Práctica 3 - Visión por computador

Javier Sáez

Nota inicial

En el desarrollo de esta práctica se utilizarán funciones que se desarrollaron en una práctica anterior. No se explicarán por ello en profundidad, aunque se hará un breve comentario sobre el resultado de la aplicación de las mismas.

Detección de puntos de Harris

Lo primero que haremos es hacer la detección de puntos de *Harris*. Situémonos en el contexto.

Los puntos de Harris de una imagen se obtienen a partir de la *Matriz de Harris* de esa imagen I. Esta matriz viene dada por:

$$H(x,y) = \nabla_{\sigma_d} I(x,y) \nabla_{\sigma_d} I(x,y)^T * g_{\sigma_i}(x,y).$$

Localmente,si W es un vecindario de un pixel , esta matriz H se puede aproximar como

$$H = \begin{pmatrix} \sum_{(x,y)\in W} I_x^2 & \sum_{(x,y)\in W} I_x I_y \\ \sum_{(x,y)\in W} I_x I_y & \sum_{(x,y)\in W} I_y^2 \end{pmatrix}$$

En la práctica, utilizaremos la función de *OpenCV*: *cornerEigenValsAndVecs* para obtener directamente en cada punto los valores propios de la matriz H. Una vez tenemos para cada punto de la imagen esta matriz H, la utilizamos para encontrar puntos interesantes en nuestra imagen. Para ello, definimos la función *corner_strenght* de la siguiente forma:

$$f_{HM}(x,y) = \frac{det(H(x,y))}{tr(H(x,y))} = \frac{\lambda_1 \lambda_2}{\lambda_1 + \lambda_2}$$

tenemos que tener cuidado con los puntos en los que $\lambda_1 + \lambda_2 = 0$. En estos casos, aplicaremos que $f_{HM} = 0$. El código que utilizaremos para ella será el siguiente:

```
def corner_strength(l1,l2):
    if (l1+l2 == 0):
        return 0
    return (l1*l2)/(l1+l2)
```

Tomaremos además aquellos puntos cuyo valor de la función *corner_strenght* sea mayor que un **umbral(threshold)**. Por último, haremos una supresión de no máximos (usando la función de la práctica anterior) para eliminar los puntos que no sean máximos de su entorno. Los puntos que obtengamos serán nuestros **puntos de Harris**.

Estos puntos serán **KeyPoints** de *OpenCV*, y estarán formados por:

- Coordenadas *x* e *y* , relativas a la imagen inicial (comentaremos esto más adelante)
- Escala relativa a la imagen actual
- Orientación.

Es importante denorar cómo calculamos la **orientación** de nuestros puntos de *Harris*. Lo haremos siguiendo las indicaciones que se proporcionan en Brown-Szeliski-Winder:

- Tomamos *dx,dy* las derivadas de la imagen
- Aplicamos a cada derivada un *Gaussiano* con $\sigma = 4.5$
- Si tenemos el pixel i, j, tomamos *ox* el valor del pixel *ij* en la derivada (a la que se le aplicó el gaussiano) en *X*, y hacemos lo propio con *oy*.
- Normalizamos estos valores, para que sean el seno y el coseno de un ángulo
- Obtenemos la orientación aplicando la arcotangente.

Los **KeyPoints** los obtendremos sobre cada nivel de una **pirámide gaussiana**. En cada nivel de la pirámide, la escala de estos puntos será proporcional al número de nivel. Con estos datos, obtenemos los **KeyPoints** del siguiente modo:

```
def get_keypoints(img,block_size,level):
    dx = maskDerivKernels(img,1,0)
    dy = maskDerivKernels(img,0,1)

    dx = gaussian2D(dx,4.5)
    dy = gaussian2D(dy,4.5)
    keypoints = []
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            if(img[i,j] > 0):
                ox = dx[i,j]
                oy = dy[i,j]
                cos,sen = ox/(math.sqrt(ox**2 + oy**2)),oy/(math.sqrt(ox**2 + oy**2))
                ori = math.atan2(sen/cos)*180/math.pi
                keypoints.append(cv2.KeyPoint(j*(2**level),
                                              i*(2**level),
                                              _size = block_size*(level+1),
                                              _angle = ori))
    return keypoints
```

level es el nivel de la pirámide gaussiana donde nos encontramos.

En resumen, el procedimiento que realizamos para obtener los **KeyPoints** en una imagen concreta es:

1. Obtener en cada pixel los valores propios λ_1, λ_2 de la matriz $H(x, y)$.
2. Obtener en cada pixel su *corner_strength*
3. Tomar los píxeles cuyo *corner_strength* supere un umbral marcado *threshold*
4. Suprimir los no máximos locales de la matriz resultante
5. Crear los *KeyPoints* con los puntos restantes

Una vez hemos dado todas las herramientas, podemos exponer la función que hemos hecho para extraer los puntos de *Harris* de una imagen. Esta recibirá como parámetros:

- La imagen en cuestión
- El nivel que ocupa esta imagen en la pirámide gaussiana
- *block_size*, el tamaño del vecindario *W*
- *ksize*, el tamaño del vecindario para el operador de *Sobel*
- *threshold*, o umbral que tendrán que superar los puntos para ser significativos

El código de la función es el siguiente:

```
def harris(src,level,block_size = 3,ksize = 3,threshold = 10):
    #Get lambda1,lambda2, eiv11,eiv12, eiv21,eiv22
    e_v = cv2.cornerEigenValsAndVecs(src,blockSize = block_size,ksize = ksize)
    #Corner strength matrix
    first_m = np.asarray([[ corner_strength(e_v[i,j,0],e_v[i,j,1])
```

```
            for j in range(src.shape[1])]
            for i in range(src.shape[0])])
    # Get values that are > than threshold
    threshold_m = np.asarray([[ first_m[i,j] if first_m[i,j] > threshold else 0
                    for j in range(first_m.shape[1])]
                    for i in range(first_m.shape[0])])

    # Supress no max in winsize X winsize neighborhood
    sup_no_max_m  = supresionNoMax(threshold_m,5)

    # Return keypoints
    return get_keypoints(sup_no_max_m,block_size,level)
```

Ahora, como hemos comentado, tenemos que aplicar esta función sobre cada uno de los niveles de una pirámide *gaussiana*, y ver cómo afectan tanto los alisamientos como los cambios de tamaño a cada nivel de la pirámide. Como podemos observar, cuando se crean los **KeyPoints** se multiplican sus coordenadas x, y por $2^{level+1}$. Esto es debido a que queremos dibujar los puntos sobre la imagen inicial, pero los estamos hallando sobre una matriz de tamaño menor (un nivel siguiente de la pirámide gaussiana), así que para poder dibujarlos sobre la original tenemos que reescalar esos puntos a la imagen original. Además, dibujaremos en cada paso los puntos obtenidos sobre una imagen inicial que acumulará los puntos de todos los niveles, para observar qué puntos se quedan en cada nivel. Los puntos los dibujaremos con la función de *OpenCV*: *cv2.drawKeyPoints*. El código que utilizamos para el primer ejercicio es el siguiente:

```
# Get gaussian pyramid
p1 = gaussianPyramid(y1c,iters = 3)

all_keypoints = np.copy(y1c).astype(np.uint8)
total_kp = 0

for i in range(len(p1)):
    colorless = cv2.cvtColor(p1[i],cv2.COLOR_RGB2GRAY).astype(np.float32)
    kp = harris(colorless,i,block_size = 7,ksize = 3,threshold = 10)
    print("Found: " + str(len(kp)) + " keypoints at level " + str(i))
    total_kp += len(kp)
    # Draw circles
    copy = np.copy(y1c).astype(np.uint8)
    copy = cv2.drawKeypoints(copy,kp,np.array([]),
        flags = cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS).astype(np.float64)
    all_keypoints = cv2.drawKeypoints(all_keypoints,kp,np.array([]),
        flags = cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
    pintaI(copy)

pintaI(all_keypoints.astype(np.float64))
print("Total keypoints: " + str(total_kp))
```

Vamos a aplicar el algoritmo sobre una imagen. En concreto, utilizaremos esta:

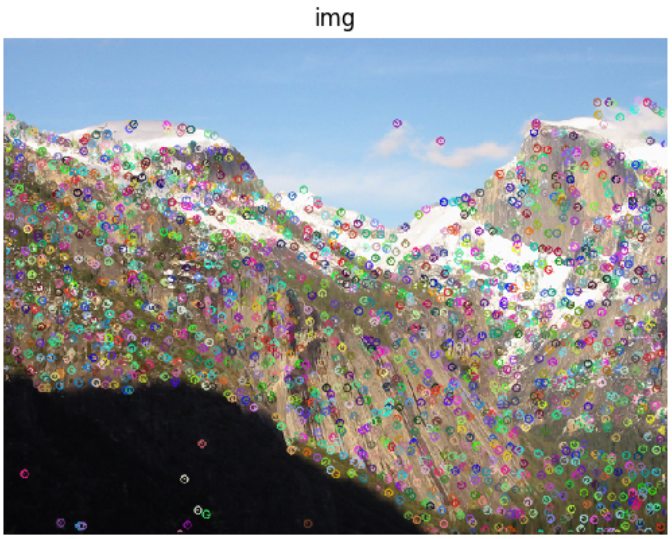
Al ejecutar el algoritmo, a parte de las imágenes, obtenemos el siguiente resultado:

```
[fjsaezm@fjsaezm P3]$ python3.6 P3.py
Found: 1740 keypoints at level 0
Found: 449 keypoints at level 1
Found: 131 keypoints at level 2
Found: 33 keypoints at level 3
Found: 11 keypoints at level 4
Total keypoints: 2364
```

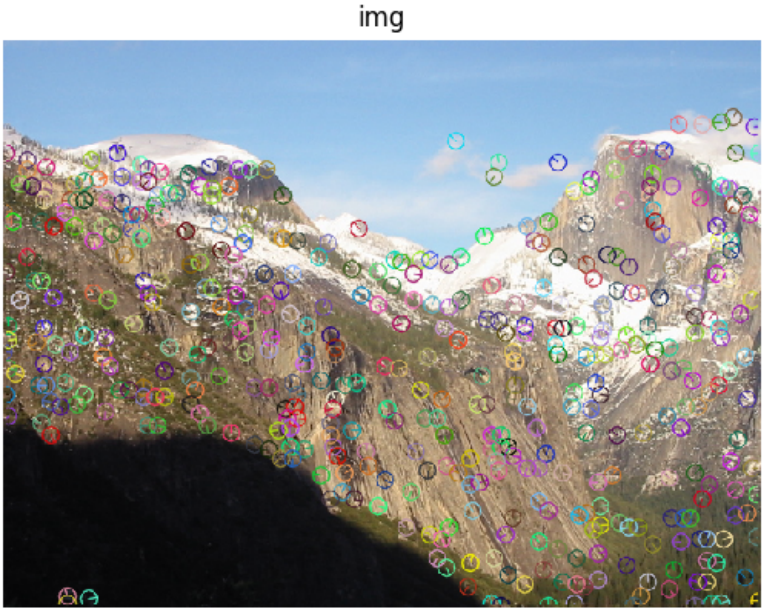
Este ha sido ejecutado con los parámetros:

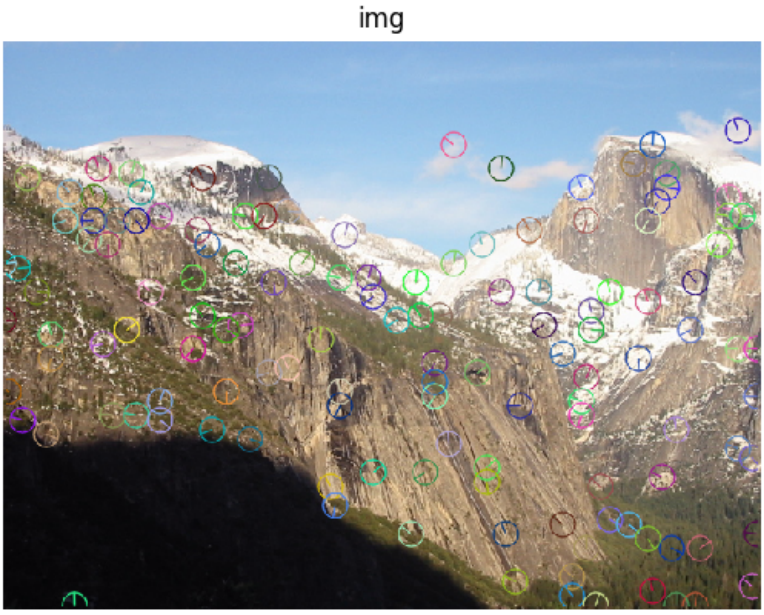
- $block_size = 7$
- $ksize = 3$
- $winsize = 5$
- $threshold = 10$

Veamos la ejecución del algoritmo sobre la pirámide gaussiana de la imagen. Mostraremos nivel por nivel.

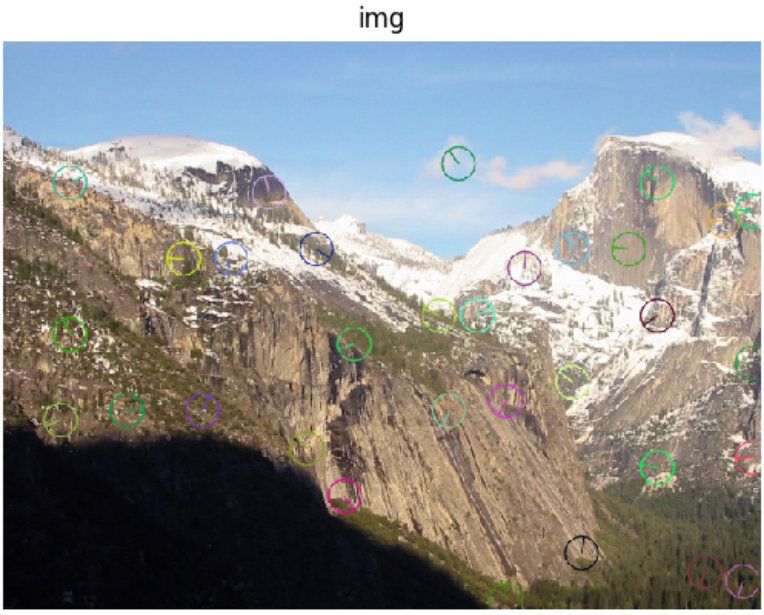


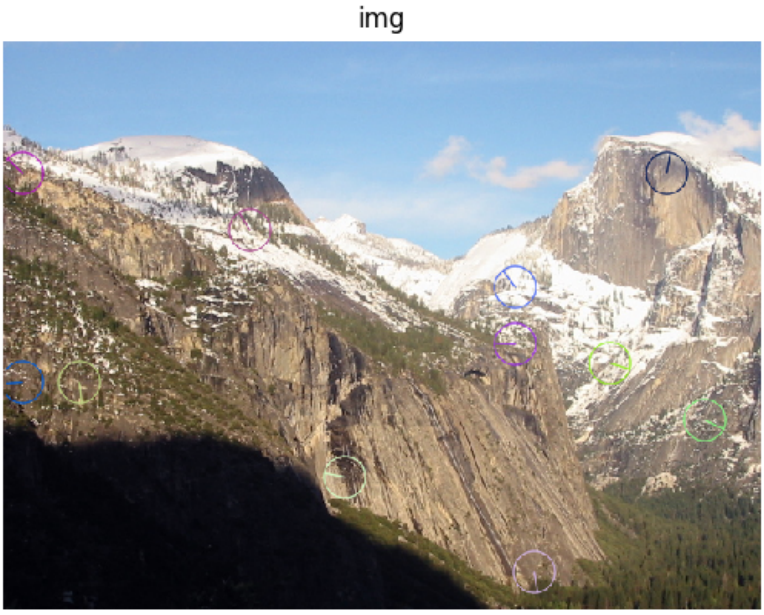
Estos son los puntos de Harris que obtenemos en la imagen original. Vemos como la mayoría de ellos están sobre la zona montañosa, que es donde observamos cambios de colores y formas más grandes, por lo que los gradientes serán más grandes en los entornos de esos puntos y por tanto tenemos más puntos de *Harris*. Veamos los siguientes niveles





En el primer nivel de la pirámide gaussiana, los puntos se reducen notablemente, podemos ver que pasamos de tener 1740 en el primer nivel a 449 en el segundo. El alisamiento que se ha producido en la imagen hace que los gradientes en la misma desciendan en los vecindarios. Además, el borrado y filas de columnas implica que la imagen es más pequeña y la supresión de no máximos afecta algo más a los vecindarios.





En estos dos últimos niveles, notamos mucho el descenso de los *puntos de harris* respecto del nivel inicial, quedándonos solo 33 y 11 puntos respectivamente. Esto tiene sentido, pues en el último nivel la imagen tendrá un tamaño muy pequeño y un tamaño de vecindario bastante amplio, con lo que la supresión de no maimos afectará mucho a la imagen, quedándose solo con los **KeyPoints** más relevantes de la misma, que serán los que tienen f_{HM} más alta.

Hay que notar que hemos tenido que usar un valor de $Threshold = 10$ para tomar que el total de puntos sea > 2000, como se pedía.

Correspondencias entre imágenes

Una vez que hemos obtenido una serie de **KeyPoints** sobre una imagen, podemos usar estos podemos usarlos para ver si están relacionados con keypoints de **otra** imagen. Esto nos permitiría encontrar partes que tengan relación en varias imágenes.

OpenCV tiene implementados los *descriptores AKAZE*, que nos proporcionarán tanto los *KeyPoints* como un extractor de **descriptores**.

Dados los *KeyPoints*, utilizaremos dos formas diferentes de encontrar las parejas de puntos que están relacionados entre las dos imágenes.

- **Brute Force + Crosscheck**. En este modo, por cada descriptor se encontrará el más cercano en el segundo conjunto de descriptores probando uno por uno. Además, al utilizar *crosscheck*, aumentaremos la consistencia , produciendo mejores resultados. El código es el siguiente

```
def matchBruteForce(im1,im2):
    # Create Akaze descriptor
    akaze = cv2.AKAZE_create()
    # Get Akaze Descriptors and KP
    kp1,d1 = akaze.detectAndCompute(im1,None)
    kp2,d2 = akaze.detectAndCompute(im2,None)
```

```
    # Create BF MAtcher Object
    bfmatcher = cv2.BFMatcher.create(crossCheck = True)
    # Match
    matches = bfmatcher.match(d1,d2)

    return kp1,kp2,d1,d2,matches
```

- **Lowe-Average 2NN**, que usará también un *matcher* de fuerza bruta, pero tomará ahora los 2 vecinos más cercanos. Luego, dado un *ratio* y tomando pares de *matches*, se quedará con un número de **matches válidos**. En concreto, dado un par de matches m, n, si la distancia de m es menor que la distancia de n por el ratio, entonces tomará m como válido. El código es el siguiente:

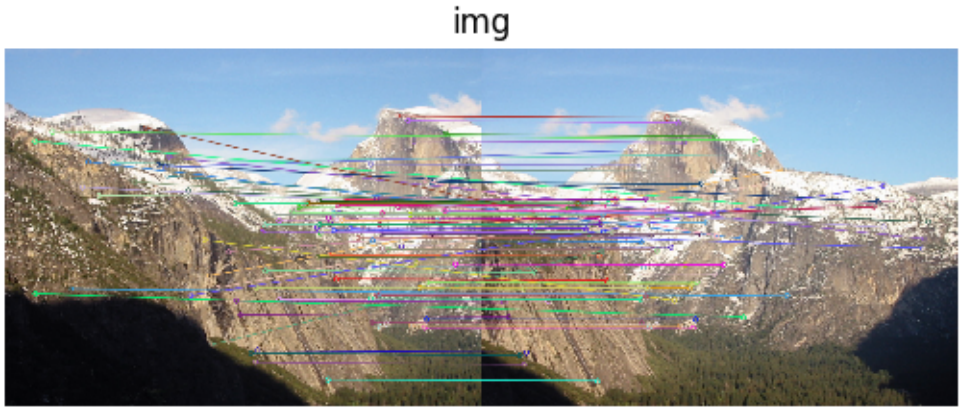
```
def matchLoweAvg2NN(im1,im2,ratio = 0.7):
    # Create Akaze descriptor
    akaze = cv2.AKAZE_create()
    # Get Akaze Descriptors and KP
    kp1,d1 = akaze.detectAndCompute(im1,None)
    kp2,d2 = akaze.detectAndCompute(im2,None)
    # Create BF Matcher
    bfmatcher = cv2.BFMatcher.create()
    # Match using 2-nn
    matches = bfmatcher.knnMatch(d1,d2,k=2)
    # Get non ambiguous matches
    valid = []
    for m,n in matches:
        if m.distance < n.distance*ratio:
            valid.append([m])

    return kp1,kp2,d1,d2,valid
```

Una vez tenemos las dos maneras de encontrar *parejas*, nos interesa tomar una *muestra* de **KeyPoints** en la imagen, pues sabemos que hay muchos. En este caso, el tamaño de la muestra será $n = 100$. Como primera imagen, utilizaremos la imagen anterior. Trataremos de buscar los *matches* con la imagen siguiente:

Vemos que esta imagen es otra fotografía de la misma montaña pero un poco desplazada, por lo que debería de ser sencillo encontrar *matches* entre estas dos imágenes. El código que usamos para esta función es el siguiente:

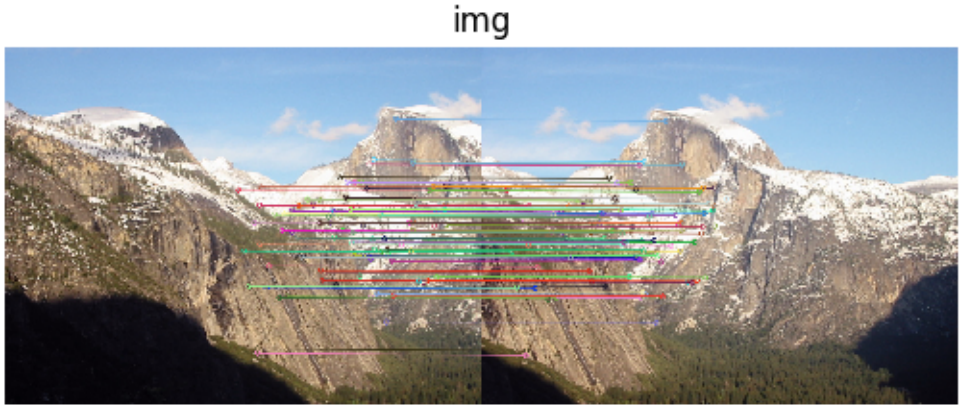
```python
def ej2():
    # Vars
    n = 100
    im1 = y1c.astype(np.uint8)
    im2 = y2c.astype(np.uint8)
    # Brute Force Maches
    k1,k2,d1,d2,matches = matchBruteForce(im1,im2)
    # Get sample size = n
    sample = random.sample(matches,n)
    img = cv2.drawMatches(im1,k1,im2,k2,sample,None,flags = 2)
    pintaI(img.astype(np.float64))

    # LoweAvg2NN matches
    k1,k2,d1,d2,matches = matchLoweAvg2NN(im1,im2)
    # Get sample size = n
    sample = random.sample(matches,n)
    # Draw matches
    img = cv2.drawMatchesKnn(im1,k1,im2,k2,sample,None,flags = 2)
    pintaI(img.astype(np.float64))
```

Vamos a ver primero el resultado de **Brute Force + Crosscheck**. La ejecución nos aporta esta imagen:



En la imagen podemos apreciar que, a pesar de que tenemos muchas parejas de puntos que efectivamente representan el mismo punto en la **imagen real**, existen algunas parejas que no se identifican con los *matches* reales, si vemos por ejemplo la línea violeta que va desde la montaña izquierda y desciende en altura hasta la imagen derecha. Detectamos por tanto que este algoritmo de **Brute Force** puede dar errores, debido posiblemente a quedarse directamente con el descriptor más cercano encontrado.

Vamos a ver qué resultados nos da **Lowe Average 2NN**:



En este caso, podemos ver que todas las líneas van de forma totalmente paralela al eje de abcisas y conectan posiblemente con el punto que representa al mismo en ambas imágenes. Es por ello que podemos afirmar que este método nos da mejores resultados en cuanto a emparejamiento de *KeyPoints* entre las imágenes.

Mosaicos

Vamos ahora a darle un uso a estos emparejamientos de **KeyPoints**. Vamos a crear a partir de varias imágenes una única imagen, uniéndolas a partir de una transformación que se llevará los **KeyPoints** de una imagen en la otra. Esta aplicación se llama **Homografía**, que no es más que un isomorfismo entre dos espacios proyectivos de la misma dimensión. Utilizaremos la expresión matricial de la homografía para operar con la posición de las imágenes.

Para ello, lo que haremos es primero crear un mosaico grande en negro y crear una homografía identidad que nos coloque la primera imagen en el centro de este mosaico. Esto lo haremos con las siguientes funciones:

```
def getCanvas(imgs):
```

```
    return np.zeros((sum([img.shape[0] for img in imgs])*2,
                     sum([img.shape[1] for img in imgs])*2)).astype(np.uint8)

def identity_h(img,canvas):
    tx = canvas.shape[1]/2 - img.shape[1]/2
    ty = canvas.shape[0]/2 - img.shape[0]/2
    id = np.array([[1,0,tx],[0,1,ty],[0,0,1]],dtype = np.float32)
    return id
```

Podemos ver que las homografías siempre serán matrices 3×3, y en este caso buscarán el centro del canvas (mosaico).

Mosaicos de dos imágenes

Ahora, puesto que **Lowe Average 2NN** nos ha dado mejores resultados al emparejar los *KeyPoints*, lo utilizaremos a la hora de emparejar los puntos para obtener la homografía. Los obtendremos primero, y luego utilizaremos **RANSAC** como parámetro en la función *cv2.findHomography* que nos devolverá la homografía que tenemos que aplicar a una imagen para que nos lleve una en la otra. Esto nos lo hará la siguiente función:

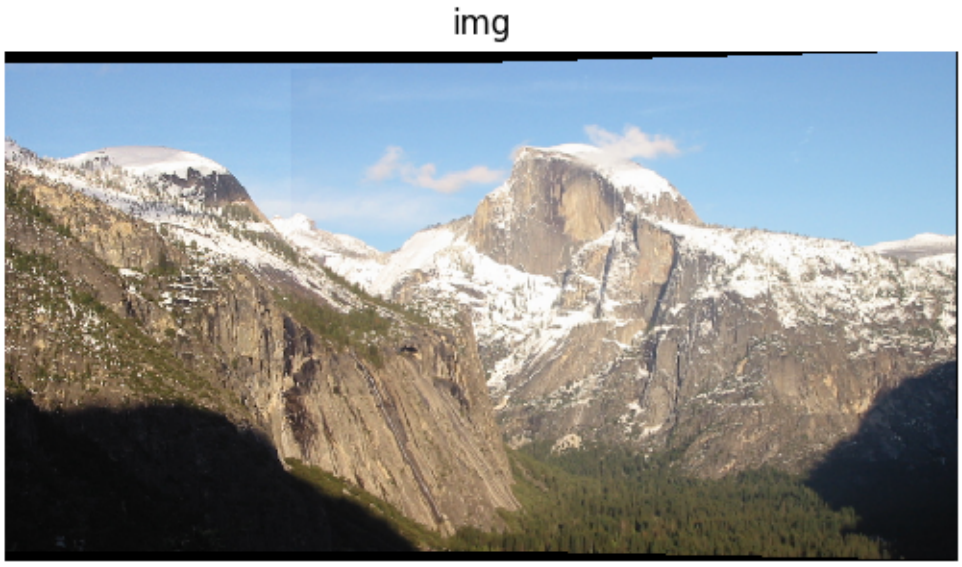
```
def homography(img1,img2):
    # Get matches
    k1,k2,d1,d2,matches = matchLoweAvg2NN(img1,img2)
    # Get and sort matching points
    orig = np.float32([k1[p[0].queryIdx].pt for p in matches]).reshape(-1,1,2)
    dest = np.float32([k2[p[0].trainIdx].pt for p in matches]).reshape(-1,1,2)
    # Get Homography. Using RANSAC
    h = cv2.findHomography(orig,dest,cv2.RANSAC,1)[0]

    return h
```

Una vez que tenemos esta homografía,queremos colocar la nueva imagen dentro de nuestro mosaico. Otra homografía ya había sido aplicada a la imagen inicial para colocarla en el centro, así que debemos componer las homografías para introducir la nueva imagen. Como tenemos la expresión matricial de la homografía, basta realizar el producto de las matrices para obtener la expresión de la homografía composición. Utilizamos la siguiente función para obtener un *canvas* con las dos imágenes en un mosaico.

```
def two_mosaic(img1,img2):
    canvas = getCanvas([img1,img2])
    h = homography(img2,img1)
    id = identity_h(img1,canvas)
    # Introduce img1 in canvas
    canvas = cv2.warpPerspective(img1,id,(canvas.shape[1],canvas.shape[0]),dst = canvas,borderMode =
    comp = np.dot(id,h)
    canvas = cv2.warpPerspective(img2,comp,(canvas.shape[1],canvas.shape[0]),dst = canvas,borderMode
    return canvas
```

La función *cv2.warpPerspective* nos aplica la homografía a la imagen que le pasemos como primer parámetro,y nos la devuelve donde le indiquemos con *dst*.

Vamos a ver el resultado de la ejecución de esta función. Al resultado, le aplicamos una función (*remove_extra*) que se encarga de eliminar toda la parte del canvas que está en negro, para que la visualización sea más sencilla.



Se puede observar que la creación del mosaico es bastante satisfactoria, creando una sensación de ser una única foto gracias a la continuidad que se consigue al hacer la homografía con las dos imágenes.

Mosaicos de n imágenes

En la última parte de la práctica, tratamos de generalizar(sin mucho éxito) el caso anterior para crear mosaicos con un número indefinido de imágenes.

Para ello, el procedimiento que se utiliza es el siguiente:

- Tomar la imagen central de las que se tienen para hacer el mosaico.
- Ir tomando imágenes a la izquierda, y realizando el procedimiento que hemos hecho para $n = 2$.
- En las homografías de cada una, hay que componer la homografía encontrada con la composición de las homografías anteriores, para que la nueva imagen a insertar por la izquierda aparezca en la posición deseada. Se hace lo propio para las imágenes que se insertan por la derecha.

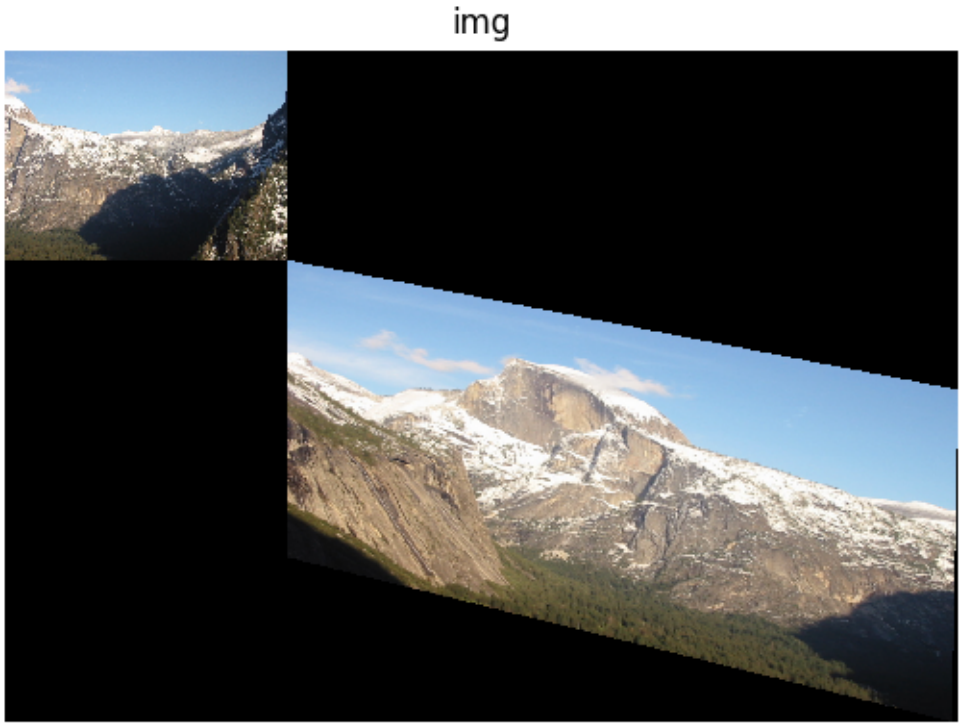
Se podría empezar de izquierda a derecha o de derecha a izquierda si se quisiese, pero al empezar desde el centro se **disminuye** el error que se propaga cuando las homografías se aplican sobre las imágenes y las unen.

El resultado que se ha obtenido no es el deseado, pues la concatenación de las imágenes no ha salido como esperaba y no he logrado detectar el error a tiempo. El código de la función es el siguiente:

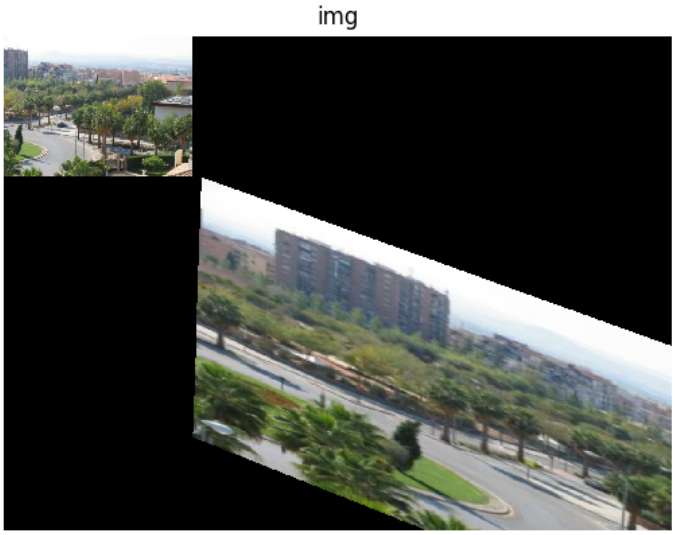
```
def n_mosaic(imgs):
    # Get starting image and big canvas
    half = int(len(imgs)/2)
```

```
    canvas = getCanvas(imgs)
    # Get id homography for half image and warp it
    id = identity_h(imgs[half],canvas)
    canvas = cv2.warpPerspective(imgs[half],id,(canvas.shape[1],canvas.shape[0]),dst = canvas,borderM
    # Create vector for homographies and composition
    homs = [None]*len(imgs)
    homs[half] = id
    # Left Part
    for i in range(half)[::-1]:
        h_i = homography(imgs[i+1],imgs[i])
        #h_i = np.dot(homs[i+1],h_i)
        h_i = np.dot(h_i,homs[i+1])
        homs[i] = h_i
        canvas = cv2.warpPerspective(imgs[i],h_i,(canvas.shape[1],canvas.shape[0]),dst = canvas,borde
    # Right Part
    for i in range(half,len(imgs)):
       h_i = homography(imgs[i],imgs[i-1])
       h_i = np.dot(h_i,homs[i-1])
       #h_i = np.dot(homs[i-1],h_i)
       homs[i] = h_i
       canvas = cv2.warpPerspective(imgs[i],h_i,(canvas.shape[1],canvas.shape[0]),dst = canvas,border

    return canvas
```

Y el resultado de la ejecución con **Yosemite** ha sido:



Y, si tratamos de ejecutarlo con la imagen de la escuela, el resultado es también muy malo.



En ambos casos podemos ver que algunas de las partes de la imagen conectan, pero las partes derecha e izquierda de la imagen no conectan bien.

Conclusión

Hemos aprendido a detectar puntos de *Harris*, que tratamos como **KeyPoints** durante la práctica y nos ayudan a obtener partes importantes de la imagen y son útiles para detectar relaciones entre imágenes y poder así detectar objetos iguales en imágenes o crear mosaicos mediante la aplicación de *homografías* entre los puntos clave.