Doble Grado en Ingeniería Informática y Matemáticas

VISIÓN POR COMPUTADOR
(E. Computación y Sistemas Inteligentes)

TRABAJO-3: Detección de puntos relevantes y Construcción de panoramas

UNIVERSIDAD DE GRANADA

Carlos Santiago Sánchez Muñoz

Grupo de prácticas 2 - Jueves

Email: carlossamu7@correo.ugr.es

20 de noviembre de 2019

Índice

Ejercicio 1

Detección de puntos Harris. Aplicar la detección de puntos Harris sobre una pirámide Gaussiana de la imagen, presentar dichos puntos sobre las imágenes haciendo uso de la función `drawKeyPoints`. Presentar los resultados con las imágenes `Yosemite.rar`. Para ello,

a) Detectar los puntos Harris en cada nivel de la pirámide a partir de la información de la función `cornerEigenValsAndVecs()`. Por cada punto extraemos una estructura KeyPoint :(x,y, escala, orientación). Estimar la escala como blockSize*nivel_piramide y la orientación del parche como la orientación del gradiente en su punto central tras un alisamiento de la imagen con un sigma=4.5. (ver http://matthewalunbrown.com/papers/cvpr05.pdf)

b) Variar los valores de umbral de la función de detección de puntos hasta obtener un conjunto numeroso (>2000) de puntos HARRIS en total que sea representativo a distintas escalas de la imagen. Justificar la elección de los parámetros en relación a la representatividad de los puntos obtenidos.

c) Identificar cuantos puntos se han detectado dentro de cada octava. Para ello mostrar el resultado dibujando los KeyPoints con `drawKeyPoints`. Valorar el resultado.

d) Calcular las coordenadas subpixel (x, y) de cada KeyPoint y mostrar imágenes interpoladas (solo un entorno 10x10 a un zoom 5x) de una selección aleatoria de 3 de ellos en donde se marquen las coordenadas originales y las corregidas.

Vamos a calcular los puntos *Harris*. Para ello tenemos que construir una pirámide gaussiana de la imagen de la cual queremos obtener los puntos Harris. Para cada nivel llamamos a la función `cornerEigenValsAndVecs()` que nos devuelve una tupla $(\lambda_1, \lambda_2, x_1, y_1, x_2, y_2)$ donde λ_1, λ_2 son los autovalores de la imagen, x_1, y_1 los autovectores de λ_1 y x_1, y_1 los autovectores de λ_2.

En primer lugar indicar que el operador de Harris que vamos a usar es el siguiente:

$$f_p = \frac{\lambda_1 \lambda_2}{\lambda_1 + \lambda_2}. \tag{1}$$

Usando los valores propios tenemos para cada píxel este criterio Harris. Ahora es posible que haya muchos puntos *Harris*, así que vamos a quedarnos con aquellos que tengan un valor mínimo para el criterio *Harris*. Para ello definimos un umbral, en mi caso lo he establecido a 10 tal y como indicaba el material proporcionado para esta práctica y me ha dado buenos resultados. El código de la función es:

```
def criterioHarris(eigenVal1, eigenVal2, threshold):
    fp = np.zeros(eigenVal1.shape)

    for i in range(eigenVal1.shape[0]):
        for j in range (eigenVal1.shape[1]):
            if eigenVal1[i][j] == 0 and eigenVal2[i][j] == 0:
                fp[i][j] = 0
            else:
                fp[i][j] = eigenVal1[i][j] * eigenVal2[i][j] / (eigenVal1[i][j]+eigenVal2[i][j])
                if fp[i][j] < threshold:
                    fp[i][j] = 0
    return fp
```

A continuación tenemos que hacer la supresión de no máximos ya implementada en prácticas anteriores. En mi caso la he adaptado para solo 2 canales pues en este ejercicio solo usaremos imágenes en Blanco y Negro. Del mismo modo le he añadido un parámetro `winSize` que es el tamaño de la ventana sobre la cual hacer la supresión.

```
def non_maximum_supression(image, winSize):
    res = np.zeros(image.shape)

    if len(image.shape) == 2:
        for i in range(image.shape[0]):
            for j in range(image.shape[1]):
                max = 0
                for p in range(-int(winSize/2), int(winSize/2)+1):
                    for q in range(-int(winSize/2), int(winSize/2)+1):
                        if i+p>=0 and j+q>=0 and (i+p)<image.shape[0] and (j+q)<image.shape[1]:
                            if max<image[i+p][j+q]:
                                max = image[i+p][j+q]

                if max<=image[i][j]:
                    res[i][j] = image[i][j]
                else:
                    res[i][j] = 0

    return res
```

Hemos creado una función `orientacion` que será útil a continuación. Dado un vector $u = (u_1, u_2)$ lo normaliza y ahora se corresponde con $(\cos\theta, \sin\theta)$. Calculamos el ángulo usando `atan2` (acepta dos parámetros no el cociente). Posteriormente lo pasamos a grados que es como nos lo pedirá `openCV`. Lo que nos devuelve `atan2` está en $(-\pi, \pi)$ por tanto cuando sea negativo le sumamos 2π o 360 grados.

```
def orientacion(u1, u2):
    # Comprobamos que no es el vector nulo
    if(u1==0 and u2==0):
        return 0;
    # Normalizamos el vector
    l2_norm = math.sqrt(u1*u1+u2*u2)
    u1 = u1 / l2_norm
    u2 = u2 / l2_norm
    # Calulamos el ángulo en grados
    theta = math.atan2(u2,u1) * 180 / math.pi
    if theta<0:
        theta += 360
    # Devolvemos en grados
    return theta
```

La siguiente función que he implementado es la que me construye los `keypoints` a través de la matriz *Harris*. Un `keypoints` a parte de tener el punto 2D tiene más parámetros como el tamaño (`_size`) y la orientación (`_angle`) explicada anteriormente.

Para el ángulo creamos en cada nivel de la pirámide las matrices $\left(\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}\right)$ donde I es la imagen. Esto lo hacemos con la `convolution` ya implementada en la Práctica 1 y usando `getDerivKernels()`. La otra opción es primero hacer el cálculo de las matrices derivadas y luego hacer la pirámide pero para mi implementación de código era mejor la otra. El tamaño lo calculamos como `block_size * nivel`. Observemos:

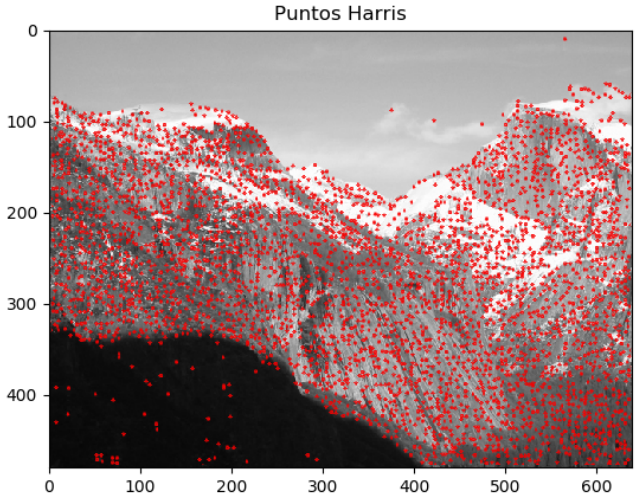
```
def get_keypoints(matrix, block_size, level):
    kp = []
    ksize = 3

    mcopy = np.copy(matrix)
    mcopy = gaussian_blur(mcopy, 4.5)
    kx, ky = cv2.getDerivKernels(1, 0, ksize)
    dx = convolution(mcopy, kx, ky)
    dx = dx.astype(np.float32)

    mcopy = np.copy(matrix)
    mcopy = gaussian_blur(mcopy, 4.5)
    kx, ky = cv2.getDerivKernels(0, 1, ksize)
    dy = convolution(mcopy, kx, ky)
    dy = dy.astype(np.float32)

    for i in range(matrix.shape[0]):
        for j in range(matrix.shape[1]):
            if matrix[i][j]>0:
                kp.append( cv2.KeyPoint( j*(2**level), i*(2**level),
                    _size = block_size*(level+1), _angle = orientacion(dx[i][j], dy[i][j]) ) )

    return kp;
```

El método que gestiona para cada nivel de la pirámide todo lo expuesto anteriormente es `getHarris`.

```
def getHarris(img, block_size, ksize, threshold, level, winSize = 5):
    # Se calculan los autovectoresy autovalores:
    vals_vecs = cv2.cornerEigenValsAndVecs(img, block_size, ksize)

    # En cada píxel tenemos (l1, l2, x1, y1, x2, y2)
    vals_vecs = cv2.split(vals_vecs)
    eigenVal1 = vals_vecs[0]
    eigenVal2 = vals_vecs[1]

    # Criterio de Harris para obtener la matriz con el valor asociado a cada píxel
    harris = criterioHarris(eigenVal1, eigenVal2, threshold)
    # Se suprimen los valores no máximos
    harris = non_maximum_supression(harris, winSize)
    # Obtenemos los keypoints
    return get_keypoints(harris, block_size, level)
```

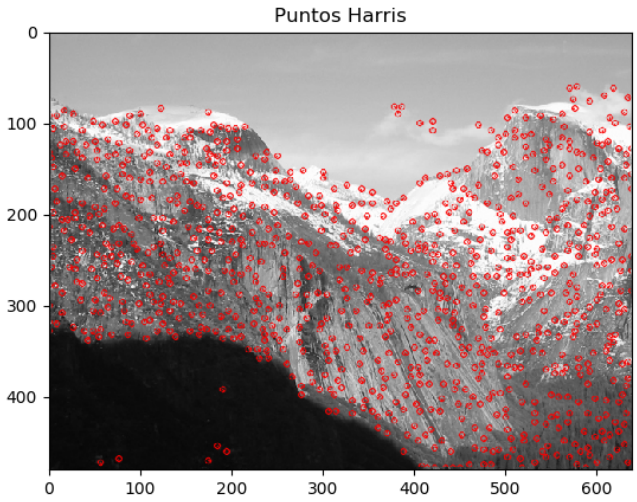
En último lugar la función `ejercicio_1` recibe como parámetro una imagen en escala de grises, calcula la pirámide gaussiana y para cada nivel llama a `getHarris()`. Imprime por pantalla los puntos *Harris* de cada nivel sobre una copia de la imagen original y por último y imprime los puntos *Harris* de todos los niveles sobre la

imagen original. Muestro directamente los resultados obtenidos.



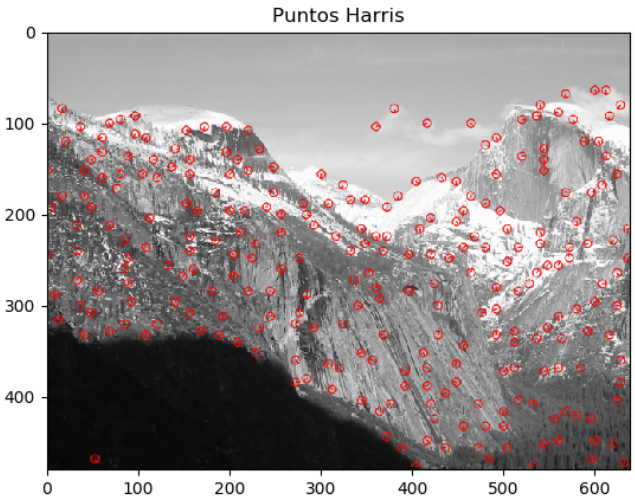
Puntos *Harris* en el nivel 0 sobre la imagen "Yosemite1.jpg"

En el nivel 0 vemos la presencia de abundantes `keypoints` en rojo. En concreto hay 3569 pues muestro por pantalla el número de `keypoints`. El umbral lo ajuste para que el número de estos puntos fuese mayor a 2000 como indica el enunciado.



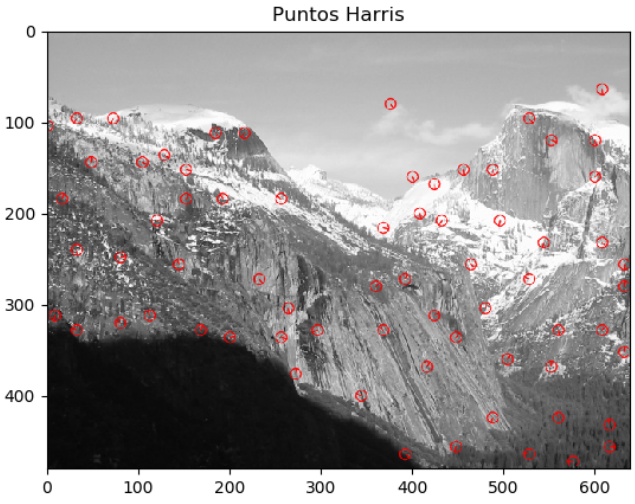
Puntos *Harris* en el nivel 1 sobre la imagen "Yosemite1.jpg"

A partir de aquí podemos observar las orientaciones, que son las rayas que parecen un radio dentro del círculo. En este nivel tenemos 929 `keypoints`. Esto se debe a que la hacer el alisado gaussiano necesario en la pirámide gaussiana estamos suavizando la imagen y especialmente los bordes por lo tanto es lógico que conforme avancemos en los niveles dispondremos de menos puntos *Harris*.



Puntos *Harris* en el nivel 2 sobre la imagen "Yosemite1.jpg"

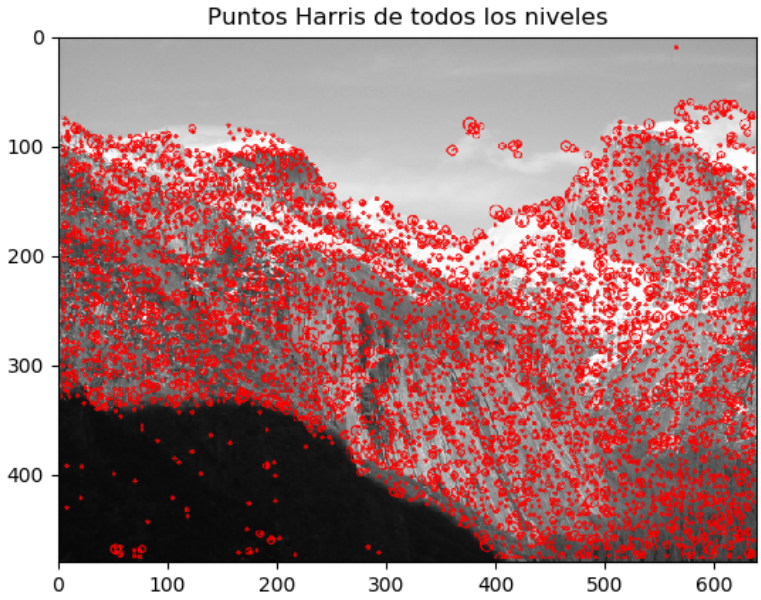
Ya sólo nos falta el último nivel. Destacar que aquí son claras las orientaciones y los puntos de interés han disminuido pero ahora son los mejores, son aquellos donde había más cambio. Hay 262 puntos señalados en la imagen. En la posterior dispondremos de 69.



Puntos *Harris* en el nivel 3 sobre la imagen "Yosemite1.jpg"

Hasta aquí hemos observado que los puntos de interés están en en la frontera de la colina, en los picos de esta, cambios de profundidad pues hay un cambio de tonalidad y sombras e incluso en la nube que hay en la parte superior de la imagen tenemos un punto *Harris* debido al contraste de esta con el suave cielo.

Por último quiero mostrar la unión de los puntos Harris de todos los niveles sobre la imagen original. El resultado queda así:



Todos los puntos *Harris* de "Yosemite1.jpg"

El número total de `keypoints` de la imagen anterior, que es la suma de los `keypoints` de las imágenes de todos los niveles, es 4829.

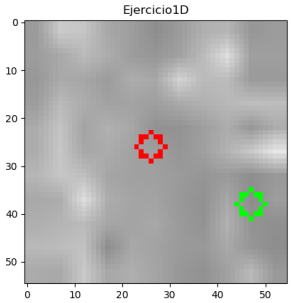
En el apartado d) se nos pide refinar los puntos Harris pues nosotros hacemos la aproximación de que si estamos en el nivel l de la pirámide y en el píxel (i, j) de ella entonces sobre la original es $(i * 2^l, j * 2^l)$. Pero esto no es exacto.

Para ello he implementado la función `refineHarris()` que haciendo uso de `cornerSubPix()` refina los puntos *Harris*.

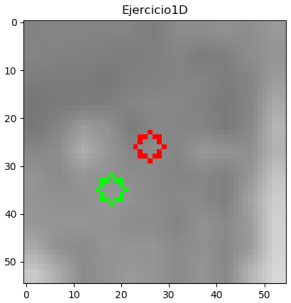
```
def refineHarris(img, points):
    ajustadosAMostrar = []
    listaRes = []

    # Ajustamos los puntos
    p = np.array([punto.pt for punto in points], dtype = np.uint32)
    p_ajustados = p.reshape(len(points), 1, 2).astype(np.float32)
    cv2.cornerSubPix(img, p_ajustados, (5, 5), (-1, -1), (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_M

    # Busco tres puntos ajustados distintos de los originales
    for j in range(0,3):
        ran = random.randint(0, len(p)-1)
        if (p[ran] != p_ajustados[ran][0]).any():
            ajustadosAMostrar.append(ran)

    img = cv2.cvtColor(img, cv2.COLOR_GRAY2RGB).astype(np.float32)

    # Para cada punto ajustado hacer...
```

```
for i in ajustadosAMostrar:
    m_ampliada = np.ndarray(shape=(img.shape[0]+2*5, img.shape[1]+2*5, 3))
    m_ampliada[:, :] = 0
    m_ampliada[5:img.shape[0]+5, 5:img.shape[1]+5] = img.copy()
    col, fil = p[i]
    col_ajustado, fil_ajustado = p_ajustados[i][0]
    res = m_ampliada[fil - 5:fil + 6, col - 5:col + 6]
    res = cv2.resize(res, None, fx = 5, fy = 5) # zoom de x5
    # Señalamos con verde el nuevo y con rojo el antiguo
    res = cv2.circle(res, (int(5*(5+col_ajustado-col)+1),
          int(5*(5+fil_ajustado-fil)+1)), 3, (0, 255, 0))
    res = cv2.circle(res, (5*5+1, 5*5+1), 3, (255, 0, 0))
    listaRes.append(res)
return listaRes
```

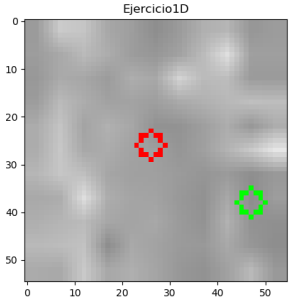
Posteriormente imprimo tres aleatorios de ellos por pantalla haciendo zoom $x5$. Esto sólo lo haremos para un nivel que le indiquemos dentro de la función `ejercicio_1` cambiando la variable `levelChosen` la cual está establecida a 1. Veamos los resultados:



Primer refinamiento de punto *Harris*



Segundo refinamiento de punto *Harris*



Tercer refinamiento de punto *Harris*

En rojo hemos mostrado el antiguo y en verde el que está refinado. Los resultados parecen coherentes.

Ejercicio 2

Detectar y extraer los descriptores AKAZE de OpenCV, usando para ello `detectAndCompute()`. Establecer las correspondencias existentes entre cada dos imágenes usando el objeto BFMatcher de OpenCV y los criterios de correspondencias "BruteForce+crossCheck y "Lowe-Average-2NN". Mostrar ambas imágenes en un mismo canvas y pintar líneas de diferentes colores entre las coordenadas de los puntos en correspondencias. Mostrar en cada caso un máximo de 100 elegidas aleatoriamente.

a) Valorar la calidad de los resultados obtenidos a partir de un par de ejemplos aleatorios de 100 correspondencias. Hacerlo en términos de las correspondencias válidas observadas por inspección ocular y las tendencias de las líneas dibujadas.

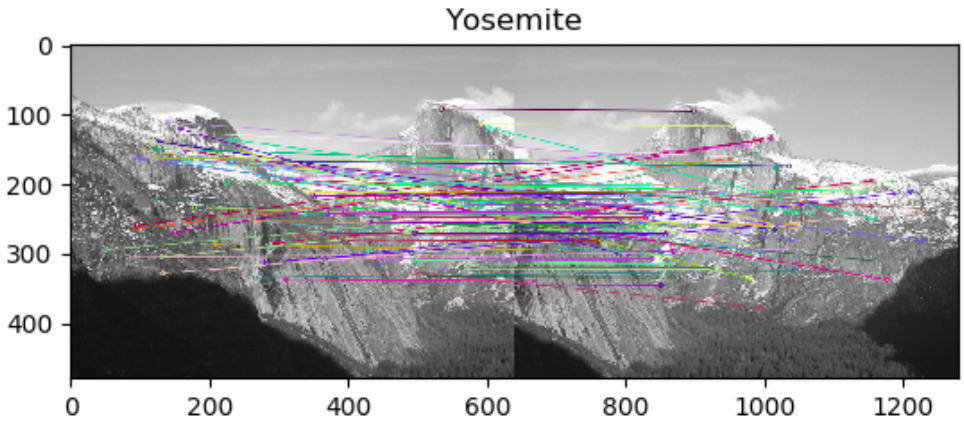
Vamos a usar el detector `AKAZE` de `OpenCV`. A partir de él obtendremos una lista de `keyPoints` y un descriptor por cada una de las dos imágenes. Haremos uso de los criterios de características que nos indica el enunciado.

En la primera parte del ejercicio usamos el criterio de correspondencias *BruteForce + crossCheck*. Para ello aprovechamos la clase `BFMatcher` y cuando la llamemos activaremos el *flag* `crossCheck`.

El criterio de validación cruzada podemos hacerlo de manualmente en vez de activar el *flag*. La otra opción es calcular `matches2to1` y comprobar con `queryIdx` y `trainIdx` los que están tanto en `matches1to2` como en `matches2to1`. Con el descriptor llamamos a `detectAndCompute()`. He llamado a mi función `getMatches_BFCC()`:

```
def getMatches_BFCC(img1, img2, n = 100, flag = 2, flagReturn = 1):
    # Inicializamos el descriptor AKAZE
    detector = cv2.AKAZE_create()
    # Se obtienen los keypoints y los descriptores de las dos imágenes
    keypoints1, descriptor1 = detector.detectAndCompute(img1, None)
    keypoints2, descriptor2 = detector.detectAndCompute(img2, None)

    # Se crea el objeto BFMatcher activando la validación cruzada
    bf = cv2.BFMatcher(crossCheck = True)
    # Se consiguen los puntos con los que hace match
    matches1to2 = bf.match(descriptor1, descriptor2)
    # Se guardan n puntos aleatorios
    matches1to2 = random.sample(matches1to2, n)

    # Imagen con los matches
    img_match = cv2.drawMatches(img1, keypoints1, img2, keypoints2, matches1to2, None, flags = flag)

    # El usuario nos indica si quiere los keypoints y matches o la imagen
    if flagReturn:
        return img_match
    else:
        return keypoints1, keypoints2, matches1to2
```

A continuación se muestra el resultado, que es un *Canvas* con las dos imágenes y los *matches* dibujados con diferentes colores.



Matches de "Yosemite1.jpg" con "Yosemite2.jpg" usando *BruteForce + crossCheck*

Mirando la imagen nos damos cuenta de que hay correspondencias correctas pero también hay otras que no son acertadas porque unen puntos de una imagen que no están en la otra debido a que son bastante similares.

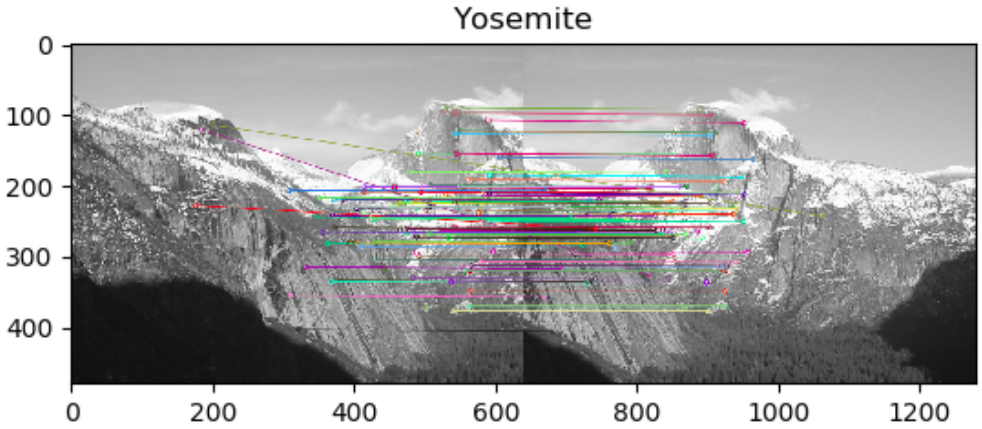
Encontramos en general rectas horizontales pero hay algunas sueltas que están en otras direcciones. Es imposible extraer las correspondencias sin tener un margen de error y el resultado que obtenemos es normal.

```
def getMatches_LA_2NN(img1, img2, n = 100, ratio = 0.8, flag = 2, flagReturn = 1):
    # Inicializamos el descriptor AKAZE
    detector = cv2.AKAZE_create()
    # Se obtienen los keypoints y los descriptores de las dos imágenes
    keypoints1, descriptor1 = detector.detectAndCompute(img1, None)
    keypoints2, descriptor2 = detector.detectAndCompute(img2, None)

    # Se crea el objeto BFMatcher
    bf = cv2.BFMatcher()

    # Escogemos los puntos con los que hace match indicando los vecinos más cercanos para la comprobación (2)
    matches1to2 = bf.knnMatch(descriptor1, descriptor2, 2)

    # Mejora de los matches ->los puntos que cumplan con un radio en concreto
    best1to2 = []
    # Se recorren todos los matches
    for p1, p2 in matches1to2:
        if p1.distance < ratio * p2.distance:
            best1to2.append([p1])

    # Se guardan n puntos aleatorios
    matches1to2 = random.sample(best1to2, n)

    # Imagen con los matches
    img_match = cv2.drawMatchesKnn(img1, keypoints1, img2, keypoints2, matches1to2, None, flags = fla

    # El usuario nos indica si quiere los keypoints y matches o la imagen
    if flagReturn:
        return img_match
    else:
        return keypoints1, keypoints2, matches1to2
```

La función que realiza todo lo pedido en este ejercicio es `ejercicio_2()`. Muestro el código aunque es sencillo:

```
def ejercicio_2(img1, img2, image_title = "Imagen"):
    print("--- EJERCICIO 2 ---")
    img1 = img1.astype(np.uint8)
    img2 = img2.astype(np.uint8)
    match_BF_CC = getMatches_BF_CC(img1, img2)
    pintaI(match_BF_CC, 0, image_title, "Ejercicio 2")
    match_LA_2NN = getMatches_LA_2NN(img1, img2)
    pintaI(match_LA_2NN, 0, image_title, "Ejercicio 2")
    input("Pulsa 'Enter' para continuar\n")
```

Ahora vamos a mostrar la imagen resultado de usar el criterio de correspondencia *Lowe-Average-2NN*.



Matches de "Yosemite1.jpg" con "Yosemite2.jpg" usando *Lowe-Average-2NN*

Con ambos criterios hemos usado las imágenes de *Yosemite* para poder hacer una pequeña comparativa ahora. A pesar de que en ambas el resultado es similar podemos destacar que este segundo criterio es mejor, al menos para este conjunto de imágenes, porque hay pocas líneas con direcciones extrañas y aquí la inmensa mayoría de las que están pintadas son correctas. En la anterior había más rectas con direcciones erróneas que aquí y había demasiadas rectas.

Ejercicio 3

Escribir una función que genere un Mosaico de calidad a partir de N=2 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()`. 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).

Lo primero es programar una función que va a ser clave en este ejercicio y el siguiente que es `getHomography()` a la cual le pasamos dos imágenes como argumento y calcula la homografía entre ellas.

Para ello necesitamos ordenar los `keypoints` de manera que el el primero la primera lista haga *match* con el primero de la segunda lista y así sucesivamente. Posteriormente les pasamos las listas ordenadas a `findHomography()` función de `OpenCV` que nos hace el trabajo duro.

```
def getHomography(img1, img2, flag=1):
    # Obtenemos los keyPoints y matches entre las dos imagenes.
    if(flag):
        kpts1, kpts2, matches = getMatches_LA2NN(img1, img2, flagReturn=0)
    else:
        kpts1, kpts2, matches = getMatches_BFCC(img1, img2, flagReturn=0)
    # Ordeno los puntos para usar findHomography
    puntos_origen = np.float32([kpts1[punto[0].queryIdx].pt for punto in matches]).reshape(-1, 1, 2)
    puntos_destino = np.float32([kpts2[punto[0].trainIdx].pt for punto in matches]).reshape(-1, 1, 2)
    # Llamamos a findHomography
    homografia , _ = cv2.findHomography(puntos_origen, puntos_destino, cv2.RANSAC, 1)
    return homografia
```

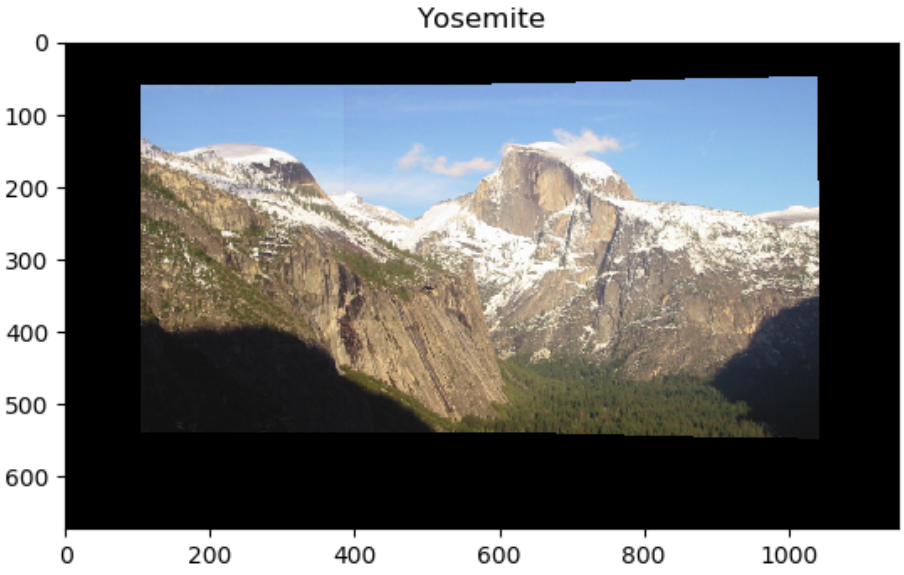
Programo también la función `getMosaic()` a la cual le damos dos imágenes y nos devuelve el mosaico resultante de ellas. Destacar que he establecido el ancho del *Canvas* como la suma de los anchos de las dos imágenes multiplicado por 0,9 (y convirtiendo a entero el resultado). Con la altura he cogido la de la primera imagen y la he multiplicado por 1,4 (convierto a entero) que es un umbral suficiente para que entre todo.

Para que las imágenes estén lo más centradas posible he hecho unas traslaciones en ambos ejes, `tx` y `ty`, de manera que esto ocurra. Lo idóneo es que estás traslaciones no sean un valor concreto sino que estén en términos relativos del tamaño de las imágenes argumento. Por ello ambas traslaciones son el 9 % del ancho y de la altura respectivamente.

Se muestra el código de lo explicado a continuación:

```
def getMosaic(img1, img2):
    homographies = [None, None]                                    # Lista de homografías
    width = int((img1.shape[1]+img2.shape[1]) * 0.9)               # Ancho del mosaico
    height = int(img1.shape[0] * 1.4)                              # Alto del mosaico

    print("El mosaico resultante tiene ñtamao ({}, {})".format(width, height))
    tx = 0.09 * width      # Calculo traslación en x
    ty = 0.09 * height     # Calculo traslación en y

    # Homografía 1
    hom1 = np.array([[1, 0, tx], [0, 1, ty], [0, 0, 1]], dtype=np.float32)
    res = cv2.warpPerspective(img1, hom1, (width, height), borderMode=cv2.BORDER_TRANSPARENT)
    # Homografía 2
    hom2 = getHomography(img2, img1)
    hom2 = np.dot(hom1, hom2)
    res = cv2.warpPerspective(img2, hom2, (width, height), dst=res, borderMode=cv2.BORDER_TRANSPARENT

    return res
```

Si usamos las imágenes “Yosemite1” y “Yosemite2” el resultado es:



Mosaico con las imágenes “mosaico010” y “mosaico011”

Obtenemos lo que esperamos, es correcto el mosaico formado y está bastante centrado gracias al juego que hemos hecho con las traslaciones.

Quiero también ver los resultados de la ejecución de nuestra formación de mosaicos usando las imágenes “mosaico01” proporcionadas en el material de la práctica.

Mosaico con las imágenes “Yosemite1” y “Yosemite2”

Efectivamente el *match* obtenido es correcto y las imágenes se unen formando el mosaico adecuado.

Ejercicio 4

Lo mismo que en el punto anterior pero usando todas las imágenes para el mosaico.

En este ejercicio vamos a adaptar la función `getMosaic()` del anterior para N imágenes. La clave para hacer esto es poner la imagen central en el centro del *Canvas*. Para ello he calculado la homografía que hace esto:

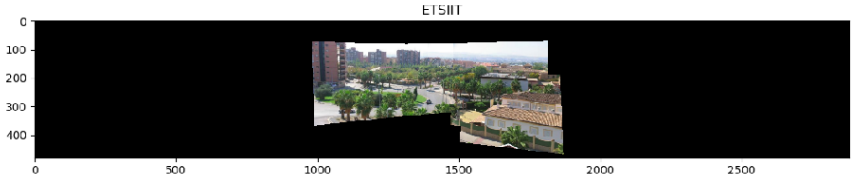
```
def identityHomography(img, mosaicWidth, mosaicHeight):
    tx = mosaicWidth/2 - img.shape[0]/2      # Calculamos traslación en x
    ty = mosaicHeight/2 - img.shape[1]/2     # Calculamos traslación en y
    return np.array([[1, 0, tx], [0, 1, ty], [0, 0, 1]], dtype=np.float32)
```

Ahora programamos una función `getMosaicN()` que recibe como argumento una lista de imágenes. Como hemos dicho calculamos la homografía de la imagen central y luego vamos calculando el resto de homografías desde el centro hacia atrás y desde el centro hacia delante. Veamos:

```
def getMosaicN(list):
    homographies = [None] * len(list)                          # Lista de homografías
    ind_center = int(len(list)/2)                              # Índice de la imagen central
    img_center =  list[ind_center]                             # Imagen central
    width = int(sum([im.shape[1] for im in list]) * 0.9)       # Ancho del mosaico
    height = list[0].shape[0] * 2                              # Alto del mosaico

    print("El mosaico resultante tiene ñtamao ({}, {})".format(width, height))

    # Homografía central
    hom_center = identityHomography(img_center, width, height)
    homographies[ind_center] = hom_center
    res = cv2.warpPerspective(img_center, hom_center, (width, height),
          borderMode=cv2.BORDER_TRANSPARENT)

    # Empezamos por el centro y vamos hacia atrás
    for i in range(0, ind_center)[::-1]:
        h = getHomography(list[i], list[i+1])
        h = np.dot(homographies[i+1], h)
        homographies[i] = h
        res = cv2.warpPerspective(list[i], h, (width, height),
              dst=res, borderMode=cv2.BORDER_TRANSPARENT)

    # Empezamos por el centro y vamos hacia delante
    for i in range(ind_center+1, len(list)):
        h = getHomography(list[i], list[i-1])
        h = np.dot(homographies[i-1], h)
        homographies[i] = h
        res = cv2.warpPerspective(list[i], h, (width, height),
              dst=res, borderMode=cv2.BORDER_TRANSPARENT)

    return res
```

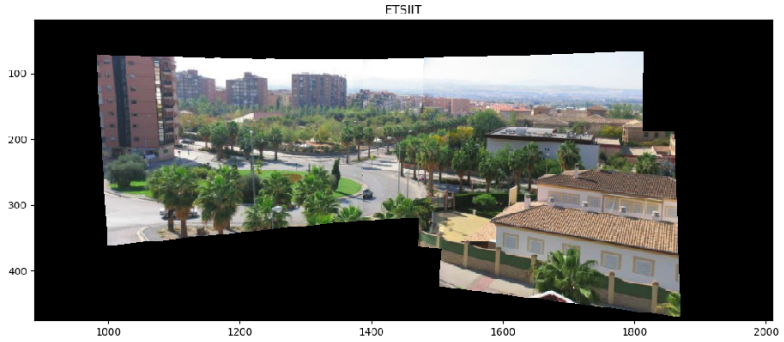
El ancho lo he calculado como la suma de los anchos de la lista de imágenes sin embargo como al unir siempre se pierde algo he multiplicado por 0,9 y luego he convertido el resultado a un número entero. Para la altura he doblado la altura de la primera imagen de la lista.

A falta de observar el resultado, esta es la clave para minimizar el error y que el mosaico quede lo mejor posible en el *Canvas*. Probemos con el conjunto de mosaicos de nuestra facultad:



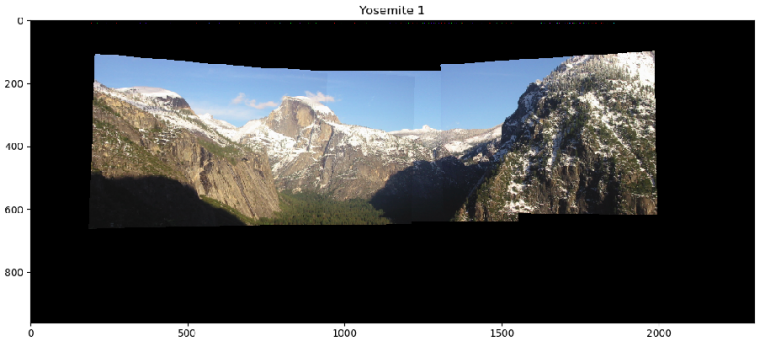
Mosaico con las imágenes "mosaico00" correspondientes a la ETSIIT

El tamaño no es el adecuado así que he hecho zoom en la ventana de `OpenCV` y luego he vuelto a guardar la imagen. Ahora sí es lo que queremos:



Mosaico con las imágenes "mosaico00" correspondientes a la ETSIIT

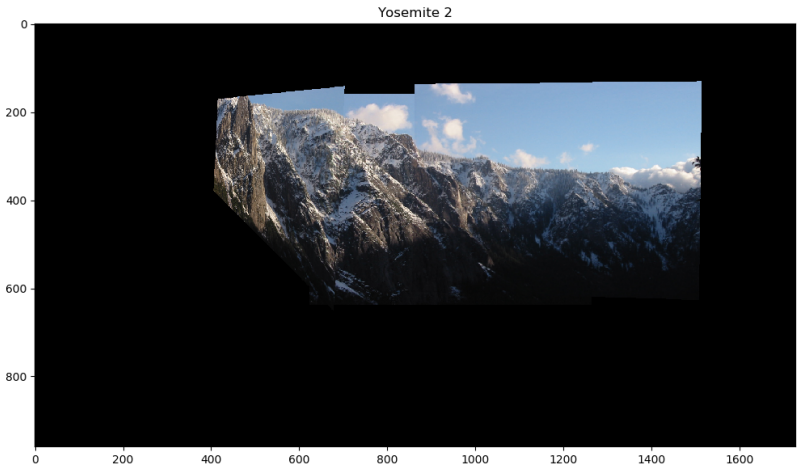
Ya que tengo todo implementado voy a testear los resultados con las imágenes de "yosemite$i$" con $i = 1, ..., 4$ y "yosemite$i$" con $i = 5, ..., 7$.



Mosaico con las 4 primeras imágenes "yosemite"

Vemos que los mosaicos obtenidos son correctos y están bastante bien centrado teniendo en cuenta que el número de imágenes es par y por tanto al asignar al

centro una, uno de los lados queda desfavorecido. Por último el segundo conjunto mencionado.



Mosaico con las 3 últimas imágenes “yosemite”

En este caso el número de imágenes era impar. De nuevo está bien centrado, y las imágenes están bien unidas confirmando que las homografías han sido calculadas correctamente.