Visión por Computador

Trabajo 2

Curso 2017/2018

Francisco Javier Caracuel Beltrán

caracuel@correo.ugr.es

4º - Grado en Ingeniería Informática – CCIA – ETSIIT

Índice

0. Consideraciones previas.

Para el desarrollo de este segundo trabajo se han utilizado las siguientes funciones del trabajo 1:

- set_c_map(imgs, cmap): asigna un esquema de color a una imagen o un conjunto de imágenes.

- show_images(imgs, names, cols, title, gray): muestra por pantalla en una misma ventana las imágenes que reciba, añadiendo su título correspondiente.

- convolution_b(img, simga, mask, border): dada una imagen, realiza una convolución con máscara gaussiana.

- convolution_c(img, kernel_x, kernel_y, sigma, border, normalize, own): dada una imagen y dos núcleos (o uno), realiza una convolución de la imagen con dichos núcleos.

- convolution_d(img, kernel_x, kernel_y, sigma, border, normalize, own, dx, dy): dada una imagen, realiza una convolución de primera derivada de tamaño ksize.

- generate_gaussian_pyr_imgs(img, n, sigma, sigma_down, border, resize): dada una imagen, genera una pirámide gaussiana de nivel *n*.

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 de Yosemite.rar.

a) Escribir una función que extraiga la lista potencial de puntos Harris a distintas escalas de una imagen de nivel de gris. Para ellos construiremos una Pirámide Gaussiana usando escalas definidas por sigma = 1,2,3,4,5. Sobre cada nivel de la pirámide usar la función de OpenCV cornerEigenValsAndVec 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 sigmaI=1.5 y sigmaD=1 respectivamente). Usar uno de los criterios de selección estudiados a partir de los autovalores y crear una matriz con el valor del criterio de 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. Ordenar de mayor a menor los puntos resultantes de acuerdo a su 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.

Para el desarrollo de este punto se han creado las siguientes funciones:

- get_block_size(): devuelve el tamaño equivalente al uso de una máscara gaussiana de sigma=1.5, en este caso 10 (6*sigma+1).

- get_ksize(): devuelve el tamaño equivalente al uso de una máscara gaussiana de sigma=1, en este caso 7 (6*sigma+1).

- selection_criteria_harris(λ1, λ2, k = 0.04): devuelve el resultado de un punto al aplicar el operador Harris utilizando los autovalores de M (λ1, λ2) y una constante k con valor 0.04.
  La fórmula utilizada para la operación es: $\lambda_1 * \lambda_2 - k * (\lambda_1 + \lambda_2)^2$.

- is_center_local_max(data): recibe una matriz de datos y devuelve True si el centro es el valor máximo de todos los elementos, False en caso contrario.

- not_max_supression_harris(points, threshold, env): suprime los valores no-máximos de una matriz (points). Elimina los puntos que, aunque tengan un valor alto de criterio Harris, no son máximos locales de un entorno dado por "env". Para mejor los resultados se permite la utilización de un umbral (threshold), aunque en este caso no ha sido necesario aplicarlo.

- get_harris(img, sigma_block_size, sigma_ksize, k, threshold, env, scale): obtiene una lista potencial de los puntos Harris de una imagen (img). Los valores de los parámetros utilizados dependen del sigma que se recibe.

- get_best_harris(points, n): recibe una lista con los puntos Harris obtenidos y devuelve los *n* puntos con mayor valor.

- show_circles(img, points, radius, orientations, color1, color2): dada una imagen, pinta sobre ella los puntos que se encuentran en *points*. Para el apartado *c* se puede especificar *orientations* como *True*. Con *color1* y *color2* se le indica los colores de los puntos y de las líneas de las orientaciones.

El objetivo de este apartado es, partiendo de una imagen (yosemite1.jpg), mostrar aquellos puntos característicos de dicha imagen.

Antes de comenzar con el procesamiento, se ha utilizado la función creada en el trabajo 1 *generate_gaussian_pyr_imgs()*, enviándole la imagen yosemite1.jpg y obteniendo una lista con las distintas imágenes que conforman la pirámide Gaussiana. Se ha modificado esta función para que todas las imágenes sean del tamaño de la original, teniendo el filtro Gaussiano aplicado. Permitir que todas las imágenes sean del tamaño original facilita el tratamiento de los puntos, ya que no es necesario recalcular cuáles son las coordenadas correspondientes en el momento de pintar los círculos sobre la imagen original.

Cuando ya se disponen de las distintas imágenes de la pirámide, se debe calcular la lista potencial de puntos Harris, guardando todos los puntos de todos los niveles con sus valores correspondientes en un mismo contenedor.

Se utiliza la estructura de datos KeyPoint de OpenCV para guardar la información de cada punto y, además, se guarda junto con cada KeyPoint su valor Harris correspondiente.

Para obtener la lista de puntos Harris de una imagen (se aplica a cada imagen de cada nivel de la pirámide) se utiliza la función de OpenCV *cornerEigenValsAndVecs()*, a la que hay que enviar la imagen sobre la que se quieren obtener los puntos, un tamaño de bloque (obtenido con la función creada *get_block_size()* que devuelve el valor 10) y el tamaño de apertura para el operador Sobel (obtenido con la función creada *get_ksize()* que devuelve el valor 7). Con esta función se obtiene una matriz de 3 dimensiones, donde las filas y columnas coinciden en tamaño con las de la imagen y en cada elemento se encuentran 6 valores (autovalores no ordenados de M, autovalores correspondientes a λ_1 y autovalores correspondientes a λ_2). Lo que interesa para calcular los puntos son los autovalores no ordenados de M, que son los requeridos por la función *selection_criteria_harris()*. Esta función tras aplicar la fórmula $\lambda_1 * \lambda_2 - k * (\lambda_1 + \lambda_2)^2$ nos devuelve una lista con unos valores pertenecientes a cada punto.

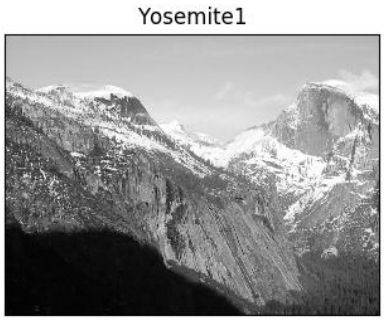
Llegados a esta situación, se deben eliminar todos los puntos que no sean máximos locales para poder obtener una lista definitiva de puntos con importancia de la imagen. Para eliminar los no-máximos locales, se crea una matriz binaria de tamaño similar a la matriz de puntos. Esta matriz se utiliza para saber si un punto se debe comprobar o no (al comenzar todos los puntos se deben comprobar). El siguiente paso es recorrer todos los

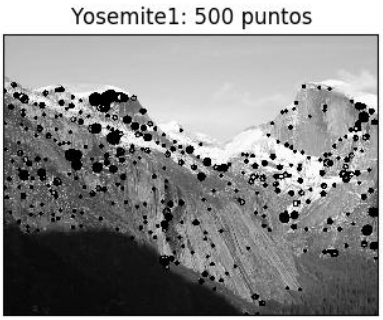
puntos de esta matriz (si están marcados como no revisados), definiendo una zona de tamaño *env = 5* para cada punto. Esto significa que el número de filas y columnas que rodean al punto central, es de tamaño 5. Cuando un punto sí es el máximo de cada zona, se guarda su KeyPoint correspondiente y será el seleccionado para mostrarse. Este proceso se realiza llamando a la función creada *not_max_suppression_harris().*

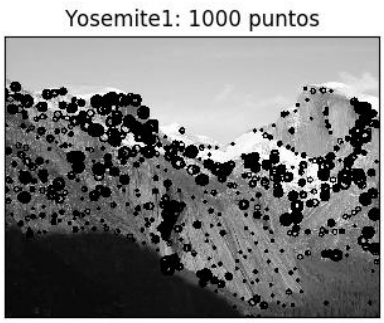
Si la imagen tiene una gran resolución, la cantidad de puntos Harris detectados puede ser de un gran tamaño. A medida que se obtienen los puntos de niveles inferiores de la pirámide, esta cantidad disminuye debido a que la máscara Gaussiana elimina información. Finalmente se tendrá un contenedor con multitud de puntos de las distintas escalas, de los que se deberán seleccionar los que mayor valor tengan. Para hacer esta operación se utiliza la función creada *get_best_harris()*, obteniendo los n mejores puntos.

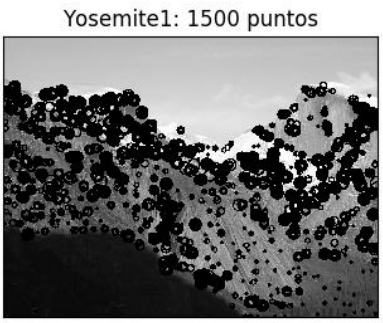
El paso final es pintar los puntos sobre la imagen. Para hacerlo se utiliza la función creada *show_circles()*, que llama a su vez a la función de OpenCV *circle()*. Teniendo en cuenta la lista con los distintos puntos, pinta un círculo del radio correspondiente al nivel de la pirámide, siendo los de menor radio los de la imagen original.

Para comprobar el resultado, se muestran sobre la imagen yosemite1.jpg los puntos Harris detectados, mostrando 500, 1.000 y 1.500 puntos.









En el resultado se puede apreciar como en zonas donde existe un mayor detalle, los círculos tienen un radio inferior, lo que demuestra que esos puntos han sido conseguidos por la imagen original que tiene toda la información.

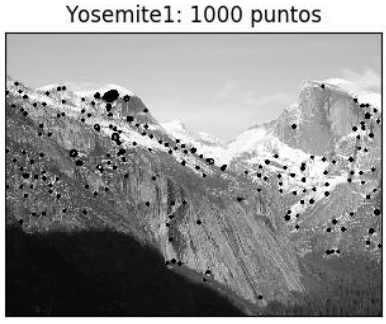
A medida que se va mostrando una cantidad mayor de puntos, aparecen los de los niveles inferiores, cuyo valor Harris es menor al no ser detectado con tanta claridad.

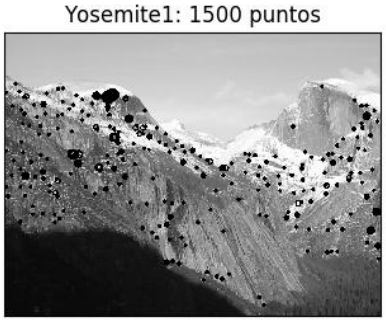
b) Extraer los valores (cx, cy, escala) de cada uno de los puntos resultantes en el apartado anterior y refinar su posición espacial a nivel sub-píxel usando la función de OpenCV *cornerSubPix()* con la imagen del nivel de pirámide correspondiente. Actualizar los datos (cx, cy, escala) de cada uno de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

Para refinar los puntos encontrados se hace uso de la función creada *refine_harris_points()*. Esta función se debe aplicar después de obtener todos los puntos Harris y antes de elegir los *n* mejores para no descartar cambios en su valor.

Para obtener los puntos refinados, se utiliza la función de OpenCV *cornerSubPix()* enviando la imagen correspondiente de cada nivel de la pirámide y la lista de todos los puntos que pertenecen a ese nivel.

Finalmente se actualizan las coordenadas de los puntos que devuelve la función de OpenCV en la posición correspondiente.





Pese a que visualmente parece que la cantidad de puntos ha disminuido, en cada imagen se encuentran los 500, 1.000 y 1.500 puntos que se indica en el título.

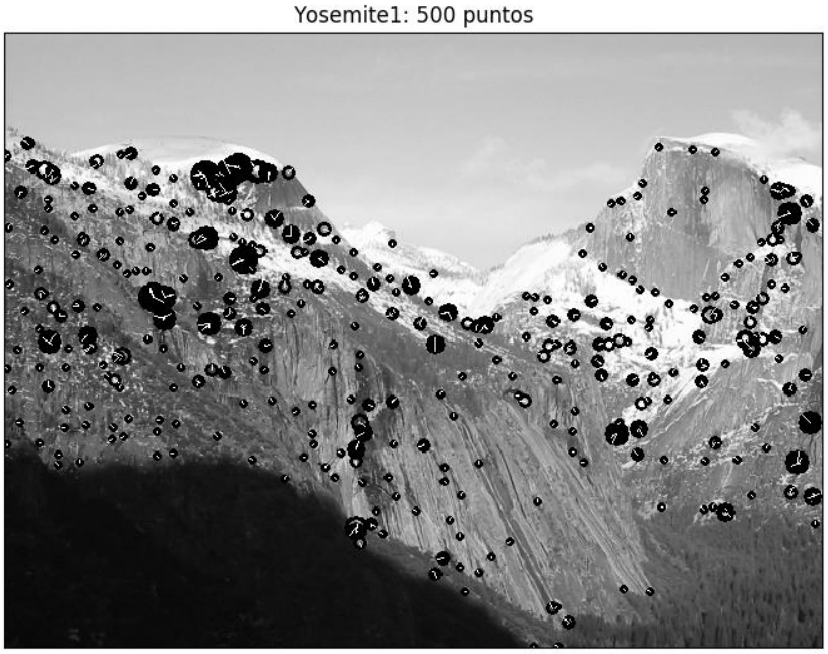
Al aplicar el refinamiento, principalmente se muestran los puntos de la imagen original, que es la que mayor detalle tiene y todos se encuentran agrupados en las mismas zonas con un fuerte contraste.

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 y derivada-y, en la escala correspondiente, como propone el paper MOPS de Bwon&Szeliski&Winder. 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.

Cuando ya se han obtenido los mejores *n* puntos Harris es el momento de calcular su orientación. Para realizar este cálculo se utiliza la función creada *get_orientations()*. Esta función actualiza los puntos añadiéndole el ángulo estimado que poseen.

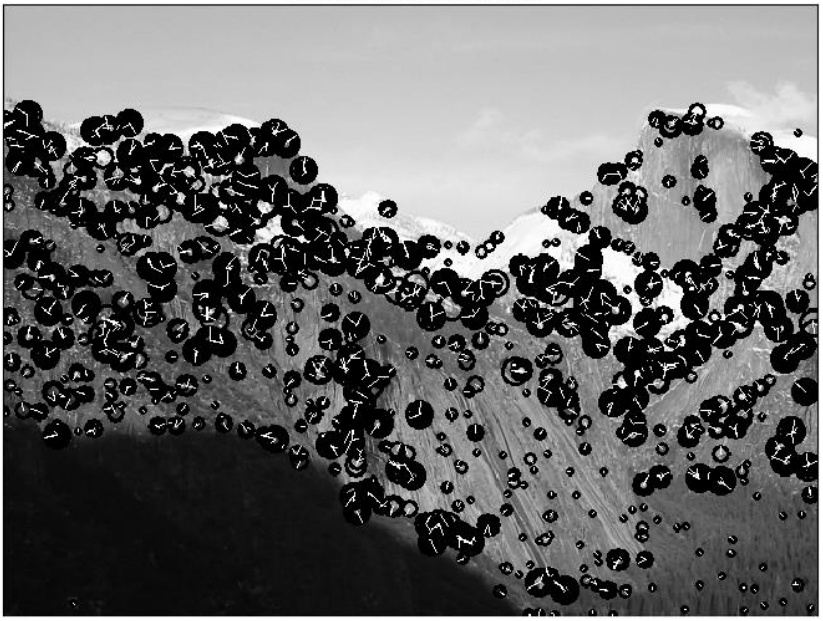
Para calcular la orientación es necesario calcular el gradiente en cada punto y para ello es necesario calcular las derivadas con respecto de x y con respecto de y de la imagen. Se utiliza la función creada en el trabajo 1 *convolution_d(),* que devuelve las derivadas. También se especifica *sigma = 5* para el alisamiento antes del cálculo de las derivadas, ya que el paper MOPS de Bwon&Szeliski&Winder indica que se utilice *4.5* y la máscara utilizada debe tener tamaño impar para su correcto funcionamiento.

El siguiente paso es calcular el gradiente, haciendo el arco tangente de cada punto con las derivadas de la imagen. Finalmente, con todos los datos obtenidos, se añade en la estructura KeyPoint de cada punto, el resultado del cálculo del ángulo de dicho punto.



Yosemite1: 1000 puntos

Yosemite1: 1500 puntos



Aparecen las imágenes de los apartados anteriores con la orientación estimada en cada punto.

d) Usar el vector de KeyPoint extraídos para calcular los descriptores SIFT asociados a cada punto.

Al disponer de una lista con los KeyPoints de la imagen, solo es necesario enviar a la función de OpenCV *compute()* la imagen y los KeyPoints ya calculados, devolviendo ésta los KeyPoints y sus correspondientes descriptores.

El resultado es una matriz con los elementos anteriores, siendo una mínima parte de los descriptores la siguiente:

```
[[  0.   0.   0.   0.  16.  55.  56.   1.   0.   0.   3.   6.
   22.  21.  29.   6.   1.  18.  11.   5.   1.   4.   5.  10.
    0.  13.   8.   0.   1.  22.  14.   0.   0.   0.   1.  76.
   71.  18.   3.   0.   0.   0.  23. 126.  38.   6.   0.   0.
    0.  21. 123.  32.   2.   0.   0.   0.   0.  48. 113.   0.
    0.   1.   0.   0.   0.   0.  52. 134.   9.   0.   0.   0.
    0.   0. 121. 134.   4.   0.   0.   0.   0.   6. 134.  58.
    0.   0.   0.   0.   0.  32. 134.   1.   0.   0.   0.   0.
    0.   0.  77. 134.   0.   0.   0.   0.   0.   0. 134. 134.
    0.   0.   0.   0.   0.   1. 134.  34.   0.   0.   0.   0.
    0.  11. 134.   1.   0.   0.   0.   0.]
[   0.   0.   0.   0.   0.   0.  27. 141.   0.   0.   0.   0.
    0.   0.  60. 141.   0.   0.   0.   0.   0.   0. 125. 141.
    0.   0.   0.   0.   0.   8. 141.  45.  15.   0.   0.   0.
    0.   0.  13. 141.   5.   0.   0.   0.   0.   0.  31. 141.
    0.   0.   0.   0.   0.   3.  77. 136.   0.   0.   0.   5.
   15.  15.  44.  13. 111.   2.   0.   0.   0.   0.   1. 141.
   73.   7.   0.   1.   1.   0.   5. 131.   5.   0.   0.  13.
   13.   9.  12.  28.   0.   0.   0.  38.  37.   1.   2.   1.
  124.  56.   0.   0.   0.   0.   0.   7.  68.  55.   9.   1.
    0.   0.   0.   4.   1.  10.  13.  38.  10.   0.   0.   1.
    0.   0.   0.  64.  27.   0.   0.   0.]
[  17. 128.  12.   0.   0.   0.   0.  12.  19. 128.  83.   0.
    0.   0.   0.   1.  94. 105.  49.   1.   0.   2.   0.   1.
   54.   6.   0.   0.   1.  35.  40.  38.  29.  11.   1.   0.
    0.   0.   0.  26. 115.  44.   2.   0.   0.   0.   0.  27.
  128.  53.   0.   0.   0.   0.   0.   8. 128.  91.  20.   4.
    0.   5.   7.  10.  16.  44.  31.   1.   0.   0.   0.   1.
   84.  82.   9.   0.   0.   0.   0.  10.  80. 117.  69.   1.
    0.   0.   0.   6.  16. 103. 128.   7.   0.   0.   0.   0.
    0.  52.  47.   4.   0.   0.   0.   0.   3. 128.  50.   0.
    0.   0.   0.   0.   1.  51. 128.   9.   0.   0.   0.   0.
    0.   5. 128.  38.   0.   0.   0.   0.]
[   0.   1.   5.  15.  17.  21.  25.   1.   0.   0.   0.  42.
   82.  48.  11.   0.   0.   0.   0.  27. 105.  23.   1.   0.
    0.   0.   0.   3.  22.  24.   9.   6.   0.   0.   5.  21.
   30.  71.  35.   0.   0.   0.   0.  49. 148.  44.   0.   0.
    0.   0.   0.  48.  84.  33.   1.   0.   5.   0.   0.   1.
   12.   9.   3.  40.   0.   0.   0.   0.   2. 159. 138.   0.
    0.   0.   0.   0.  42. 159.  11.   0.   0.   0.   0.   0.
   19. 159.  23.   0.   7.   0.   0.   0.   1.  19.  31.  30.
    0.   0.   0.   0.  17. 159.  60.   0.   0.   0.   0.   0.
   66. 159.   5.   0.   0.   0.   0.   0.  46. 159.  12.   0.
    0.   0.   0.   0.   1.  53.  33.  15.]
```

2. Usar el detector descriptor SIFT de OpenCV sobre las imágenes de Yosemite.rar. 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".

Para la realización de este ejercicio se han creado tres funciones:

- get_keypoints_descriptors(img, own): dada una imagen, devuelve sus KeyPoints y descriptores correspondientes.

- get_matches_bf_cc(img1, img2, n, flag, get_data): dadas dos imágenes, calcula *n* matches entre ellas utilizando la técnica "BruteForce+crossCheck".

- get_matches_knn(img1, img2, k, ratio, n, flag, get_data, improve): dadas dos imágenes, calcula *n* matches entre ellas utilizando la técnica "BruteForce+crossCheck". Permite la opción de elegir los mejores matches de entre todos los encontrados (por defecto deshabilitado).

Este ejercicio consiste en mostrar en una misma ventana dos imágenes diferentes y los puntos Harris que se considera que se encuentran en ambas imágenes (matches).

Para calcular los matches se pueden utilizar dos técnicas: "BruteForce+crossCheck" o "Lowe-Average-2NN". En ambas técnicas es necesario disponer de los KeyPoints y descriptores de cada imagen, por lo que se crea una función *get_keypoints_descriptors()* que los devuelven. Esta función permite indicar a través del parámetro *own* si se utiliza las funciones propias creadas en apartados anteriores o se utiliza la de OpenCV. En este caso se ha utilizado la propia de OpenCV *sift.detectAndCompute()*.

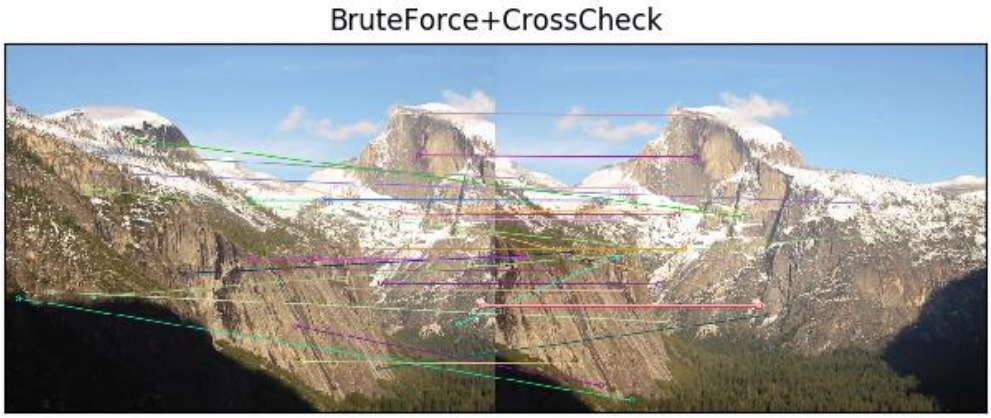
Cuando ya se disponen de los datos necesarios, se crea el objeto BFMatcher, indicando si se realiza la validación cruzada activando el flag *crossCheck = True* en el caso de utilizar la técnica "BruteForce+crossCheck".

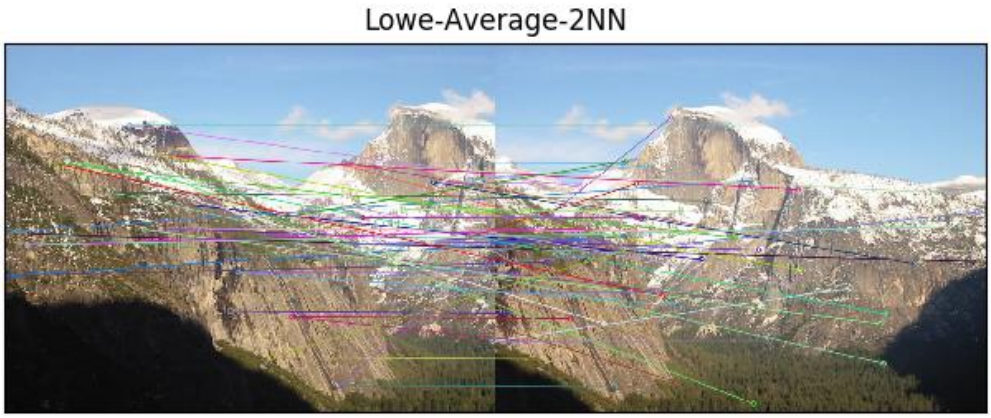
Para la técnica "BruteForce+crossCheck" se llama a la función del objeto BFMatcher *match()*.

Para la técnica "Lowe-Average-2NN" se llama a la función del objeto BFMatcher *knnMatch()*, indicando *k = 2* como parámetro para el cálculo del vecino más cercano.

Finalmente, con las funciones de OpenCV *drawMatches()* y *drawMatchesKnn()* se envían las dos imágenes, sus KeyPoints y los matches entre ambas imágenes.

Se ha establecido un límite en la visualización de matches que se encuentra en 30. Los puntos mostrados se han elegido de manera aleatoria de entre los disponibles.





Con ambas técnicas se encuentra una irregularidad en los matches. Debido a que la inclinación de la cámara en el momento del disparo es similar, se pueden identificar los matches correctos como aquéllos cuya línea que los une es horizontal.

El listado de matches que se han encontrado es bastante numeroso y al haberlos elegido aleatoriamente no se puede garantizar que coincidan.

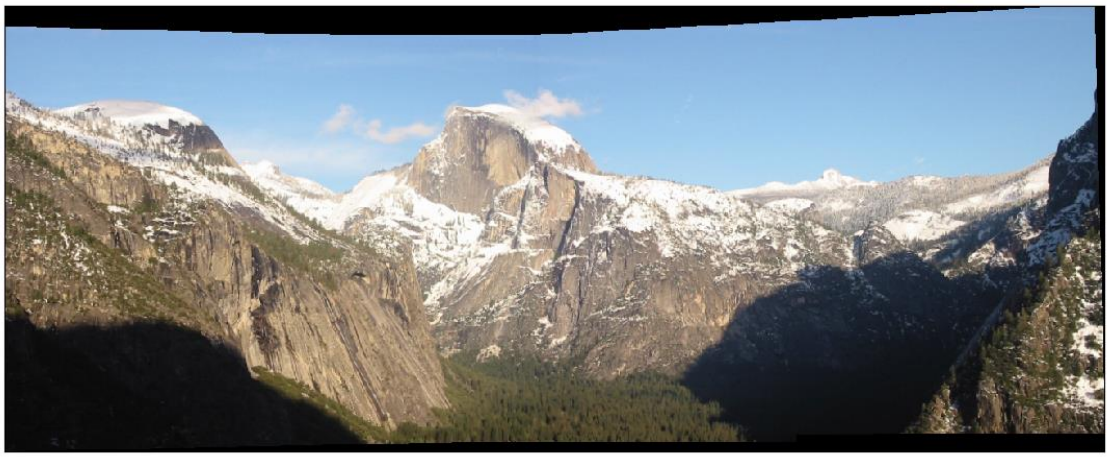
Un problema que se presenta al no coincidir los puntos en ambas imágenes se puede dar en la generación de mosaicos. Al necesitar estos al menos 4 puntos, si no son válidos, el mosaico puede que no se cree correctamente. El hecho de elegir puntos aleatorios también permite que en varias ejecuciones se puedan generar mosaicos correctamente y en otras, no se encuentren los puntos mínimos que coincidan.

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.

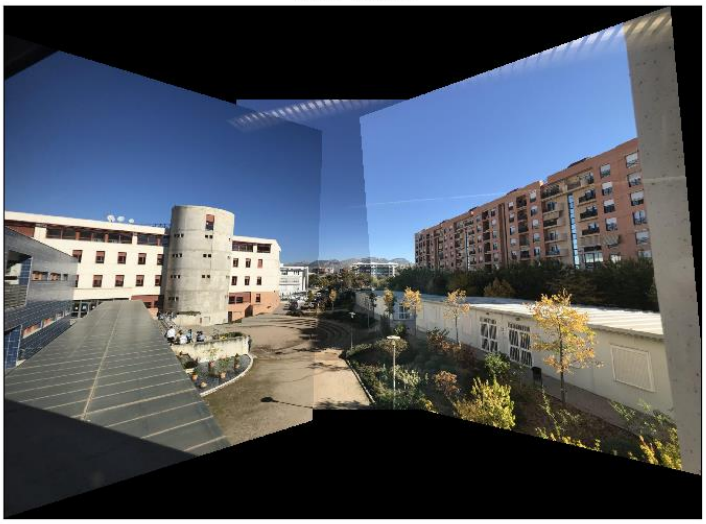
Directamente se ha desarrollado una función que sea capaz de realizar un mosaico con n imágenes, siendo $n > 1$. Se explica en ejercicio 4 el proceso llevado a cabo para realizarlo.

Los mosaicos creados con 3 imágenes para este ejercicio son:

Yosemite Mosaic



ETSIIT Mosaic



Debido a la explicación dada en el ejercicio anterior sobre la aleatoriedad de los matches, es posible que, en algunas ejecuciones del script, el mosaico de la ETSIIT salga de manera incorrecta. Es suficiente con volver a ejecutarlo para que se generen los matches que coincidan correctamente. Se ha implementado una mejora de los matches generados para reducir este problema.

4. Punto 3 pero para N > 5.

Para la realización de este ejercicio se han creado las siguientes funciones:

- get_homography(img1, img2, improve): calcula la homografía entre las dos imágenes.
- get_homography_to_center(img, width, height): calcula la homografía que permite desplazar una imagen a una ventana de tamaño width x height.
- get_mosaic(imgs, crop, improve): genera un mosaico a partir de n imágenes. Se puede indicar si se quiere recortar los bordes sobrantes del mosaico.
- crop_image(img): recorta todos los bordes negros de una imagen. Código utilizado de: https://stackoverflow.com/questions/13538748/crop-black-edges-with-opencv

Para la generación del mosaico se deben realizar una serie de transformaciones en las imágenes. La función que se encarga de su generación es *get_mosaic()*.

El primer paso que se debe realizar para generar el mosaico es calcular el tamaño que tendrá. En este caso se ha establecido el ancho como la suma de todos los anchos de las distintas imágenes que componen el mosaico. La altura será el doble de la altura de la imagen central.

Se ha creado una función *crop_image()* que elimina todos los bordes negros que se tienen en el mosaico tras añadir todas las imágenes. Como inicialmente no se conoce cuál es el tamaño mínimo que debe tener el contenedor, esta función permite trabajar con un tamaño mucho mayor al que finalmente se devolverá.

Cuando se crean mosaicos o panoramas, la imagen con mayor importancia y sobre la que recae el peso de una correcta visualización es la imagen central. Es sobre esta imagen central sobre la que se van uniendo el resto de imágenes hacia los extremos.

Para comenzar a colocar las imágenes, se calcula la que debe estar en el centro y se genera una homografía que sea capaz de trasladar esta imagen central, al centro del mosaico. Esta homografía se calcula manualmente en la función *get_homography_to_center()*. La homografía será una matriz 3x3 rellena de 1 en su diagonal, el desplazamiento en píxeles del eje X en la fila 1, columna 3 y el desplazamiento en píxeles del eje Y en la fila 2, columna 3. El resto de posiciones de la matriz tendrán valor 0.

Cuando ya se tiene la imagen central y su homografía, se utiliza la función de OpenCV *warpPerspective(),* que recibe la imagen y la homografía calculada. En este punto se tiene un contenedor del tamaño que se haya calculado y la imagen central en el centro de él.

El siguiente paso es colocar las imágenes desde el centro hasta los extremos en su lugar correspondiente del mosaico. Para ello se ha dividido este proceso en dos partes:

1. Cálculo de las homografías desde el centro hacia el extremo izquierdo.
2. Cálculo de las homografías desde el centro hacia el extremo derecho.

Se explica solo el proceso de colocar las imágenes que se encuentran a la izquierda de la principal, siendo el proceso de las imágenes derechas similar.

Al ser la imagen central la que tiene el peso del mosaico, para colocar el resto se debe partir de ella, calculando la homografía entre la imagen inmediatamente a su izquierda y ella (imagen central). Cuando ya se tiene la homografía, se sabe cuál es la transformación que se debe realizar sobre la imagen de la izquierda para que encaje con la central. El problema se encuentra ahora en que la imagen central no está en la posición original, por lo que visualmente no casan. Como se ha calculado la homografía que desplaza la imagen central al centro del mosaico, se multiplica esta homografía con la calculada entre ambas imágenes. En este momento se vuelve a utilizar la función de OpenCV *warpPerspective()* para trasladar la imagen de la izquierda al lugar correcto del mosaico.

El resto de imágenes que se encuentran en la parte izquierda no vuelven a utilizar la imagen central, sino que se realizan los cálculos con la que se encuentra a su lado.

Siendo n = 5, el número de imágenes del mosaico. Si se calcula la homografía entre la imagen i1 y la imagen i2, es necesario multiplicar esta homografía por la homografía de la imagen i2 y la imagen i3, ya que la última contiene la traslación que se debe llevar a cabo para colocar la imagen i1 en su lugar correspondiente del mosaico.

Se han realizado dos pruebas:

1. Mosaico con 13 imágenes propias:

2. **Mosaico con las 10 imágenes "mosaico0**.jpg":**