para ello tenemos una tupla donde guardamos la información del punto. Por lo que para cada punto de cada escala, se refina a nivel de subpíxel llamando a la función de *OpenCV* `cornerSubPix` [3]. Almacenaremos en una tupla, igual que antes, los nuevos valores para las coordenadas de x e y. Por lo que, simplemente modificaremos los valores *(cx, cy, escala, valor_harris)*, por los nuevos *(cx_refinados, cy_refinados, escala, valor_harris)*.

```
def refinarPuntosHarris(lista_imagenes, puntos_harris, tam_ventana):

    # Seleccionamos las coordenadas X e Y de nuestros puntos harris
    selected_points_nueva = [(i[0],i[1]) for i in puntos_harris]

    # Convertimos en Array
    puntos = np.array(puntos_harris)

    # Recorremos todas las escalas que haya
    for i in range(int(np.max((puntos[:,2])))+1):

        # Ordenar por escala, y hacer subgrupos
        selected_points_nueva = [[punto[0],punto[1]]
                                 for punto in puntos[puntos[:,2]==i]]

        # Convertimos a array esa lista
        lista = np.array(selected_points_nueva, dtype=np.float32).copy()

        # Refinamos a nivel de subpixel
        # 40 es el número de iteraciones y 0.001 es la precisión
        cv2.cornerSubPix(image=lista_imagenes[i], corners=lista,
                         winSize=(tam_ventana, tam_ventana),
                         zeroZone=(-1, -1),
                         criteria=(cv2.TERM_CRITERIA_EPS +
                                   cv2.TERM_CRITERIA_COUNT, 40, 0.001))

        # Actualizamos la tupla con las nuevas coordenadas de X e Y
        for punto in lista:
            puntos_refinados.append((punto[0], punto[1],i))

    return puntos_refinados
```

Visualizamos el resultado:

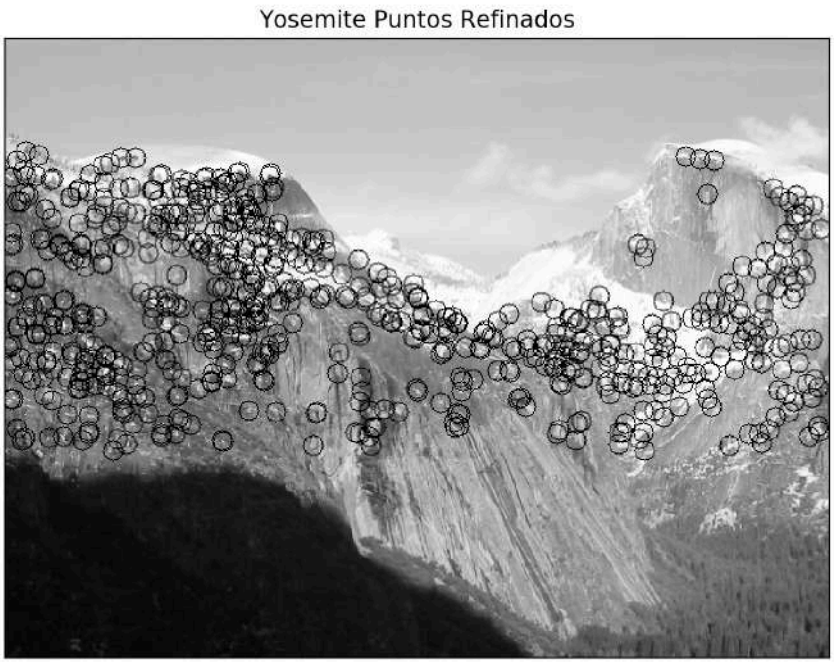

Figura 5: Visualización de Puntos Harris Refinados en la imagen de Yosemite

Para comprobar que de verdad se han refinado los puntos, comparemos las tuplas de los puntos sin refinar con los refinados.

- **Puntos Harris Sin Refinar:**
  $[(234, 162, 0, 0.50370437), (103, 182, 0, 0.45097011), (106, 205, 0, 0.34004107), \ldots, (262, 506, 0, 0.01240451), (176, 572, 0, 0.012256866), (282, 338, 0, 0.012148639)]$
- **Puntos Harris Refinados:**
  $[(234.97, 162.47375, 0), (104.01307, 180.96718, 0), (104.66759, 206.36581, 0), \ldots, (262.0, 506.0, 0), (176.0, 572.0, 0), (282.0, 338.0, 0)]$

Como se ven ahora los valores son más precios ya que se usan como flotantes, pero esto supone tener cuidado a la hora de dibujarlos, ya que para crear los círculos, las coordenadas de los puntos deben ser como enteros.

Apartado c

Calcular la orientación relevante de cada punto Harris usando el arco tangente del gradiente en cada punto. Previo a su cálculo se deben aplicar un alisamiento fuerte a las imágenes derivada-x e y derivada- y, en la escala correspondiente, como propone el paper MOPS de Brown&Szeliski&Winder. (Apartado 2.5). Añadir la información del ángulo al vector de información del punto. Pintar sobre la imagen original círculos con un segmento indicando la orientación estimada en cada punto.

***Nota**: He sacado este ejercicio, con las anotaciones que se dieron en clase.*

obtenerOrientacionPunto: función que obtiene el ángulo del gradiente de ese punto. Recibiremos por entrada:

- El *vector de imágenes* de nuestra pirámide gaussiana, siendo la imagen reescalada correspondiente.
- Y los *puntos Harris refinados* obtenidos de la anterior función, a los cuales calcularemos su dirección.

En esta función, calculamos las derivadas de la imagen en la escala correspondiente con respecto x e y. Para ello hacemos uso de las funciones definadas en la *práctica 1*. El proceso es el siguiente: después de haber obtenido los puntos refinados, calculamos el ángulo del gradiente de todos los puntos. Pero antes de derivar aplicaremos un alisado (filtro gaussiano) fuerte σ=4.5. Luego calculamos el ángulo del punto Harris refinado. Para calcular el ángulo haremos:

$$arctangente\left(\frac{dy}{dx}\right)$$

Una vez hecha esta operación, tenemos que convertir el ángulo a radianes para que OpenCV pueda trabajar con ello y así visualizarlos, para ello multiplicamos el resultado por $(180/\pi)$. Tras esto añadimos un nuevo campo a nuestra tupla, la cual tendrá la orientación de ese ángulo junto con su escala y su coordenada x e y . Observemos como estará creada la tupla:

$$[(234.97000122070312, 162.4737548828125, 0, 0.0), \dots,$$
$$(213.85371398925781, 61.584850311279297, 0, 90.000002504478161), \dots,$$
$$(254.0, 74.0, 0, 74.1974855954399), \dots, (147.0, 633.0, 0, 36.012718814025831), \dots,$$
$$(176.0, 572.0, 0, 73.667292161117672), \dots]$$

```
def obtenerOrientacionPunto(lista_imagenes, puntos_refinados):

    # Tupla donde guardaremos las orientaciones
    # (x, y, escala, orientacion)
    orientaciones = []

    # Convertimos a array los puntos refinados
    puntos = np.array(puntos_refinados)

    # Recorremos para cada escala de la imagen
    for i in range(int(np.max((puntos[:,2])))+1):

        # Calculamos las derivadas en X e Y de la imagen
        # y las alisamos con sigma=4.5 (funciones P1)
        img1 = alisar_imagen(lista_imagenes[i], sigma=4.5,
                             borde=cv2.BORDER_DEFAULT)
        derivada_x, derivada_y = primera_derivada(img1, 1,
                                                  cv2.BORDER_DEFAULT)

        # Me quedo con los puntos refinados de una la escala i
        refined_points_nueva = [[punto[0],punto[1]]
                                for punto in puntos[puntos[:,2]==i]]

        # Para cada punto añado (x, y, escala, orientacion)
        for punto in refined_points_nueva:
              orientaciones.append((punto[0], punto[1], i,
```

```
            ((np.arctan2(derivada_y[int(punto[0]),int(punto[1])],
                         derivada_x[int(punto[0]),int(punto[1])]))
              *180/np.pi)))

    # Convierto a array
    angulo = np.array(orientaciones)

    return orientaciones
```

Por último, creamos la imagen de salida. Para ello usaremos la función creada **obtenerOrientacionPunto**. Esta función es la que se ha usado para las salidas de los apartados anteriores. La función recibe como entrada:

- La *imagen original* a la que le dibujaremos los círculos y los ángulos.
- Los *puntos Harris*, refinados o no, dependiendo del apartado que estemos ejecutando.
- El *título de la imagen* cuando representemos.
- Las *orientaciones* de cada punto (variable opcional).
- Y una variable booleana *usarOrientaciones* con la que distinguiremos si queremos pintar las orientaciones o no.

La imagen de salida contendrá un círculo por cada punto Harris, cuya radio será inversamente proporcional a la escala, es decir, cuanto más pequeña sea la escala de la imagen, mayor será el círculo del punto Harris. Contemplaremos si usamos puntos de una escala solo o de varias escalas. También, visualizaremos los asociados a cada punto. Donde el punto será igual a el centro del círculo más el $(\cos(\text{ángulo}), \text{sen}(\text{ángulo}))$.

```
def dibujarCirculoLineas(imagen, puntos_harris, titulo,
                  orientaciones = None, usarOrientaciones = False):

    # Convertimos en array las orientaciones
    array = np.array(orientaciones)

    # Recorremos para todos los puntos harris
    for punto in puntos_harris:

        escala = punto[2] # Obtenemos la escala

        # Si hay más de una escala, el círculo deberá estar centrado
        # en cada punto y de radio proporcional a la escala
        if escala > 0:
            # Dibujamos los círculos en la imagen original
            cv2.circle(img=imagen, center=(int(punto[1]*escala*2),
                       int(punto[0]*escala*2)), radius=(escala+1)*6,
                       color=0, thickness=0)

            # Si usamos los ángulos
            if usarOrientaciones:
```

```
                # Comparar tuplas puntos refinados con orientaciones
                # Si (x,y,escala) es = nos quedamos con ese punto
                p = array[(array[:,0]==punto[0]) &
                   (array[:,1]==punto[1]) & (array[:,2]==punto[2])]

                # Obtenemos el ángulo de ese punto
                angulos = p[:,3]

                # Dibujamos el ángulo de ese punto
                cv2.arrowedLine(img=imagen,
                    pt1=(int(punto[1])*(escala+1),
                         int(punto[0])*(escala+1)),
                    pt2=(int(punto[1])*(escala+1) +
                         math.floor(np.sin(angulos)*(escala+1)*6),
                         int(punto[0])*(escala+1) +
                         math.floor(np.cos(angulos)*(escala+1)*6)),
                    color=0,thickness=0)

        # Si solo tenemos puntos en la escala original
        else:
            # Dibujamos los círculos en la imagen original
            cv2.circle(img=imagen, center=(punto[1], punto[0]),
                       radius=8, color=0, thickness=0)

            # Si usamos los ángulos
            if usarOrientaciones:
                # Comparamos (x,y,escala) de ambas tuplas y nos
                # quedamos con ese punto
                p = array[(array[:,0]==punto[0]) &
                    (array[:,1]==punto[1]) & (array[:,2]==punto[2])]

                # Obtenemos el ángulo de ese punto
                angulos = p[:,3]

                # Dibujamos el ángulo de ese punto
                cv2.arrowedLine(img=imagen,
                    pt1=(int(punto[1]),int(punto[0])),
                    pt2=(int(punto[1])+
                         math.floor(np.sin(angulos)*8),
                         int(punto[0]) +
                         math.floor(np.cos(angulos)*8)),
                    color=0,thickness=0)

    representar_imagenes([imagen], [titulo])
```

Figura 6: Visualización de Puntos Harris Refinados con sus respectivos Ángulos

En la imagen, hemos usados los mismo parámetros que en los anteriores apartados, como hemos dicho antes se ve como todos los puntos corresponden a la misma escala. También se aprecia la dirección de los puntos. Además se observa como los círculos están seleccionando las esquinas. El resultado fue escogido entre los mejores que hice mediante prueba y error, variando el umbral, tamaño de la ventana…

Apartado d

Usar el vector de keyPoint extraidos para calcular los descriptores SIFT asociados a cada punto (cv2.DescriptorExtractor_create.compute())

Para realizar este apartado, parto de la tupla que hemos obtenido con los puntos refinados (*cx, cy, escala, valor_harris*). Debemos convertir esta tupla en objetos de tipo Keypoint. Para ello nos creamos una función a la que le pasemos por entrada nuestros puntos Harris, para que podamos transformarlos en un vector de Keypoints. Para ello usamos la función de OpenCV `KeyPoints` [4], a la cual le pasamos las coordenadas de nuestros puntos y la escala, y con ellos obtenemos el vector de Keypoints. Luego para visualizarlos usamos las funciones de OpenCV [5] `SIFT.compute` y `drawKeypoints`.

```
def keypoints(puntos_harris, imagen):

    # Recorremos nuestros puntos Harris
    for punto in puntos_harris:
        keypoints.append(cv2.KeyPoint(x=punto[1], y=punto[0],
                         _size=(punto[2]+1)*6))
```

```
    # Usamos el descriptor de SIFT
    sift = cv2.xfeatures2d.SIFT_create()

    # Obtenemos los keypoints y descriptores para dibujarlos
    keypoints, descriptor = sift.compute(imagen, keypoints)

    # Dibujamos los puntos
    resultado = cv2.drawKeypoints(imagen, keypoints, imagen.copy(),
                color=(0,0,255), flags=0)

    # Los representamos
    representar_imagenes([resultado], ["Mis Keypoints"])

    return (keypoints, descriptor)
```

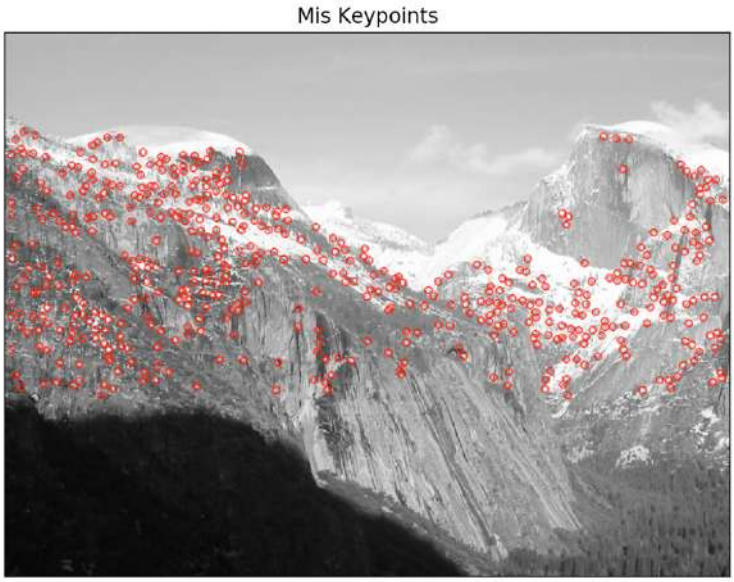

Figura 7: Visualización de Puntos Harris con el vector de Keypoints

Como se aprecia, podemos decir con una alta confianza que la salida es parecida a la de las anteriores figuras. Puesto que son nuestro puntos, lo que único que hemos hecho ha sido cambiarlos a otro tipo de dato.

Ejercicio 2

Usar el detector- descriptor SIFT de OpenCV sobre las imágenes de Yosemite (cv2.xfeatures2d.SIFT_create()). Extraer sus listas de keyPoints y descriptores asociados. Establecer las correspondencias existentes entre ellos usando el objeto *BFMatcher* de *OpenCV*. Valorar la calidad de los resultados obtenidos en términos de correspondencias válidas usando los criterios de correspondencias "BruteForce+crossCheck y "Lowe-Average-2NN". (2.0 puntos) (*NOTA: Si se usan los resultados del*

puntos anterior en lugar del cálculo de SIFT de OpenCV la valoración es de 2.5 puntos)

Para realizar este ejercicio usaremos el detector SIFT. Primero explicaremos la extracción de sus listas de keyPoints y descriptores asociados para el criterio **BruteForce+crossCheck**.

establecerCorrespondenciasFuerzaBruta [6]: la función tiene por entrada dos imágenes con las cuales estableceremos las correspondencias. Extraemos los keypoints y descriptores de cada imagen. Una vez obtenidos, obtenemos las correspondencias entre ambas imágenes, para ello usamos un objeto de tipo BFMatcher el cual las obtendrá mediante fuerza bruta. Lpasamos a su constructor para que use la normal L2 y que use *cross-check* (validación cruzada). Con esto obtenemos un vector de correspondencias, el cual nos identifica los keypoints que hay relacionados entre ambas imágenes. Por último, ordenamos por mejor distancia y visualizar 100 de ellas.

```
def establecerCorrespondenciasFuerzaBruta(imagen1, imagen2):

    '''
    :param imagen1: queryImage
    :param imagen2: trainImage
    '''
    # Usamos el detector SIFT
    sift = cv2.xfeatures2d.SIFT_create()

    # Extraemos sus listas de keyPoints y descriptores asociados
    keypoints_imagen1, descriptors_imagen1 =
                    sift.detectAndCompute(image=imagen1, mask=None)
    keypoints_imagen2, descriptors_imagen2 =
                    sift.detectAndCompute(image=imagen2, mask=None)

    # Sacamos las correspondencias por fuerza bruta usando BFMatcher
    # Le pasamos que use la norma NORM_L2 y cross check a TRUE
    correspondecias = cv2.BFMatcher(normType=cv2.NORM_L2,
                                    crossCheck=True)

    # Sacamos las correspondencias entre ambas imágenes
    matches = correspondecias.match(descriptors_imagen1,
                                    descriptors_imagen2)

    # Ordenamos las correspondencias por mejor distancia
    matches = sorted(matches, key=lambda x: x.distance)

    # Cogemos 100 de las correspondecias extraidas para apreciar la
    # calidad de los resultados
    resultado = cv2.drawMatches(img1=imagen1,
              keypoints1=keypoints_imagen1, img2=imagen2,
              keypoints2=keypoints_imagen2, matches1to2=matches[:100],
              outImg=imagen2.copy(), flags=0)

    representar_imagenes([resultado],
                         ['Correspondencias por Fuerza Bruta'])
```

En la imagen de salida veremos todos los keypoints extraídos de ambas imágenes y las 100 mejores correspondencias.

Figura 8: Correspondencias entre Yosemite1 y Yosemite2

Figura 9: Correspondencias entre Yosemite 5 y Yosemite6

En ambas imágenes se ve que las correspondencias son buenas ya que los puntos hacen referencia a la misma posición de la montaña.

Segundo, explicaremos la extracción de las listas de keyPoints y descriptores asociados para el criterio **Lowe-Average-2NN**. La idea principal es la misma que para el de fuerza bruta, solo que ahora obtenemos las mejores *k* coincidencias, en este caso *k*=2 y nos quedamos con la mejor distancia de ambas, es decir, la menor.

`establecerCorrespondenciasFuerzaBruta` [6]: la función tiene por parámetro de entrada dos imágenes con las cuales estableceremos las correspondencias. Extraeremos los keypoints y descriptores de cada imagen. Una vez obtenidos, obtenemos las correspondencias entre ambas imágenes, para ello usamos un objeto de tipo BFMatcher sin fuerza bruta (sin validación cruzada) y usando la normal L2. Con esto obtenemos un vector de correspondencias, el cual nos identifica los keypoints que hay relacionados entre ambas imágenes. Usaremos `BFMatcher.knnMatch()` para obtener las *k* mejores coincidencias, siendo k=2. Es decir, nosotros tenemos la posición de un punto, y en la otra imagen obtenemos dos posibles correspondencias, de entre esas dos nos quedamos con la mejor distancia de ambas y visualizar 100 de ellas.

```python
def establecerCorrespondencias2NN(imagen1, imagen2):

    # Usamos el detector SIFT (Inicializar)
    sift = cv2.xfeatures2d.SIFT_create()

    # Extraemos sus listas de keyPoints y descriptores asociados
    keypoints_imagen1, descriptors_imagen1
                    sift.detectAndCompute(image=imagen1, mask=None)
    keypoints_imagen2, descriptors_imagen2 =
                    sift.detectAndCompute(image=imagen2, mask=None)

    # Sacamos las correspondencias con BFMatcher
    # le pasamos que use la norma NORM_L2 y cross check a FALSE
    correspondecias = cv2.BFMatcher(normType=cv2.NORM_L2,
                                    crossCheck=False)

    # Usaremos BFMatcher.knnMatch() para obtener los k mejores
    # coincidencias, tomamos k=2
    matches = correspondecias.knnMatch(descriptors_imagen1,
                                       descriptors_imagen2, k=2)

    # Aplicamos la distancia entre ambas coincidencias y
    # y nos quedamos la mejor
    aceptadas = []
    for m, n in matches:
        if m.distance < 0.70 * n.distance:
            aceptadas.append([m])

    # Cogemos 100 de las correspondecias extraidas para apreciar la
    # calidad de los resultados
    numero_correspondencias = 100

    resultado = cv2.drawMatchesKnn(img1=imagen1,
                    keypoints1=keypoints_imagen1, img2=imagen2,
                    keypoints2=keypoints_imagen2,
                    matches1to2=aceptadas[:numero_correspondencias],
                    outImg=imagen2.copy(), flags=2)

    representar_imagenes([resultado],['Correspondencias 2NN'])
```

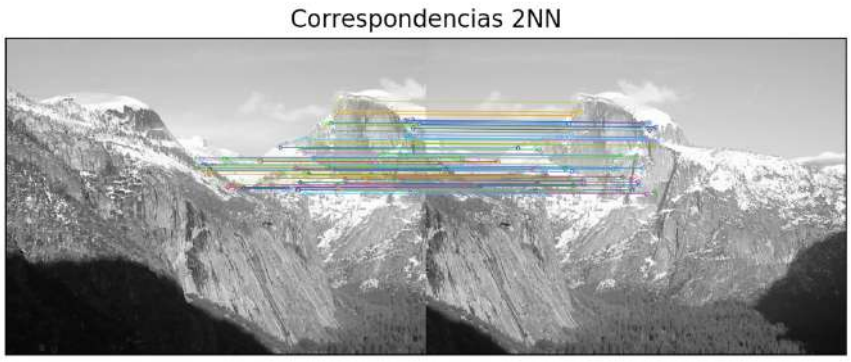

Figura 10: Correspondencias entre Yosemite1 y Yosemite2

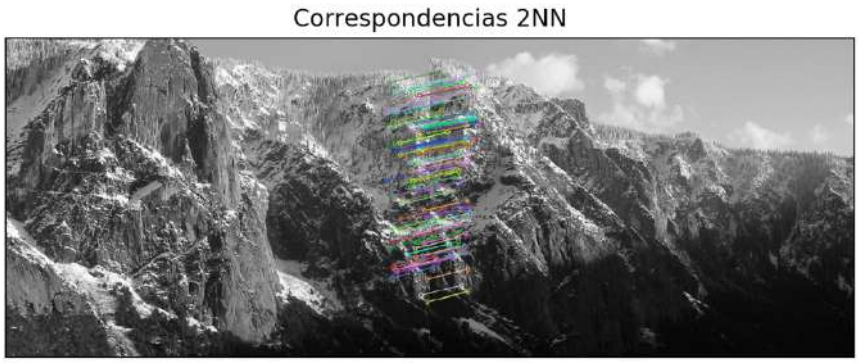

Figura 11: Correspondencias entre Yosemite 5 y Yosemite6

Una vez realizado vamos a comparar ambos criterios, aunque en *match KNN* no hemos dibujado los keypoints, podemos que ver que entre las Figura 9 y la 11, se escogen más o menos las mismas correspondencias. Sin embargo, para la Figura 8 y la 10, aunque ambas correspondencias estén bien, cuando usa el KNN, vemos como coge los correspondencias de una región de la montaña. Y sin embargo, para el de fuerza bruta, visualizando el mismo número de correspondencias selecciona correspondencias que están por toda la imagen.

Ahora vamos a realizar tales correspondencias usando los keypoints del *apartado 1D*. Para ello nos usamos una función que mediante los nuevos keypoints y mediante el descriptor SIFT establece la correspondencia entre las dos imágenes. La idea es la misma, solo que ahora le pasamos los keypoints ya obtenidos de nuestra función del 1D.

```
def correspondenciasMisKeyPoints (keypoints1, imagen1, descriptor1,
                                  keypoints2, imagen2, descriptor2):

    # Sacamos las correspondencias por fuerza bruta usando
    # un objeto de tipo BFMatcher
    # le pasamos que use la norma NORM_L2 y cross check a TRUE
    bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)

    # Sacamos las correspondencias
    matches = bf.match(descriptor1, descriptor2)

    # Ordenamos las correspondencias por mejor distancia
    matches = sorted(matches, key=lambda x: x.distance)

    # Dibujo los puntos con sus correspondencias
    resultado = cv2.drawMatches(imagen1, keypoints1, imagen2,
              keypoints2, matches[:50], outImg=imagen2.copy(), flags=0,
              matchColor=(0,0,255))

    # Representamos la imagen
    representar_imagenes([resultado],
                         ["Correspondencias entre misKeypoints"])
```

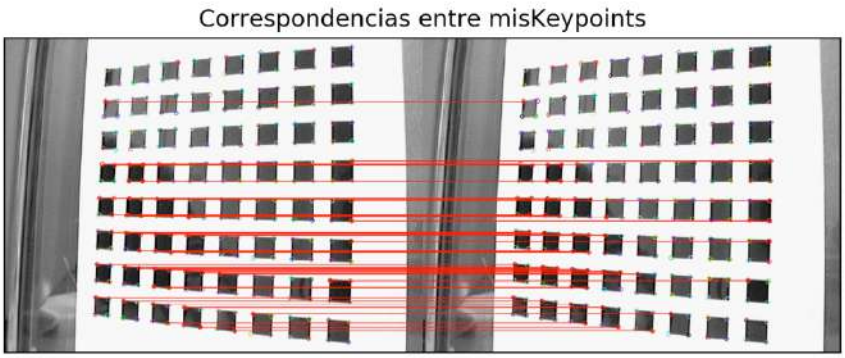

Figura 12: Correspondenciasentre Tableros mediante Fuerza Bruta

Ejercicio 3

Escribir una función que genere un Mosaico de calidad a partir de N=3 imágenes relacionadas por homografías, sus listas de keyPoints calculados de acuerdo al punto anterior y las correspondencias encontradas entre dichas listas. Estimar las homografías entre ellas usando la función cv2.findHomography(p1,p2,CV_RANSAC,1). Para el mosaico será necesario. a) definir una imagen en la que pintaremos el mosaico; b) definir la homografía que lleva cada una de las imágenes a la imagen del mosaico; c) usar la función cv2. warpPerspective() para trasladar cada imagen al mosaico (ayuda: mirar el flag *BORDER_TRANSPARENT* de warpPerspective para comenzar).

[7] Para este apartado he usado la técnica **Lowe-Average-2NN** del apartado anterior junto con el uso de las homografías. Para realizar el mosaico 3N, primero se necesita entender el mosaico 2N. Expliquemos que hacemos:

`mosaicoN3`: esta función genera un mosaico para 3 imágenes y tiene por parámetro de entrada tres imágenes con las cuales estableceremos las correspondencias dos a dos (A con B y B con C), mediante la función **establecerCorrespondencias2NN**. Extraeremos los keypoints y descriptores de cada imagen. Una vez obtenidos, obtenemos las correspondencias entre ambas imágenes, para ello usamos un objeto de tipo BFMatcher. Con esto obtenemos dos vectores de correspondencias, el cual nos identifica los keypoints que hay relacionados entre las imágenes. Usaremos `BFMatcher.knnMatch()` para obtener las *k* mejores coincidencias, siendo k=2. Una vez hecho, calculamos las dos homografías de la imagen. Una vez construidos estos vectores se usa la función de OpenCV `findHomography` a la cual se le pasan como parámetros los dos vectores, el uso de la técnica *cv2.RANSAC* y 1. Con esto, obtenemos 2 homografías. Y a partir de las homografía ya podemos construir nuestro mosaico.

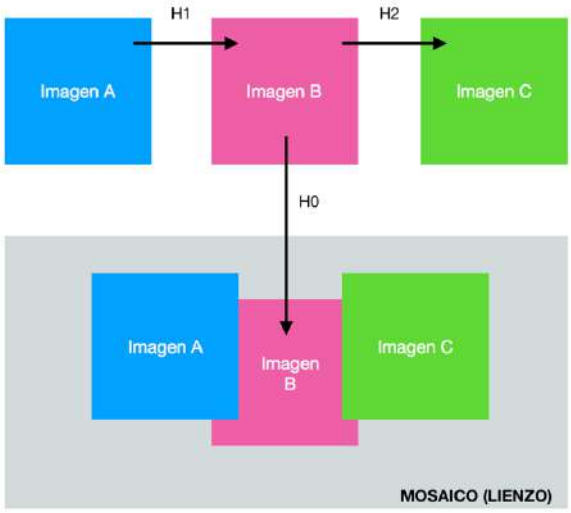

Figura 13: Generación de un mosaico para una 3 imágenes

Vamos a empezar a construir nuestro mosaico. Lo primero que tenemos que hacer es crear una imagen que tenga de altura y anchura lo mismo que las tres imágenes juntas, así para cuando las peguemos no haya problemas de visualizarlo. Como se ve en la figura 13, trasladamos la imagen central (B) al centro del mosaico. Para ello hacemos uso de la matriz de traslación [8]. Una vez colada en nuestra imagen de salida, basta usar la función de OpenCV `warpPerspective()` con las imágenes restantes y homografías calculadas.

```
def mosaicoN3(imageA, imageB, imageC):

    # Usamos el detector SIFT (Inicializar)
    descriptor = cv2.xfeatures2d.SIFT_create()

    # Extraemos las listas de keyPoints y descriptores a cada imagen
    keypointsA, descriptorA = descriptor.detectAndCompute(imageA, None)
    keypointsB, descriptorB = descriptor.detectAndCompute(imageB, None)
    keypointsC, descriptorC = descriptor.detectAndCompute(imageC, None)

    # Convertimos los objetos de Keypoint a Arrays
    kpsA = np.float32([kp.pt for kp in keypointsA])
    kpsB = np.float32([kp.pt for kp in keypointsB])
    kpsC = np.float32([kp.pt for kp in keypointsC])

    # Calculamos las correspondencias entre la IMAGEN A y la IMAGEN B
    matchesAB = establecerCorrespondenciasKNN(imageA, imageB)

    # Creamos la homografia si existen un mínimo de correspondencias
    if len(matchesAB) > 4:

        # Construimos los puntos que se le pasaran a la homografia
        puntosA = np.float32([kpsA[i] for (_, i) in matchesAB])
        puntosB = np.float32([kpsB[i] for (i, _) in matchesAB])

        # Calculamos la homografia
        H, status = cv2.findHomography(puntosA, puntosB, cv2.RANSAC, 1)

    # Calculamos las correspondencias entre la IMAGEN B y la IMAGEN C
    matchesBC = establecerCorrespondenciasKNN(imageB, imageC)
```

```
    # Creamos la homografia si existen un mínimo de correspondencias
    if len(matchesBC) > 4:
        # construct the two sets of points
        puntosB = np.float32([kpsB[i] for (_, i) in matchesBC])
        puntosC = np.float32([kpsC[i] for (i, _) in matchesBC])

        # compute the homography between the two sets of points
        H2, status2 = cv2.findHomography(puntosB, puntosC, cv2.RANSAC,1)

    # Primero calculamos translación de la imagen central a su
    # homografia al mosaico
    translacion = np.matrix([[1, 0, imageB.shape[0]], [0, 1,
                        imageB.shape[1]], [0, 0, 1]], dtype=np.float32)

    # La pegamos con warpPerspective
    result = cv2.warpPerspective(imageB, translacion,
                  dsize=(imageB.shape[1]*4, imageB.shape[0]*4),
                  borderMode=cv2.BORDER_TRANSPARENT)

    # Pegamos la siguiente imagen, a partir de la translación y su H
    cv2.warpPerspective(imageA, translacion*H, dst=result,
                        borderMode=cv2.BORDER_TRANSPARENT,
                        dsize=(imageA.shape[1]*4, imageA.shape[0]*4))

    # Pegamos la siguiente imagen, a partir de la translación, de la
    # anterior H y de su H
    cv2.warpPerspective(imageC, translacion*H*H2 , dst=result,
                        borderMode=cv2.BORDER_TRANSPARENT,
                        dsize=(imageC.shape[1]*4, imageC.shape[0]*4))

    return result
```

Figura 14: Generación de un mosaico con 3 imágenes

Ejercicio 4

Lo mismo que en el punto anterior pero para N > 5.

Para realizar este ejercicio hemos usado la misma idea que en el anterior, solo que ahora hacemos uso para establecer las correspondencias del criterio **BruteForce+crossCheck**. Uso este criterio, para ver que tanto uno como otro hacen un buen mosaico. Ahora debemos contemplar que tenemos N imágenes. Para ello nos creamos una imagen negra que tenga de alto y ancho la suma de las todas las imágenes que vamos a montar. Primero obtenemos con el descriptor-detector SIFT, los keypoints y descriptores de cada imagen para nuestro vector de imágenes. Luego calculamos mediante fuerza bruta los vectores de correspondencias entre las imágenes. Así que si tenemos *n* imágenes, obtendremos *n-1* imágenes, una por cada pareja de imágenes adyacente. Todo esto, estará metido un bucle, el cual recorra la lista de imágenes. Por tanto, los keypoints de la imagen *i* estarán en correspondencia con los keypoints de la imagen *i+1*.

Ahora procedemos a calcular la matriz de homografía cada par de imágenes. Para ello hacemos el mismo que proceso que en el anterior mosaico, solo que ahora deberemos tener un bucle donde vayamos recorriendo las imágenes y calculando las homografías de imágenes adyacente. Una vez que hemos calculado todas las homografías pasamos a crear el mosaico.

Primero, seleccionaremos la imagen del centro de nuestro vector de imágenes, y la trasladaremos al centro de la imagen con fondo negro creada. Luego iremos pegando las demás imágenes con la función `warpPerspective()` para sus respectivas imágenes y homografías. Una vez tenemos definido esto, tenemos que coser las imágenes que aparecen a la derecha y a la izquierda de la imagen trasladada al centro. Ahora, tenemos que tener en cuenta, que no podemos crear un bucle y meter todas las imágenes de golpe en el mosaico, ya que debemos tener en cuenta las que están a la derecha o izquierda de la imagen trasladada. Por tanto debemos crear primero la parte derecha y luego la parte izquierda.

- **Para la parte izquierda**: recorremos las imágenes desde la primera imagen hasta la imagen del centro, y nos creamos un bucle que vaya multiplicando las homografías, ya que la imagen *i* será la homografía *i al centro del mosaico* ($translación \cdot (productos\ de\ H[i])$). Esto hace que la imagen se cosa en el lugar que le toca.
- **Para la parte derecha**: recorremos las imágenes desde la del centro+1 hasta la última imagen de nuestro vector. Nos creamos un bucle que vaya multiplicando las homografías, ya que la imagen *i* será la homografía *i al centro del mosaico*. Sin embargo, ahora tendremos que hacer la inversa de las homografías (trabajamos con matrices regulares 3x3), ya que si antes la homografía *i* llevaba la imagen *i* a la imagen *i+1*, ahora la inversa lleva la imagen *i+1* a la *i*. Esto hace que la imagen se cosa en el lugar que le toca.

Debemos tener en cuenta que la función `warpPerspective`, debemos pasarle *BORDER_TRANSPARENT* para que así se pueda ver la imagen de manera uniforme.

```python
def mosaicoN(lista_imagenes):

    # Para extraer los puntos clave y los descriptores utilizamos SIFT
    sift = cv2.xfeatures2d.SIFT_create()

    # Para establecer las correspondencias entre las imágenes usaremos:
    # Brute-force matcher create method
    correspondencias = cv2.BFMatcher(normType=cv2.NORM_L2,
                                     crossCheck=True)

    # Saco los keypoints y descriptores de cada imagen del mosaico
    for i in range(len(lista_imagenes)):

        # Detectar y extraer características de la imagen
        kps, desc = sift.detectAndCompute(image=lista_imagenes[i],
                                          mask=None)

        # Convertir los keypoints con estructura Keypoint a un Array
        kps = np.float32([kp.pt for kp in kps])

        # Guarda en un lista las características de cada imagen
        keypoints.append(kps); descriptors.append(desc)

    # Obtengo el vector de correspondencias y la homografía de cada par
    # de imagenes adyacentes en el mosaico horizontal
    for i in range (len(lista_imagenes)-1):

        # Obtengo las correspondencias de la imagen i con la i+1
        matches = correspondencias.match(descriptors[i],
                                         descriptors[i+1])

        # Ordeno las coincicencias por el orden de la distancia
        matches = sorted(matches, key=lambda x: x.distance)
                         coincidencias.append(matches[i])

        # Extraigo los keypoints de la imagen i que están en
        # correspondencia con los keypoints de la imagen i+1
        keypoints_imagen1 = np.float32([keypoints[i][j.queryIdx]
                             for j in matches])
        keypoints_imagen2 = np.float32([keypoints[i+1][j.trainIdx]
                             for j in matches])

        # Calcular la homografía entre los dos conjuntos de puntos
        h, status = (cv2.findHomography(srcPoints=keypoints_imagen1,
                     dstPoints=keypoints_imagen2, method=cv2.RANSAC,
                     ransacReprojThreshold=1))
        homografia.append(h)

        # Borramos contenido keypoints para usar en siguiente iteracion
        np.array([row for row in keypoints_imagen1 if len(row)<=3])
        np.array([row for row in keypoints_imagen2 if len(row)<=3])

    # Nos creamos un imagen confondo negro, con un tamaño específico
    # (para que quepan las demás fotografías)
    ancho = lista_imagenes[0].shape[0]*6
    alto = lista_imagenes[0].shape[1]*4

    # Obtenemos la imagen del centro
    centro = len(lista_imagenes) // 5
```

```
    # Definimos la traslacion que nos pone la imagen central del mosaico
    # en el centro
    translacion = np.matrix([[1, 0, lista_imagenes[centro].shape[1]],
                     [0, 1, lista_imagenes[centro].shape[0]],
                     [0, 0, 1]], dtype=np.float32)

    # Llevamos esa imagen al centro de nuestro mosaico
    mosaico = cv2.warpPerspective(src=lista_imagenes[centro],
              M=translacion, dsize=(ancho, alto),
              borderMode=cv2.BORDER_TRANSPARENT)

    # Calculamos las homografias que delas imagenes de la
    # izquierda de la imagen central
    for i in range(0, centro):

        # Definimos la traslacion para las imágenes de la izquierda
        izquierda = np.matrix([[1, 0, 1],[0, 1, 1],[0, 0, 1]],
                              dtype=np.float32)

        for j in range(i, centro): izquierda = homografia[j] * izquierda

        # Las llevamos al mosaico
        cv2.warpPerspective(src=lista_imagenes[i],
             M=translacion*izquierda, dst=mosaico, dsize=(ancho, alto),
             borderMode=cv2.BORDER_TRANSPARENT)

    # Calculamos las homografias de las imagenes de la derecha de la
    # imagen central, debemos usar las inversas de las homografías, ya
    # que las homografias que se usan son de la imagen i a la i-1
    for i in range(centro + 1, len(lista_imagenes)):

        # Definimos la traslacion para las imágenes de la derecha
        derecha = np.matrix([[1, 0, 1], [0, 1, 1], [0, 0, 1]],
                            dtype=np.float32)

        for j in range(centro, i):
            derecha = derecha * np.linalg.inv(homografia[j])

        # Las llevamos al mosaico
        cv2.warpPerspective(lista_imagenes[i], M= translacion*derecha,
                            dst=mosaico, dsize=(ancho, alto),
                            borderMode=cv2.BORDER_TRANSPARENT)

    return mosaico
```

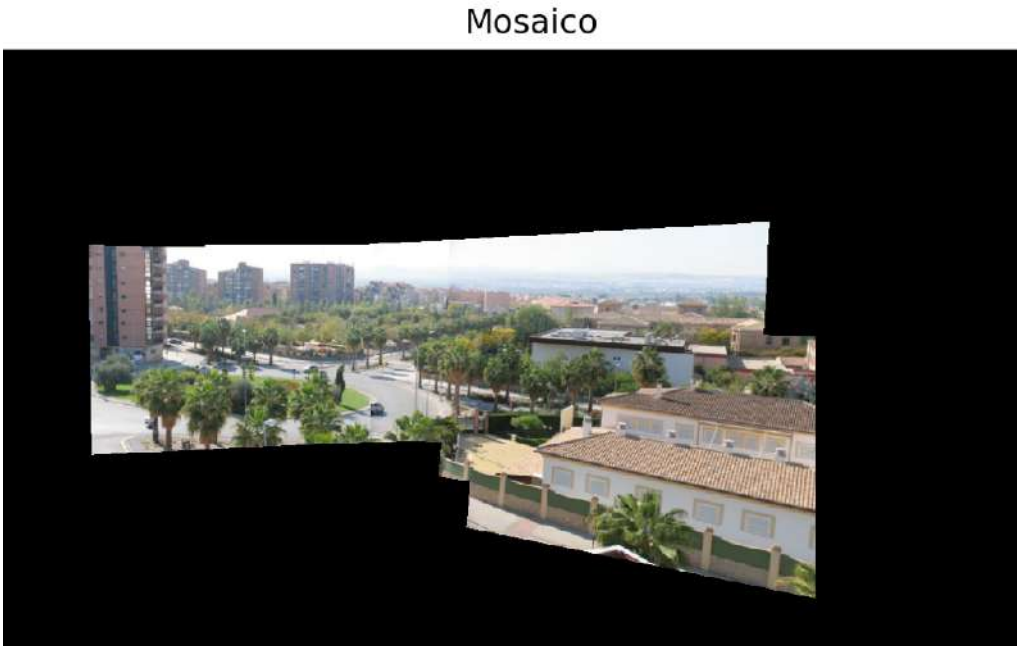

Figura 14: Mosaico para N imágenes.

También podríamos haber hecho con esta función el mosaico para 3N.

Bonus (usar las imágenes de Yosemite)

Ejercicio 1

Implementar de forma eficiente el detector propuesto en el paper de Brown&Szeliski&Winder (http://matthewalunbrown.com/papers/cvpr05.pdf)

Ejercicio 2

Implementar de forma eficiente el descriptor propuesto en el paper de Brown&Szeliski&Winder (2 puntos)

Ejercicio 3

Implementar de forma eficiente la estimación de una homografía usando RANSAC. (2 puntos)

Para realizar este ejercicio nos hemos basado en [9]. Primero debemos leer dos imágenes las cuales mediante `establecerCorrespondenciasFuerzaBruta`, obtendremos los correspondencias de ambas imágenes a partir de sus keypoints y descriptores. Una vez calculadas las correspondencias ya podremos pasar a aplicar el algoritmo RANSAC. Aplicaremos esta técnica,

sobre un subconjunto de datos aleatorios de las correspondencias calculadas anteriormente. Deberemos poner un tope de iteraciones a este programa.

```
# Vamos a crear una función con la que obtengamos la homografia si
# existen 4 correspondencias entre las imágenes
def obtenerHomografia(correspondencias):

    # Recorremos las correspondencias
    for i in correspondencias:

       # Obtenemos los puntos
        x = np.matrix([i[0], i[1], 1])
        y = np.matrix([i[2], i[3], 1])

        # Apuntes teoría Tema 3 (p. 48): Solving for Homographies
        ecuacion1= [x[0],x[1],1,0,0,0,y[0]*x[0],y[0]*x[1],y[0]*x[2]]
        ecuacion2= [0,0,0,x[0],x[1],1,y[1]*x[0],y[1]*x[1],y[1]*x[2]]

        lista.append(ecuacion1, ecuacion2)

    # Convertimos la lista en una matriz
    matriz = np.matrix(lista)

    # (Solution h = eigenvector SVD with smallest eigenvalue)
    # Hacemos la composición SVD
    s,v,d = np.linalg.svd(matriz)
    # Cogemos el minimo valor (homografía es 3x3)
    h = np.reshape(np.min(d),(3, 3))

    return h
```

```
# Algoritmo Ransac, creamos la homogbrafia a partir de las
# correspondencias de la anterior función
def ransac(correspondencias, umbral):

    # Variables a usar en el algorimot
    inliers = []

    # Obtenemos la longitud de las correspondencias
    len_correspondencias = len(correspondencias)

    # Establecemos un número de iteraciones
    for i in range(100):

        # Obtenemos 4 valores aleatorios de las correspondencias
        correspondencia1 = correspondencias[randrange (0,
                                            len_correspondencias)]
        correspondencia2 = correspondencias[randrange (0,
                                             len_correspondencias)]
        correspondencia3 = correspondencias[randrange (0,
                                             len_correspondencias)]
        correspondencia4 = correspondencias[randrange (0,
                                            len_correspondencias)]

        # Juntamos las correspondencias por columnas
        resultado = np.vstack((correspondencia1, correspondencia2))
        resultado = np.vstack((resultado, correspondencia3))
```

```
        resultado = np.vstack((resultado, correspondencia4))

        # Obtener homografia de esos puntos
        H = obtenerHomografia(resultado)

        # Obtener el mayor conjunt de inliers
        inliers = []
        for i in range(len_correspondencias):
            inliers = (correspondencias [i])
            i.append(inliers)

    return H, inliers
```

Referencias

[1] Función de OpenCV `cornerEigenValsAndVecs`
https://docs.opencv.org/2.4/modules/imgproc/doc/feature_detection.html?highlight=cornereigenvalsandvecs#cornereigenvalsandvecs

[2] Apuntes de Teoría "*Computer Vision: Corner and blob detection*", p. 36
https://docs.opencv.org/3.3.0/dc/dc3/tutorial_py_matcher.html

[3] Función de OpenCV `cornerSubPix`
https://docs.opencv.org/2.4/doc/tutorials/features2d/trackingmotion/corner_subpixeles/corner_subpixeles.html

[4] Función de OpenCV `KeyPoint`
https://docs.opencv.org/2.4/modules/features2d/doc/common_interfaces_of_feature_detectors.html

[5] Función de OpenCV `SIFT.compute`
https://docs.opencv.org/3.1.0/da/df5/tutorial_py_sift_intro.html

[6] Feature Matching
https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_matcher/py_matcher.html

[7] Feature Matching + Homography
https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_feature_homography/py_feature_homography.html
https://www.learnopencv.com/homography-examples-using-opencv-python-c/

[8] Apuntes de Teoría "*Computer Vision: Feature Matching*"

[9] Apuntes de Teoría Tema 5 "*Computer Vision: Feature Matching*".
Optimal RANSAC - Towards a Repeatable Algorithm for Finding the Optimal Set, by Anders Hast, Johan Nysjö and Andrea Marchetti. Apuntes MIT
http://6.869.csail.mit.edu/fa12/lectures/lecture13ransac/lecture13ransac.pdf
https://es.wikipedia.org/wiki/RANSAC