Práctica 2 - Detección de puntos relevantes y construcción de panoramas

Álvaro Fernández García

Noviembre, 2018

---

1. Detección de puntos SIFT y SURF. Aplicar la detección de puntos SIFT y SURF sobre las imágenes, representar dichos puntos sobre las imágenes haciendo uso de la función drawKeyPoints. Presentar los resultados con las imágenes Yosemite.rar.

   (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. Justificar la elección de los parámetros en relación a la representatividad de los puntos obtenidos.

   A continuación se muestran los resultados obtenidos:

Figura 1: 1246 puntos SURF, con un umbral Hessiano de 700

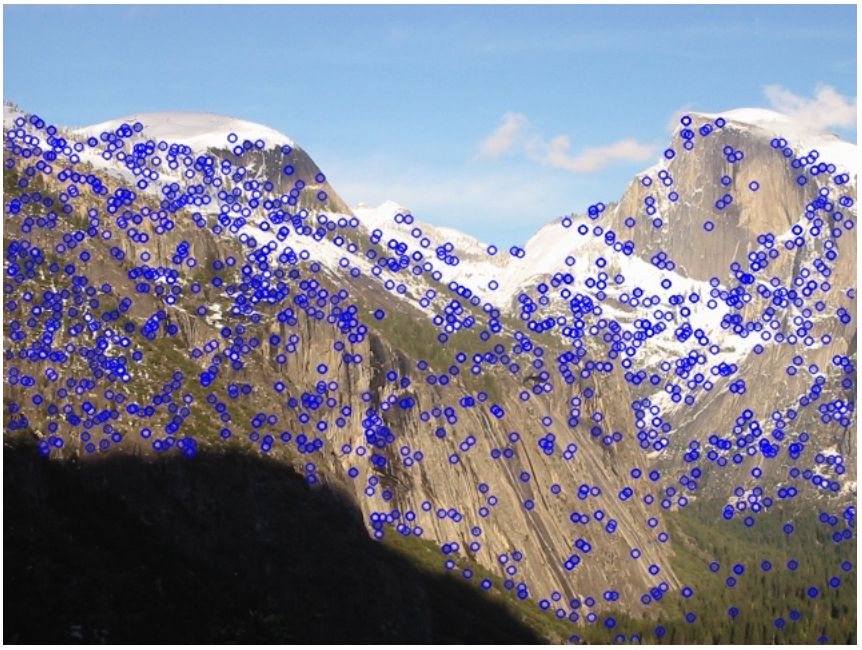


Figura 2: 1351 puntos SIFT, con un umbral de contraste = 0.04 y un umbral de borde = 2.7

Los umbrales especificados para ambas funciones se han obtenido realizando varios experimentos en los que se han ajustado sus valores de acuerdo al siguiente conocimiento:

- Umbral Hessiano: solo las regiones cuya Hessiana sea superior a este umbral serán recogidas por el detector. Por tanto, a mayor valor, menos puntos.
- Umbral de contraste: sirve para filtrar las regiones débiles en zonas de bajo contraste. A mayor valor, menos puntos.
- Umbral de borde: utilizado para filtrar los bordes de la imagen. A mayor umbral, más puntos se conservan.

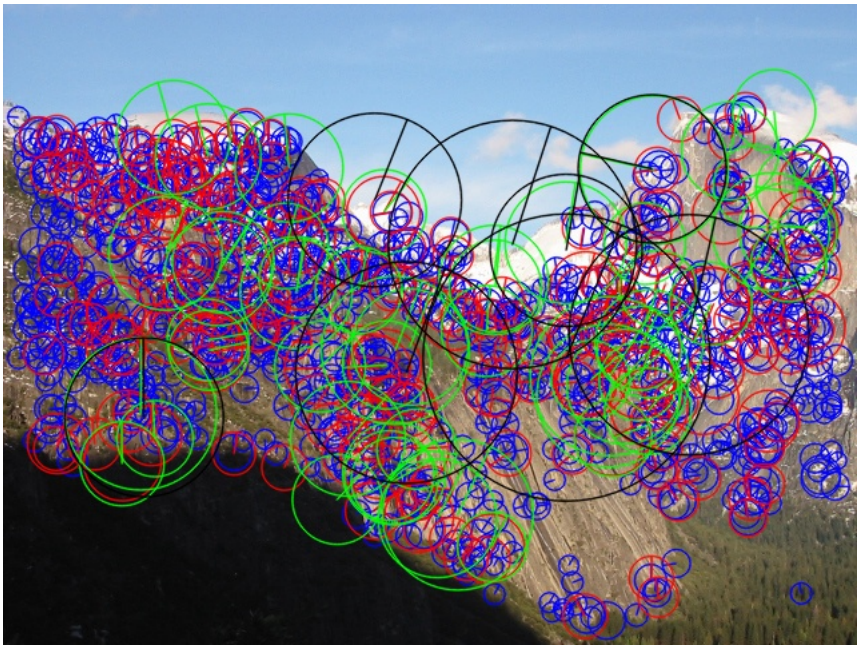
Con los valores por defecto de ambas funciones se obtenían demasiados puntos, por tanto, se han ido modificando hasta obtener una cantidad de puntos superior y cercana a 1000. Posteriormente, se han seguido variando los umbrales hasta obtener las imágenes mostradas.

He considerado que los resultados son satisfactorios cuando los puntos están más o menos esparcidos por las distintas zonas de interés de la imagen (cumbres, falda de la montaña, bosque...) y no aparecen puntos irrelevantes como, por ejemplo, un punto perdido en el cielo.

Resulta complejo establecer otro criterio distinto a la inspección visual que nos sirva para determinar la bondad de los puntos extraídos, ya que en este apartado, no los estamos utilizando para nada más.

(b) Identificar cuántos puntos se han detectado dentro de cada octava. En el caso de SIFT, identificar también los puntos detectados en cada capa. Mostrar el resultado dibujando sobre la imagen original un círculo centrado en cada punto y de radio proporcional al valor de sigma usado para su detección (ver circle()) y pintar cada octava en un color.

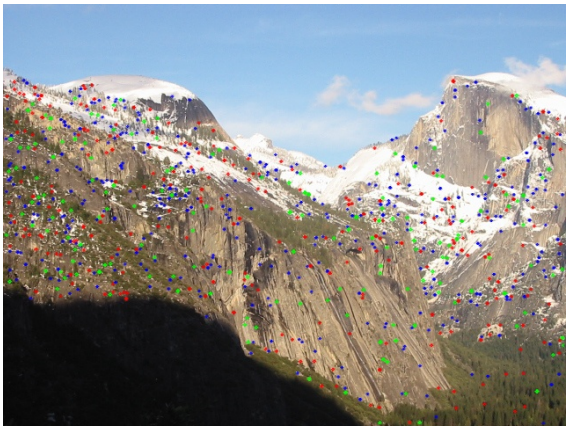
Puntos SURF por octavas:



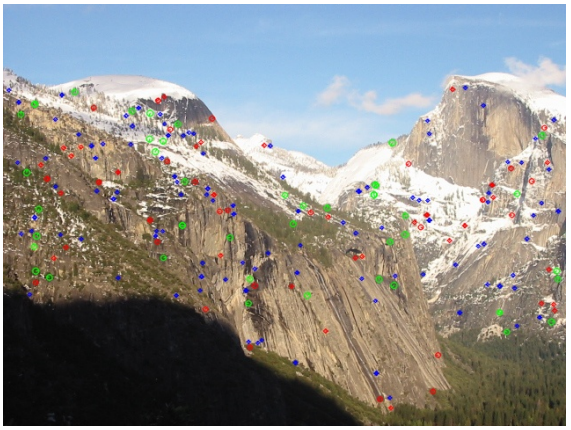
Octava	Nº Puntos	Color
0	970	Azul
1	215	Rojo
2	53	Verde
3	8	Negro

Cuadro 1: Número de puntos en cada octava y su correspondiente color

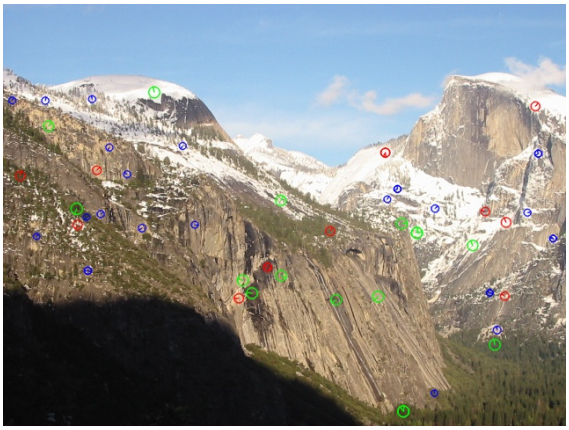
Puntos SIFT por octavas y capas. Capa 1 en azul, 2 en rojo y 3 en verde:



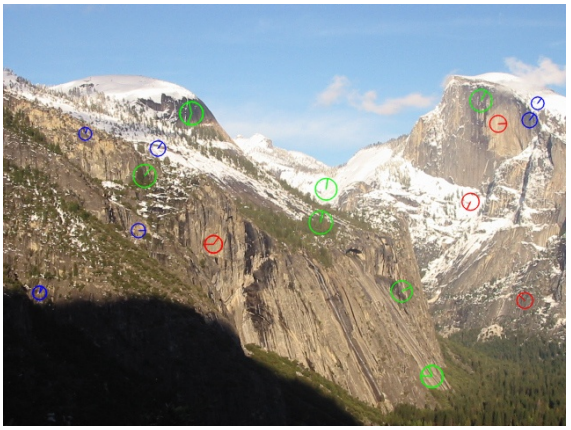
(a) Octava -1



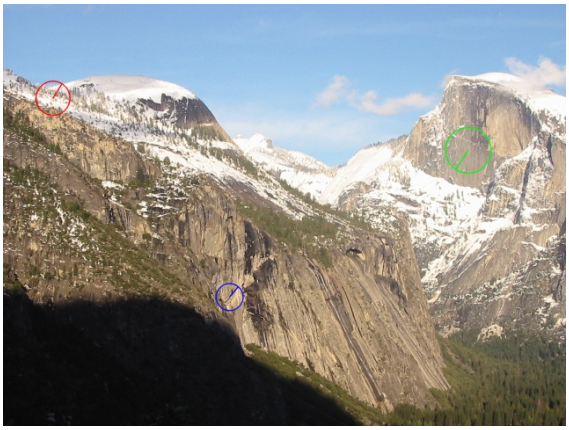
(b) Octava 0



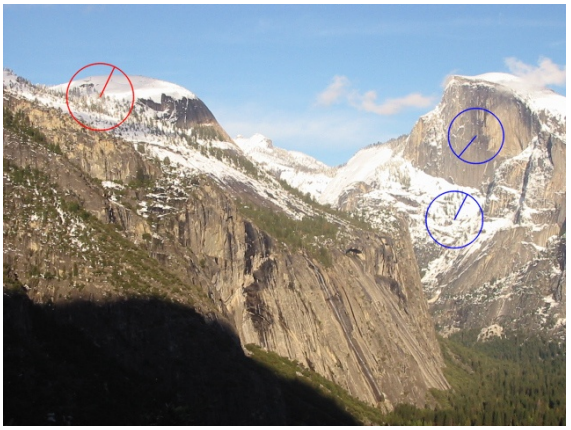
(a) Octava 1



(b) Octava 2



(c) Octava 3



(d) Octava 4

Octava	Capa 1	Capa 2	Capa 3	Total puntos
-1	494	321	205	1020
0	131	71	47	249
1	26	13	16	55
2	7	5	9	21
3	1	1	1	3
4	2	1	0	3

Cuadro 2: Número de puntos en cada octava y capa

Uno de los problemas que tiene el detector de Harris es que no es invariante ante la escala, es decir, una región detectada a una determinada escala de la imagen no tiene porqué ser detectada a otra diferente.

Para solventar este problema, SURF y SIFT constuyen un espacio de escalas utilizando la pirámide Gaussiana. Al fin y al cabo, procesar la imagen utilizando diferentes tamaños de ventana es equivalente a fijar un tamaño concreto y utilizar las distintas octavas de la imagen.

Por ello, a medida que nos acercamos a octavas superiores, las regiones detectadas son de mayor tamaño.

Por otra parte, el hecho de que conforme aumentamos la escala el número de puntos detectado disminuya se debe a la propia naturaleza de la imagen. Predominan los pequeños detalles y no hay muchas regiones de gran tamaño que sean relevantes.

Por último, la octava -1 de SIFT es un sobremuestreo de la imagen original. Es otra escala más que nos proporciona más puntos relevantes (ya que precisamente, lo que nos interesa es tener un número razonablemente grande de puntos).

(c) Mostrar cómo con el vector de keyPoint extraídos se pueden calcular los descriptores SIFT y SURF asociados a cada punto usando OpenCV.

Basta con utilizar la función compute() del modelo correspondiente. Esta función recibe la imagen y los keyPoints y nos devuelve nuevamente los puntos junto con los descriptores.

Para mostrar que, efectivamente, los descriptores son los mismos que los extraídos directamente de la imagen con detectAndCompute() se ha diseñado el siguiente fragmento de código:

```
surf_descriptors = surf.compute(src, kp_surf)[1]
sift_descriptors = sift.compute(src, kp_sift)[1]

surf_check = surf.detectAndCompute(src, None)[1]
sift_check = sift.detectAndCompute(src, None)[1]

cond1 = np.all(surf_descriptors == surf_check)
cond2 = np.all(sift_descriptors == sift_check)

if cond1 and cond2:
  print("\nLos descriptores son iguales")
else:
  print("\nLos descriptores NO son iguales")
```

Tras ejecutarlo, obtenemos el mensaje "Los desciptores son iguales". Esto demuestra que el cálculo de los puntos es independiente de los descriptores.

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 y los criterios de correspondencias "BruteForce+crossCheck y "Lowe-Average-2NN". (NOTA: Si se usan los resultados propios del puntos anterior en lugar del cálculo de SIFT de OpenCV se añaden 0.5 puntos)

   (a) 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 100 elegidas aleatoriamente.

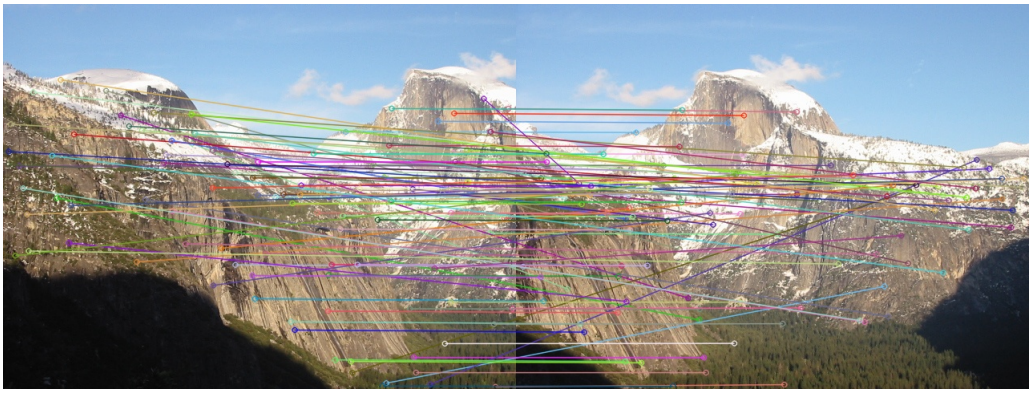


Figura 5: Fuerza Bruta + Cross Check: 100 aleatorios

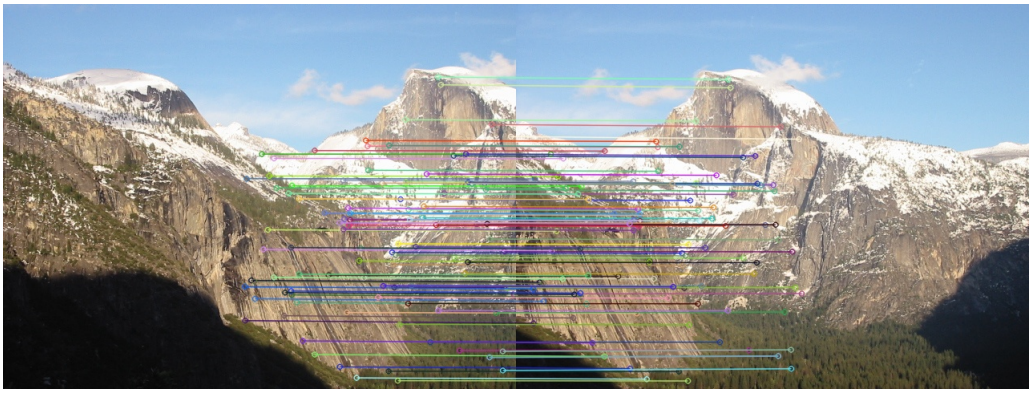


Figura 6: Lowe Average 2NN: 100 aleatorios

Nota: Como KNN se ha utilizado BFmatcher.knnMatcher(), pues el de FLANN es simplemente una optimización que funciona más rápido para grandes cantidades de puntos. Como no tenemos una gran cantidad de puntos, he utilizado el otro que además es más simple.

(b) Valorar la calidad de los resultados obtenidos en términos de las correspondencias válidas observadas por inspección ocular y las tendencias de las líneas dibujadas.

Fuerza Bruta + Cross Check:

A grandes rasgos podemos decir que hay correspondencias correctas y correspondencias erróneas, aunque en esta muestra parecen abundar más las erróneas. Esto se puede justificar por la presencia de líneas entrecruzadas que no muestran una tendencia concreta. Además, podemos añadirle el hecho de que en la imagen de la izquierda aparecen correspondencias en puntos que ni siquiera están en la imagen derecha, esos indudablemente serán incorrectos.

No obstante, aunque pocas, también hay algunas correspondencias correctas. En principio, son aquellas que sí muestran una tendencia: son paraleleas entre sí. Y digo en principio porque se da el caso de algunas líneas paralelas que representan correspondencias equívocas (por ejemplo, en la zona de bosque).

En definitiva, para esta muestra, podríamos concluir que los resultados del algoritmo no son de mucha calidad.

Lowe Average 2NN:

A diferencia del caso anterior, ahora sí observamos que todas las líneas siguen una tendencia concreta y prácticamente, todas ellas son correctas. Podemos concluir que este algoritmo, para esta muestra, presenta una gran calidad.

(c) Comparar ambas técnicas de correspondencias en términos de la calidad de sus correspondencias (suponer 100 aleatorias e inspección visual).

Si realizamos la comparación teniendo en cuenta las imágenes del apartado (a), la respuesta está clara: Lowe Average 2NN presenta mejores resultados.

Esta conclusión resulta obvia, al fin y al cabo, es una mejora del algoritmo de fuerza bruta: en lugar de quedarnos únicamente con el punto más cercano, KNN con k=2 busca, utilizando el criterio de fuerza bruta, los dos vecinos más cercanos a cada punto. Después, aplicamos el ratio de D. Lowe: si las distancias entre el vecino más cercano y el segundo más cercano son muy parecidas, es señal de que no es una buena correspondencia. En un buen match, el punto más cercano tiene que ser significativamente más cercano que el resto (concretamente, Lowe sugiere que la distancia del más cercano, debe ser menor que el 70 % del segundo más cercano). Con este ratio, nos quedamos con los mejores matches, descartando los malos. Como evidencian los resultados, esto mejora considerablemente las correspondencias.

No obstante, para ser más justos con la comparación, aquí se muestran dos imágenes que reflejan las 100 mejores correspondencias de ambos algoritmos (considerando que las mejores correspondencias son las que presentan menor distancia entre sus puntos):

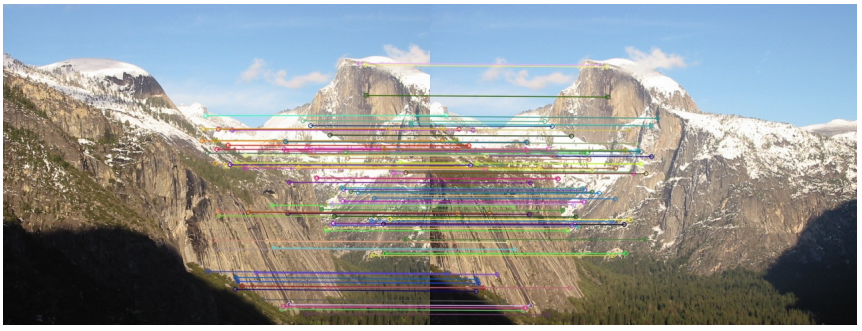


Figura 7: Fuerza Bruta + Cross Check: 100 mejores

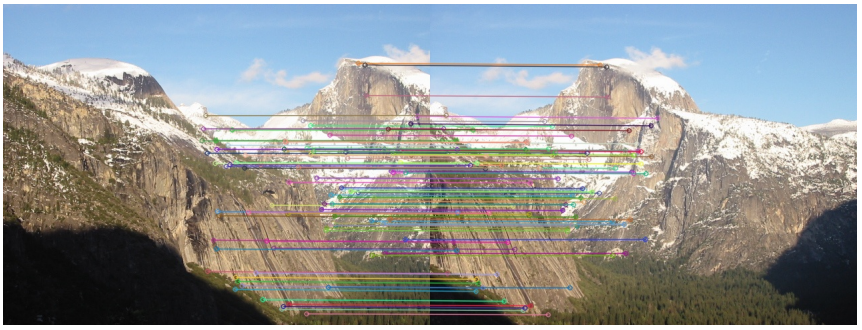


Figura 8: Lowe Average 2NN: 100 mejores

Ahora sí, en ambos casos, la líneas siguen una tendencia clara y las correspondencias son correctas.

En definitiva, aunque el filtrado de Lowe mejore y mucho los resultados, aplicando sólamente fuerza bruta con cross check, también somos capaces de obtener buenas correspondencias para estimar la homografía.

Nota: También podrían haberse mostrado los resultados que da KNN sin aplicar el ratio de Lowe y mostrando sólamente el vecino más cercano, pero eso sería equivalente a una fuerza bruta sin cross check y no me parece muy apropiado para la comparación que pide el 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: Usar el flag BORDER TRANSPARENT de warpPerspective).

Para este ejercicio, se ha realizado una función que recibe como entrada una lista de imágenes ordenadas de izquierda a derecha según su orden en el mosaico. A partir de esta lista, construye una única imagen que contiene el panorama.

El proceso seguido es el siguiente:

(1) Crear la imagen que contendrá el resultado: como no sabemos la disposición que tendrá, debemos reservar suficiente espacio. Crearemos una imagen que tendrá como alto la altura de la primera imagen más 400 píxeles extra y como ancho, la suma del ancho de todas las imágenes más 400 píxeles.

(2) Hallar las correspondencias entre cada par de imágenes: si suponemos N=3, habrá que calcular 2 correspondencias: la primera imagen con la segunda y la segunda con la tercera. Para ello, llamamos a la función construida en el ejercicio anterior, utilizando el criterio Lowe Average 2NN (también podría haberse utilizado el otro) y pidiendo que nos devuelva las 100 mejores correspondencias junto con los keyPoints de ambas imágenes.

(3) Preparar los puntos para findHomography(): tras la llamada a la función del paso anterior, por cada par de imágenes, tenemos los keyPoints de cada una de ellas junto con las correspondencias. No obstante, el método de OpenCV findHomography() requiere como parámetros los puntos en el plano original (los keyPoint de la primera imagen del par) y los puntos en el plano destino (los keyPoint de la segunda imagen del par). Por tanto, en este paso, utilizando los keyPoints y sus correspondencias, extraemos las dos listas de puntos necesarias.

(4) Hallar las homografías: por cada par de imágenes (y ultilizando las parejas de puntos del paso anterior) extraemos la homografía utilizando findHomography().

(5) Invertir las homografías halladas: tras el paso anterior (y suponiendo N=3), tenemos las homografías de 1 a 2 y de 2 a 3. No obstante, para construir el mosaico, necesitamos las homografías de 3 a 2 y de 2 a 1 (por ejemplo, necesitamos saber que transformación

aplicarle a la imagen 3 para que encaje con la 2). Para hallarlas basta con invertir las que ya tenemos.

(6) Transladar al plano la primera imagen: la colocaremos centrada y un poco a la izquierda. Para ello creamos nuestra propia homografía de translación:

$$\begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix}$$

Utilizaremos $t_x = t_y = 200$. Llevamos la imagen al mosaico con warpPerspective().

(7) Transladar al mosaico el resto de imágenes: Para ello utilizamos warpPerspective() y las homografías invertidas ya calculadas. Además, no debemos olvidar ir "acumulando" las homografías ya aplicadas multiplicando las matrices. Si no lo hacemos, la transformación que apliquemos no tendrá en cuenta las tranformaciones aplicadas previamente y el resultado no será correcto.

(8) Eliminar la parte sobrante del mosaico.

Dicho esto, aquí se muestran ejemplos del funcionamiento de la función construyendo mosaicos de tres imágenes con los archivos de Yosemite:

4. Lo mismo que en el punto anterior pero para N > 5 (usar las imágenes para mosaico).

   Para este ejercicio se ha utilizado la misma función que en el apartado anterior. A continuación se muestra un mosaico con 10 imágenes:

Bonus:

3. Implementar de forma eficiente la estimación de una homografía usando RANSAC.

   El algoritmo RANSAC implementado recibe las correspondencias (puntos fuente y puntos destino) y un umbral que representa la distancia máxima a la que se deben encontrar el punto determinado por la homografía y el punto destino para ser considerado un inlier. Devuelve la mejor homografía estimada.

   En el interior del algoritmo se realizan los siguientes pasos de forma iterativa:

   (1) Elegir 4 correspondencias aleatorias (4 es lo mínimo necesario para poder estimar una homografía).

   (2) Estimar una homografía con esas 4 correspondencias: para ello utilizamos el algoritmo "Direct Linear Transformation", que explicaremos más adelante.

   (3) Contar el número de inliers para la homografía estimada. Para ello realizamos el siguiente proceso: para cada correspondencia (p_1, p_2) calculamos $p_3 = H \cdot p_1$. Si la distancia euclídea entre entre p_2 y p_3 es menor que el umbral dado como argumento, entonces el punto es un inlier, si no, es considerado un outlier.

   Nota: para poder realizar estas operaciones los puntos deben de expresarse como: $\begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix}$.

   Además, tras realizar $H \cdot p$, tenemos el punto como $\begin{bmatrix} p_x \\ p_y \\ k \end{bmatrix}$, por lo que será necesario

dividir por k para que nos quede de la forma $\begin{bmatrix} p_x/k \\ p_y/k \\ 1 \end{bmatrix}$.

(4) Comprobar si ha sido la mejor homografía hasta el momento. Para ello utilizamos el número de inliers para esa H. Si se ha obtenido el mayor número de inliers, esa H será la mejor hasta el momento.

Se han fijado un total de 500 iteraciones (para que no tarde demasiado). Tras ellas, se devuelve la mejor homografía encontrada.

Falta por explicar el algoritmo que se usa para estimar la homografía a partir de 4 correspondencias: Direct Linear Transformation. El proceso es sencillo:

(1) Creamos la matriz A a partir de las correspondencias: Por cada pareja de puntos (p, p') obtenemos las siguientes dos ecuaciones:

$$\begin{bmatrix} x & y & 1 & 0 & 0 & 0 & -x'x & -x'y & -x' \\ 0 & 0 & 0 & x & y & 1 & -y'x & -y'y & -y' \end{bmatrix}$$

(2) Realizar la descomposición en valores singulares de A:

$$UDV^T = A$$

(3) La matriz H será el último vector propio de V^T, expresado como una matriz $3x3$.

Para ejemplificar su funcionamiento, aquí se muestra un mosaico construido utilizando la implementación propia para estimar la homografía, en lugar de la proporcionada por OpenCV:

El resultado es muy parecido al del ejercicio 3.