

Práctica 2

Visión por Computador:

Detección de puntos relevantes y construcción de panoramas.

Óscar David López Arcos, 75571640-B
odlarcos@correo.ugr.es
Grupo: Miércoles 11:30-13:30

Índice

Ejercicio 1	3
Ejercicio 2	6
Ejercicio 3	7
Ejercicio 4	9

Ejercicio 1

Aplicar la detección de puntos SIFT y SURF sobre las imágenes, representar dichos puntos sobre las imágenes haciendo uso de la función drawKeyPoints.

- a) **Variar los valores de umbral de la función de detección de puntos hasta obtener un conjunto numeroso (≥ 1000) de puntos SIFT y SURF que sea representativo de la imagen.**

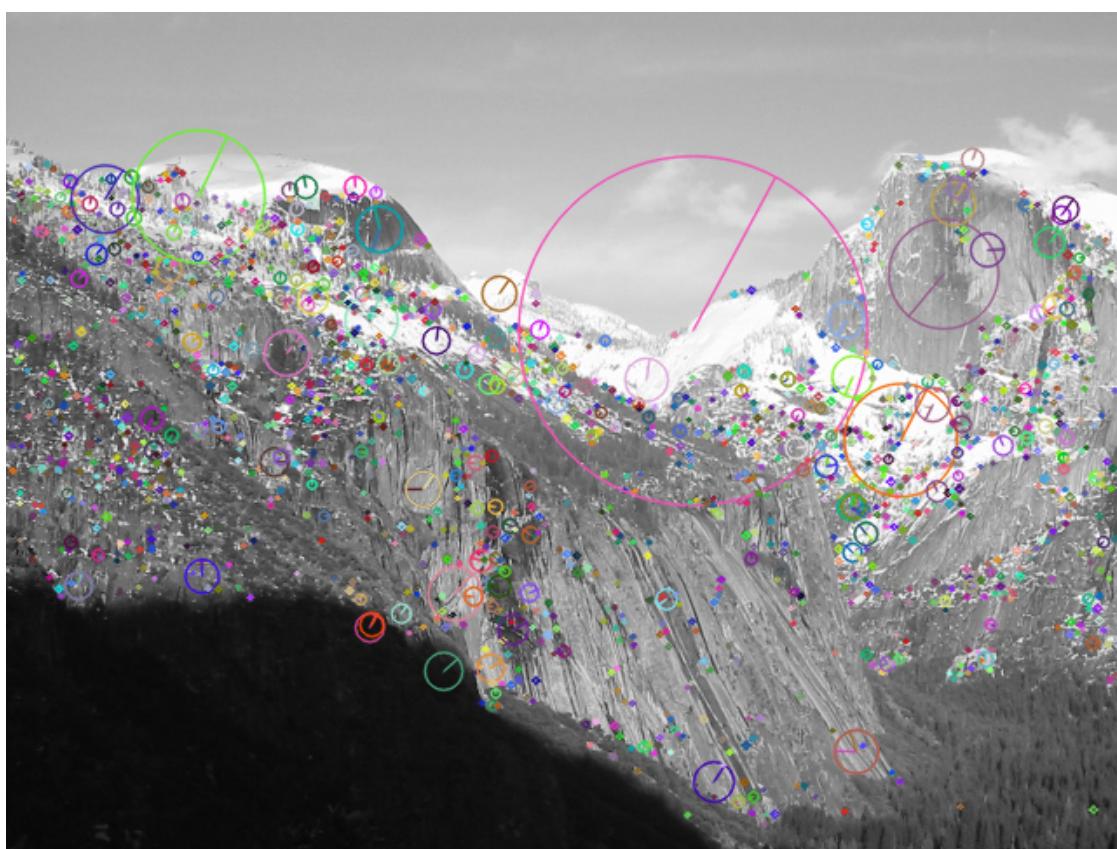
Como ya sabemos, tanto Sift como Surf son algoritmos usados para extraer características relevantes de la imagen. Aunque ambos siguen la misma filosofía, Surf es más rápido al emplear el uso de la imagen integral.

Las funciones SIFT y SURF de openCV aplican estos algoritmos y aceptan una serie de parámetros en el método “**create**” que podremos modificar para realizar este ejercicio.

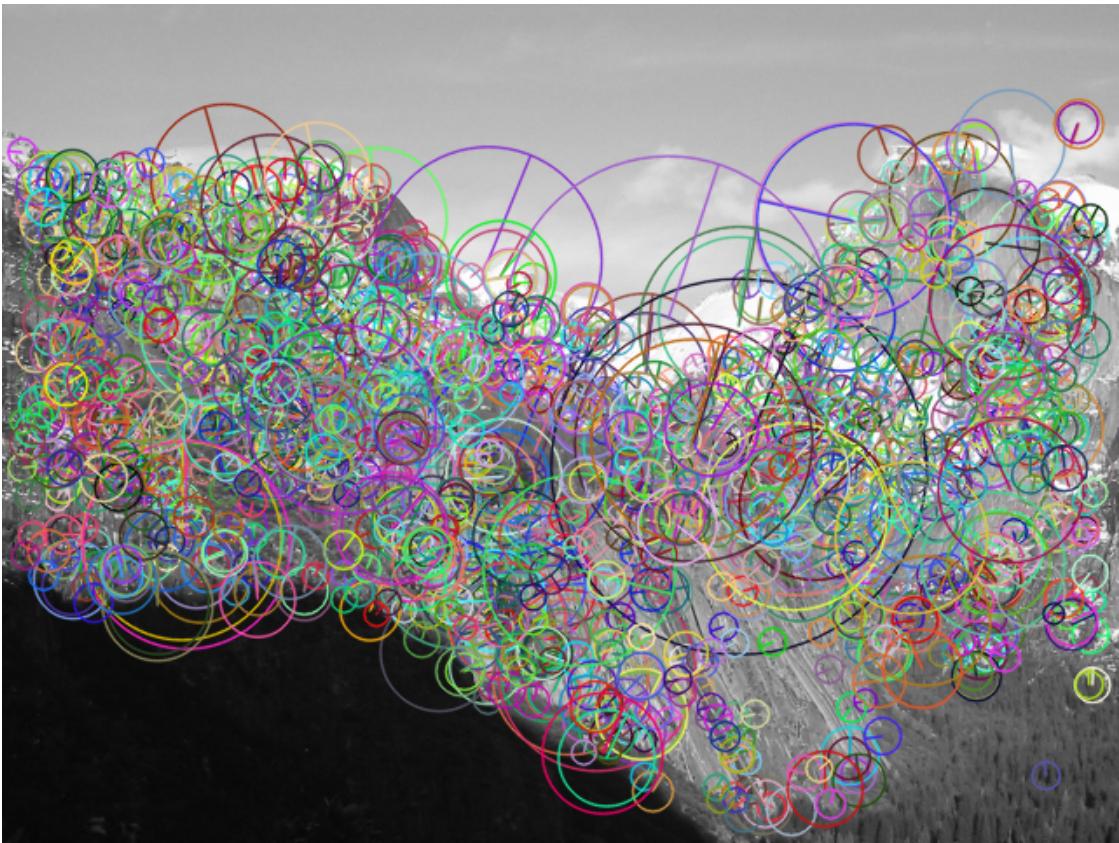
El propósito de estos parámetros viene detallado en los manuales de referencia (https://docs.opencv.org/3.4/d5/d3c/classcv_1_1xfeatures2d_1_1SIFT.html y https://docs.opencv.org/3.4.4/d5/df7/classcv_1_1xfeatures2d_1_1SURF.html).

Los valores usados por defecto en el caso de Sift serán suficientes para cumplir los requisitos indicados por el enunciado. Estos valores son: 0 para el numero de características, 3 capas en cada octava, un umbral de contraste de 0.04, un límite de umbral de 10 y un valor de sigma de 1.6 (usado en la octava 0). El número de octavas es computado automáticamente a partir de la resolución de la imagen.

He modificado el contrastThreshold, aumentándolo a 0.06 para producir menos keypoints.



Por otro lado, en Surf he modificado el umbral a 500 (**surf.setHessianThreshold(500)**), ya que el valor recomendado para la mayoría de imágenes se encuentra en el intervalo 300-500. El resto de parámetros por defecto permiten obtener de igual forma un número de puntos superior a 1000. Estos parámetros incluyen un número de octavas igual a 4 y capas por octava igual a 3.



SURF (1510 KP)

b) Identificar cuántos puntos se han detectado en cada octava (y capa para SIFT). Mostrar el resultado sobre la imagen original.

Para resolver este ejercicio, debemos buscar en la clase **Keypoint** de OpenCV y ver los atributos que contiene (https://docs.opencv.org/3.4/d2/d29/classcv_1_1KeyPoint.html).

El atributo `octave` nos indica en qué octava fue detectado ese keypoint, sin embargo, en el caso de SIFT devuelve un número compuesto por el valor de la octava, la capa donde fue detectado y la escala.

Para deshacer esta operación, debemos observar el código de la función (https://github.com/opencv/opencv_contrib/blob/master/modules/xfeatures2d/src/sift.cpp) y realizar la operación inversa (`unpackOctave`) obteniendo los tres valores antes comentados.

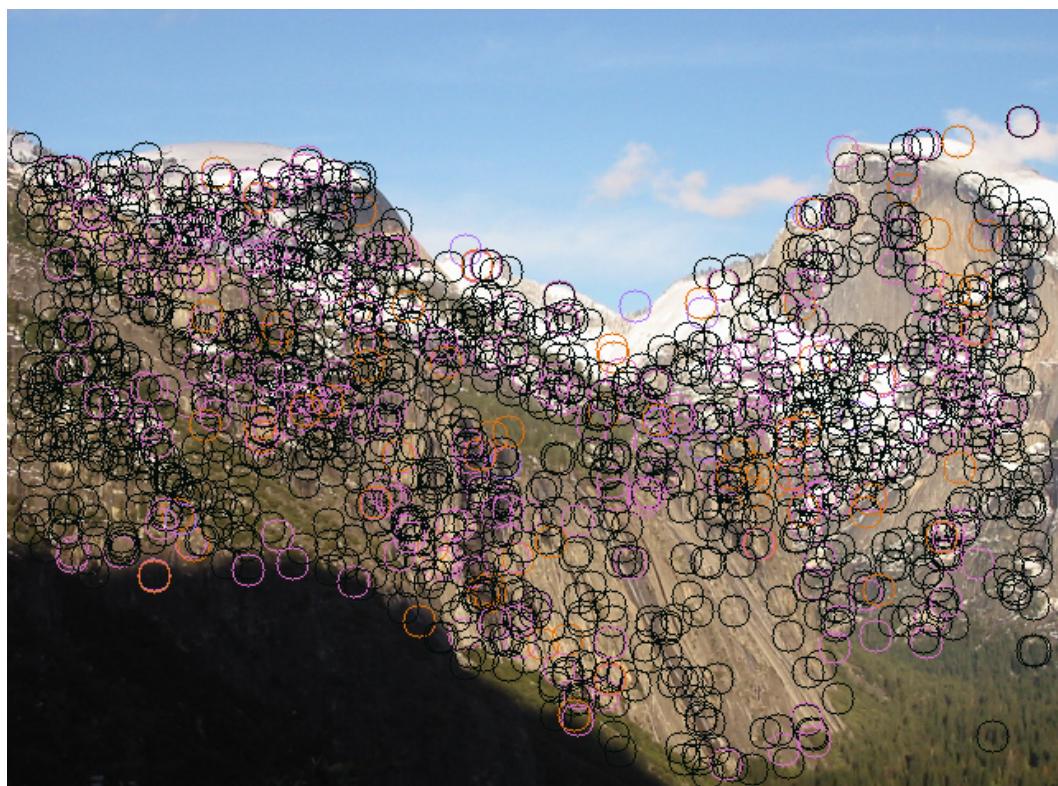
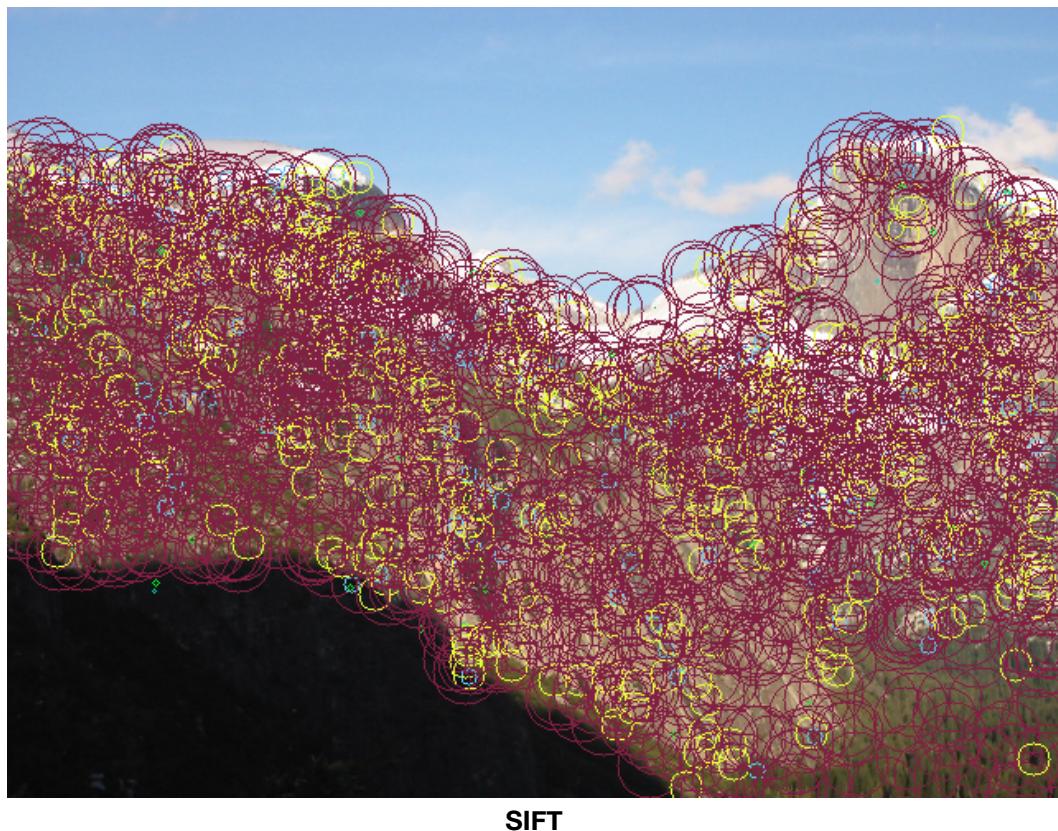
Una vez resuelto este problema, procedemos a resolver el ejercicio:

Definimos las funciones **get_octave_sift/surf**, que se encargarán de calcular en qué octava (y capa para Sift) aparece cada punto. Lo mostramos por consola con **showFoundKeypoints**.

Posteriormente, las funciones **draw_circle_sift/surf**, dibujarán dichos puntos sobre la imagen original. Para ello, se crea un color para cada capa (**create_colors**) y, en el caso de sift, el radio del círculo se calcula en función de la escala con la que fue encontrado (ya que este parámetro está relacionado con el sigma).

Por limitaciones de la función circle, las imágenes deben ser a color para que los círculos se aprecien distintos. Por otro lado, utilizando surf no podemos acceder al sigma con el que fue detectado (al menos de forma sencilla), por lo que todos los círculos tendrán el mismo tamaño.

Destacar que la mayoría de los Keypoints son detectados en la octava y capa 0, ya que aquí la imagen presenta la mayor resolución.



c) Mostrar como con el vector de keyPoint extraídos se puede calcular los descriptores

Para este apartado debemos utilizar la función “**compute**” de OpenCV que, dado una imagen y sus keypoints, nos devolverá el descriptor asociado. El descriptor asocia a cada punto información acerca de su localización y contexto, haciendo posible la comparación de imágenes.

Ejercicio 2

Establecer las correspondencias entre las imágenes usando los criterios “BruteForce+crossCheck” y “Lowe-Average-2NN”

Para construir las correspondencias basadas en Fuerza Bruta con correlación cruzada, creamos el objeto **bf** llamando al método **BFMatcher** de OpenCV con la distancia **NORM_L1** (recomendada según la documentación https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_matcher/py_matcher.html) y activando la opción **crossCheck**.

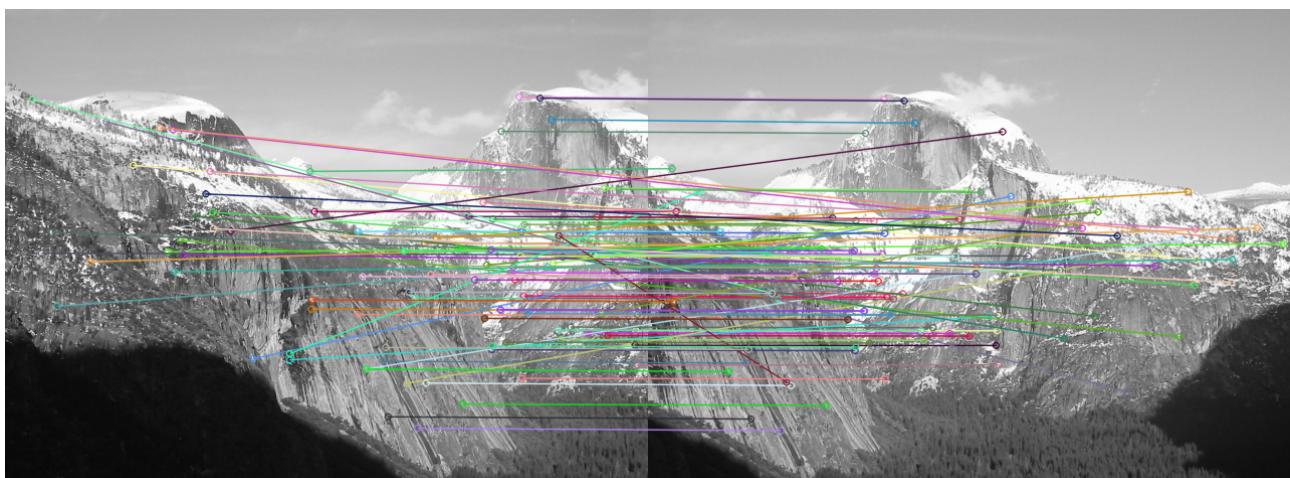
Calculamos las correspondencias con el método **match** o **knnMatch** de OpenCV. Estos métodos se encargarán de, en base a las opciones especificadas en el constructor, calcular las correspondencias haciendo uso de los keypoints y descriptores de ambas imágenes (**detectAndCompute**).

El método **match** compara cada punto de una imagen con todos los de la otra, repitiendo este procedimiento para ambas imágenes. Posteriormente, para cada punto elige como pareja aquel que haya obtenido menor valor en el cálculo de la distancia especificada. Como hemos activado validación cruzada, sólo escogerá los puntos de ambas imágenes cuyo cálculo de pareja se corresponda en ambos sentidos.

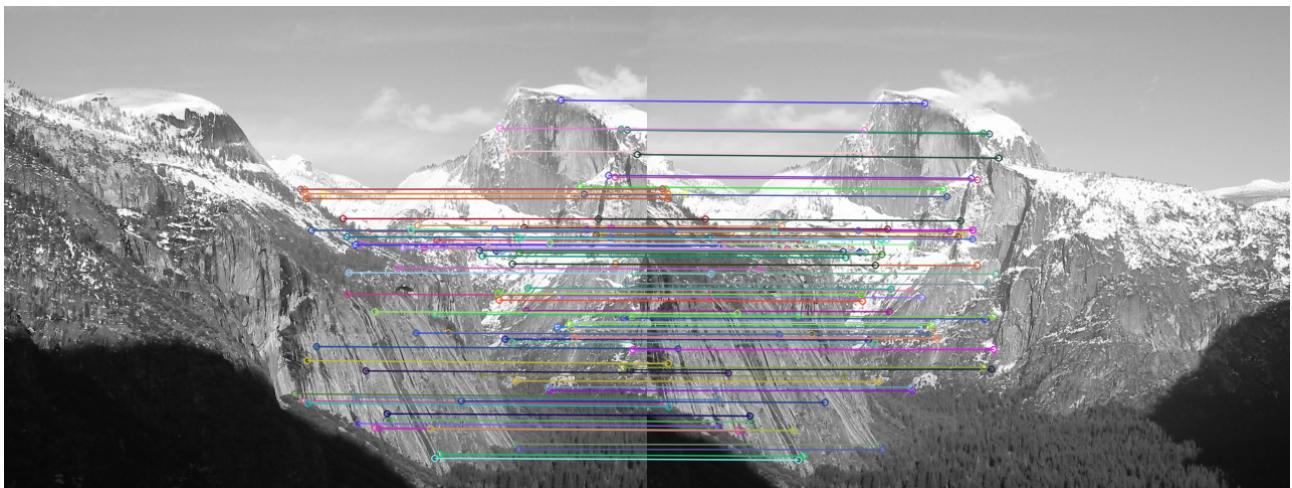
El método **knnMatch** con $k=2$ sólo calcula las correspondencias de una imagen con la otra, quedándose con el más cercano. Es necesario aplicar el criterio Lowe-Average para filtrar los resultados y, como propone en su paper, quedarnos únicamente con las correspondencias de los puntos (i,j) , siendo $d(i,j)$ la distancia más pequeña y cumpliéndose que $d(i,k) < 0,7 \times d(i,j)$, siendo $d(i,k)$ la siguiente distancia más pequeña desde el punto i .

a) Mostrar 100 correspondencias aleatorias con cada criterio

Para mostrar las correspondencias aleatoriamente, desordenamos el vector de puntos con la clase **random** y mostramos los 100 resultantes:



BRUTEFORCE+CROSSCHECK



LOWE-AVERAGE-2NN

b) Valorar la calidad de los resultados

En general ambos métodos ofrecen buenos resultados (líneas horizontales), siendo los de 2NN algo mejores. Esto último es lógico ya que el criterio de Lowe realiza una criba en los puntos para quedarnos con las correspondencias que menos dudas generan.

El BF ofrece resultados mucho mejores si en lugar de elegir 100 aleatorios nos quedamos con las 100 mejores correspondencias. Esto se debe principalmente a que las imágenes tienen puntos muy parecidos y la elección de la correspondencia no tiene en cuenta que pudiera existir otra casi igual de buena.

c) Comparar ambas técnicas

Como he explicado antes, si escogemos puntos aleatorios, 2NN suele ser mejor debido a la aplicación de Lowe-Average, donde nos quedamos con los puntos que menos dudas generan a la hora de establecer su correspondencia.

BF puede funcionar muy bien gracias a la validación cruzada, pero, a diferencia de Lowe-Average-2NN, el algoritmo no contempla que puedan existir varias correspondencias posibles para un mismo punto. En la mayoría de imágenes a cada uno se le podrían asignar potencialmente varias correspondencias; este algoritmo elige únicamente la mejor, sin tener en cuenta que la siguiente en discordia podría ser casi tan buena como la primera.

En temas de velocidad 2NN es superior, puesto que no tiene que comparar los puntos de ambas imágenes con todos los de la otra (con una basta).

Ejercicio 3

Escribir una función que genere un mosaico de calidad para N=3 imágenes.

Para resolver este ejercicio he propuesto dos formas distintas que explicaré a continuación, definidas en las funciones **mosaico_R** y **mosaico_mat**.

Ambas reciben por parámetro un vector de imágenes y crean la imagen **frame**, que será el marco donde se irá construyendo el mosaico. Para asegurarnos que hay espacio de sobra, le damos un tamaño del doble de la suma del ancho y alto de todas las imágenes que queremos componer.

Desplazamos la primera imagen al centro del marco con la función **move_to_center**, creando una matriz de homografía que cumpla este propósito. Para **mosaico_R** usaremos la imagen ya compuesta, mientras que **mosaico_mat** solo nos interesaría la matriz de homografía.

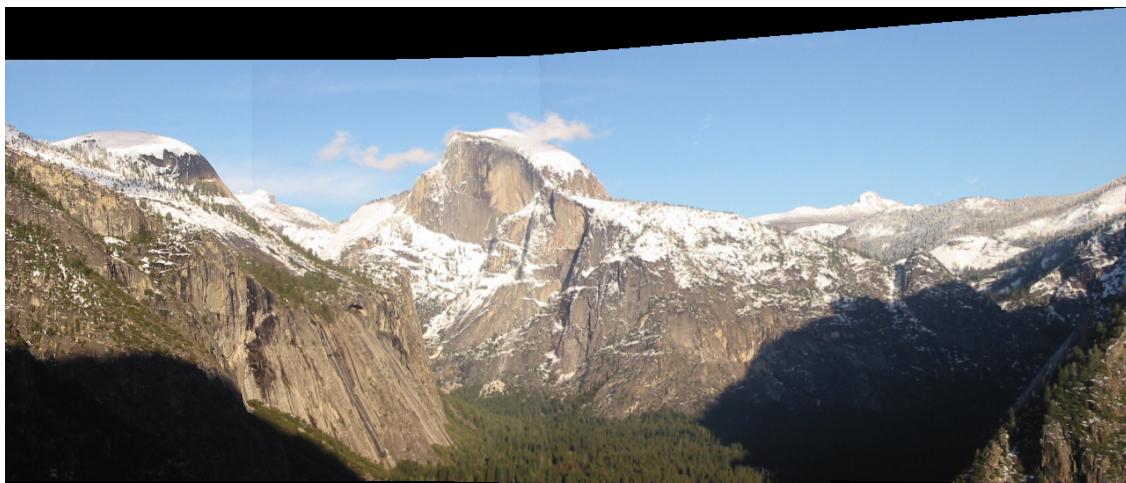
Una vez llegados a este punto, ambas funciones difieren en el cálculo del mosaico:

- **mosaico_R**: Compone el mosaico paso a paso, pegando una imagen detrás de otra. La primera imagen se traslada al centro del marco y, de aquí en adelante, se irán calculando las homografías del resto de imágenes con el mosaico parcial para construirlo poco a poco.
- **mosaico_M**: Se calculan previamente las homografías necesarias para trasladar cada imagen al mosaico (guardándolas en el vector **homografias**). Posteriormente, se trasladan todas las imágenes al marco.

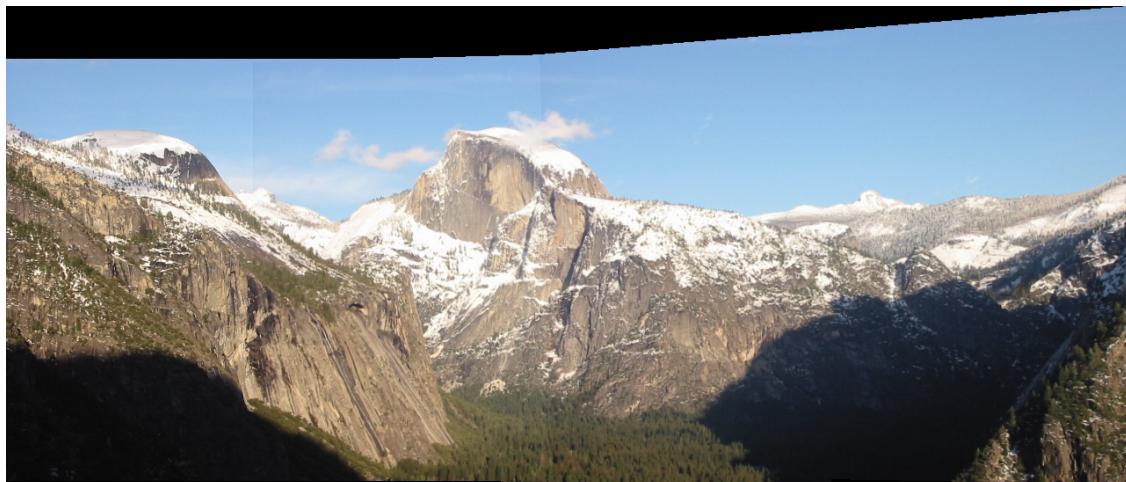
Para obtener dichas homografías, se calcula la necesaria para componer imágenes 2 a 2 y se multiplica por la matriz de homografía acumulada para trasladarla a su posición en el marco. La primera homografía que usaremos será aquella que permita trasladar la primera imagen al centro del mosaico.

Cabe destacar que para obtener las homografías (**get_homeography**) es necesario calcular los matches entre ambas imágenes (como vimos en el ejercicio anterior). Posteriormente, la función **findHomography** calculará la matriz necesaria para llevar a cabo esa traslación (<https://www.programcreek.com/python/example/89367/cv2.findHomography>) y **warpPerspective** compondrá las imágenes en una sola

Por último, eliminamos los bordes negros restantes del mosaico (**crop_image**) (Referencia: <https://stackoverflow.com/questions/13538748/crop-black-edges-with-opencv>) y mostramos el resultado. Las funciones están implementadas para controlar si las imágenes son RGB o escala de grises.



MOSAICO_M



MOSAICO_R

Como se puede apreciar, ambos métodos componen el mosaico de forma correcta aunque **mosaico_M** se ajusta mejor a lo pedido en el enunciado, ya que antes de trasladar las imágenes calcula todas las homografías. Además, es mucho más rápido a la hora de componerlo.

Ejercicio 4

Implementar el punto anterios para $N > 5$

El mismo código del ejercicio anterior nos valdrá para este apartado. Los resultados obtenidos son:



MOSAICO_M



MOSAICO_R