



**UNIVERSIDAD
DE GRANADA**

**Práctica 3.
Detección de puntos relevantes
y
Construcción de panoramas**

Lukas Häring García

December 22, 2019

Tabla de contenidos

1 Primer Apartado	2
1.1 Detector de puntos Harris	2
1.2 Apartados	3
1.2.1 Puntos de Harris según el umbral	3
1.2.2 Resultados	4
1.2.3 Valoración	5
1.3 Puntos de Harris y subpixels	5
1.3.1 Resultados	6
1.3.2 Valoración	6
2 Segundo Apartado	7
2.1 Detectar y extraer los descriptores AKAZE	7
2.2 BruteForce + crossCheck	7
2.2.1 Resultados	8
2.2.2 Valoración	8
2.3 Lowe-Average-2NN	9
2.3.1 Resultados	9
2.3.2 Valoración	10
3 Tercer Apartado	11
3.1 Mosáico de 2 Imágenes	11
3.1.1 Resultados	12
3.1.2 Valoración	12
4 Cuarto Apartado	13
4.1 Mosáico de N Imágenes	13
4.1.1 Resultados	14
4.1.2 Valoración	14
5 Bonus	15
5.1 Ransac	15
5.1.1 Resultados	16

1 Primer Apartado

El primer ejercicio consiste en la detección de puntos Harris sobre imágenes, que vulgarmente conocemos como detector de esquinas. Estas esquinas son utilizadas para inferir características de una imagen , además hace uso de una imagen en escala de grises lo cual facilita el cómputo al no tener que pasar por cada canal.

1.1 Detector de puntos Harris

En mi código el algoritmo está implementado con el nombre de *ejercicio1*, esta función hace uso de diferentes funciones y está estructurada de la siguiente manera:

1. **Relleno de la imagen.** Se rellena la imagen de ceros a la derecha y abajo de la original. Se llenarán tantos ceros a la derecha y debajo hasta alcanzar la **potencia de 2** más cercana al alto y al ancho de la imagen original. Así conseguiremos evitar cualquier tipo de pérdida para cada escala. La función recibe el nombre de *padding_image* y solo necesita como argumento la imagen original.
2. **Cálculo del gradiente.** Se ha calculado la derivada en "x" y la derivada en "y" sobre la imagen original utilizando la función de OpenCV *cv2.Sobel*, cuya máscara será creada con un $\sigma = 4.5$, como se indica en el enunciado.
3. **Espacio de escalas.** Vamos a utilizar diferentes espacios de escala para las derivadas y la imagen con el relleno. Para ello voy a utilizar una función propia llamada *gaussian_scales*, que requiere como argumentos la imagen y el número de escalas. Esta función hace uso de la función de OpenCV *cv2.pyrDown*, esta función se encargará de suavizar y redimensionar la imagen que le pasemos. Debemos comentar que el gradiente en x y el gradiente en y tras varias pruebas no se ven alterados al aplicar este método.
4. **cornerEigenValsAndVecs por escala.** Para cada escala creada anteriormente sobre la imagen original, vamos a aplicar la función de OpenCV **cornerEigenValsAndVecs**, este método nos devolverá para cada pixel, los autovalores y los autovectores para la detección de bordes, la función es llamada *harris_scales* y tiene como parámetros el espacio de escala gauiano y el *blocksize*. Los parámetros utilizados son, $\sigma = 1$ que utilizará este método para suavizar, *blockSize* = 5 que representa la región de vecindario al que va a calcular la matriz de covarianza de derivadas, este valor es constante ya que lo que vamos a reducir es la imagen, por lo que esquinas más grandes se encontrarán en escalas más tardías. Finalmente, para cada pixel, se va a calcular la media harmónica de los autovalores $\frac{\lambda_1 \cdot \lambda_2}{\lambda_1 + \lambda_2}$.

También comentar que *cornerEigenValsAndVecs* acepta la imagen en formato *float* o *uint8*, en mi caso, he utilizado *uint8*, no habrá ninguna diferencia en cuanto al resultado, pero los vamores del umbral serán números más pequeños.

5. **Supresión de no-máximos y filtro.** Una vez obtenida para cada pixel la media harmónica anteriormente explicada, vamos a realizar una supresión de no-máximos y un filtro de dicha media, para así eliminar aquellos "pixels" que no sean tan interesantes. Por cada "pixel" que pase satisfactoriamente estas condiciones, devolverá para su correspondiente escala un objeto de OpenCV "KeyPoint", este elemento almacena la siguiente información: La posición sobre la imagen original ($x \cdot 2^n, y \cdot 2^n$), siendo n el número de escala, el diámetro de la circunferencia, que es calculado de la siguiente forma $((k + 1) \cdot \text{blockSize}) \cdot 2.0$ y finalmente el ángulo de la esquina, que es calculada utilizando el gradiente de la siguiente forma $\arctan\left(\frac{dy}{dx}\right)$. Es por ello que uno de los argumentos será el espacio de escalas del gradiente, por lo que para la escala del pixel actual, miraremos en el la escala del gradiente en la misma posición. Finalmente como dicho ángulo tiene que estar en grados, utilizaremos el método de conversión a grados de Numpy *np.degrees*, todo esto está implementado en la función *maximums_harris_scales* y recibe como parámetros, el espacio de escalas de la imagen, del gradiente en "x", del gradiente en "y", el tamaño de bloque utilizado anteriormente y el umbral de la media harmónica.
6. **Supresión de pixeles cercanas a la frontera.** Como se ha utilizado padding, las regiones cercanas a las fronteras han creado puntos indeseables, por lo que se ha implementado una función llamada *suppress_near_border*, el cual recibe la imagen y la distancia a la frontera.

1.2 Apartados

1.2.1 Puntos de Harris según el umbral

Está implementado como *ejercicio1_c* y recibe los resultados obtenidos con la función *ejercicio1*. He decidido que se muestren los *KeyPoints* para cada una de las escalas una al lado de la otra y en la consola se imprima la cantidad de puntos por escala y la total. Para dibujar los *KeyPoints*, se ha hecho uso del método de OpenCV, *drawKeypoints* con el flag de dibujado *cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS*, este flag activará el dibujado el ángulo del gradiente y el diámetro de los puntos.

1.2.2 Resultados

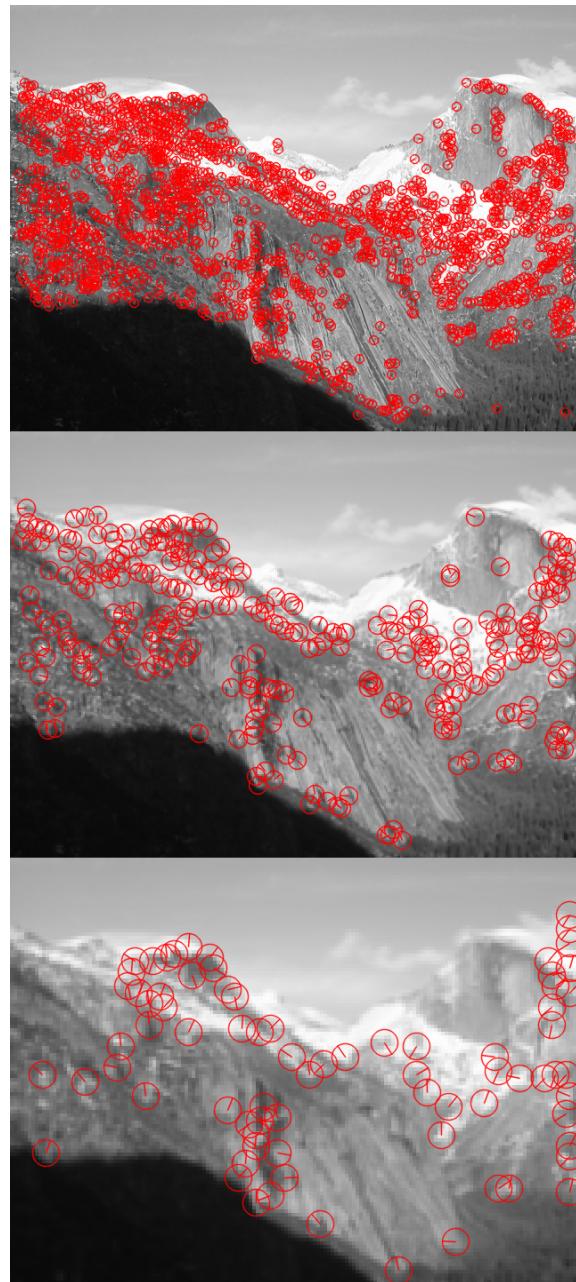


Fig. 1: **yosemite1.jpg** de 3 escalas con umbral de 0.33

1.2.3 Valoración

Se ha implementado la visualización para las diferentes escalas para que no se solapen y además porque al ir alisando, estamos desplazando píxeles por lo que los picos en las siguientes escalas no tienen por que estar proyectadas en la imagen original.

Vemos claramente que en cada escala la cantidad de esquinas va decreciendo, esto es debido a que hemos ido dejando los picos pequeños y los picos grandes se van haciendo más escasos. En la imagen original podemos observar que hay muchos *KeyPoints*, esto también es debido a ruido que puede contener la imagen y que no ha sido reducido hasta que no se ha aplicado un alisamiento.

Se ha utilizado el umbral de 0.33 de forma empírica para obtener un conjunto de 2061 puntos, a medida que vamos decrementando dicho umbral, van apareciendo más puntos.

Además, el ángulo del gradiente parece ser correcto ya que, por ejemplo, si nos fijamos en la intersección de la montaña y el cielo, estos apuntan hacia el cielo.

1.3 Puntos de Harris y subpixels

De la misma forma que el apartado anterior, también hace uso del resultado obtenido con *ejercicio1*, esta función es llamada *ejercicio1_d* y se encarga de aproximar los puntos obtenidos a las coordenadas "reales". Para ello, se escogen "N pixeles", que es pasado como argumento, de forma aleatoria utilizando el método de la librería Random, *random.sample*. Ahora, se calcula de estos puntos aleatorios, los subpixeles. Para ello, utilizamos la función de **OpenCV** *cornerSubPix*, este recibe como argumentos, la imagen; los píxeles; la ventana de búsqueda, en nuestro caso, (2, 2); la zona cero, es decir, el radio de la zona vecina donde no puede encontrarse el punto, con (-1, -1) para desactivarla y los criterios de parada, en mi caso, 100 iteraciones o un error menor al 0.01.

Vamos a dibujar en primer lugar al pixel en el centro de la imagen de color **Rojo** y para para el subpixel obtenido, lo dibujaremos de color **Verde**, la ventana será de 9x9 (ya que con 10x10 estaríamos perdiendo una fila o columna) con un zoom de x5, es decir la imagen resultante es de 45x45px. Además, como se eliminaron los píxeles cercanos a la frontera, estas regiones estarán siempre dentro de nuestra imagen.

1.3.1 Resultados



Fig. 2: **yosemite1.jpg** de 3 escalas con umbral de 0.33

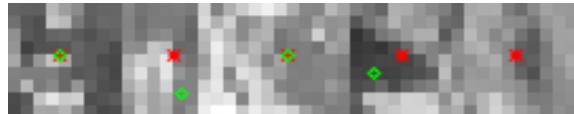


Fig. 3: **yosemite3.jpg** con umbral de 0.33



Fig. 4: **Tablero1.jpg** con umbral de 0.33

1.3.2 Valoración

Se han mostrado para cada imagen, 5 píxeles. Podemos observar en las dos figuras "yosemite1" y "yosemite3" que es muy difícil apreciar cuáles son las esquinas, pero si que podemos observar un patrón casi triangular en ellas, por ejemplo en "yosemite1", el último pixel o en "yosemite3", en el penúltimo pixel. Para ello he elegido además, la imagen "Tablero1", ya que presenta de forma más clara las esquinas, por ejemplo los tres primeros píxeles, se ve claramente el pixel rojo, y el subpixel verde en la esquina.

También podemos observar que en todas las imágenes hay píxeles donde el subpixel no se ha movido, esto puede ocurrir cuando como se comentó antes, el suavizado ha desplazado los píxeles y se han proyectado en vez de en una esquina, en el interior tras escalarlo hacia la original, véase la imagen "Tablero1" las dos últimas, lo cual por ejemplo una solución podría ser utilizar únicamente la imagen original o evitar el uso de padding.

2 Segundo Apartado

En este ejercicio vamos a utilizar un detector y extractor de KeyPoints, nosotros vamos a utilizar AKAZE, que es una versión optimizada de KAZE. Una vez obtenidos estos KeyPoints vamos a utilizar un Matcher,

2.1 Detectar y extraer los descriptores AKAZE

Como vamos a utilizar AKAZE, se ha creado una función que lo crea, necesitando como parámetro ambas imágenes para extraer sus KeyPoints y nos lo devuelve junto a sus descriptores, llamada *create_akaze*. Esta función hace uso de *cv2.AKAZE_create()*, para crearlo, además, podemos utilizar un threshold para aceptar un punto si fuera necesario, el cual yo no lo he utilizado. También hace uso de *detectAndCompute*, este necesitará como argumento: una de las imágenes y ninguna (None) máscara de operación y devuelve los KeyPoints y los descriptores de la imagen.

Para pintar las correspondencias en ambas imágenes, se han creado dos funciones de dibujado, una para correspondencias simples (*draw_matches*) y otro para las compuestas (*draw_matches_knn*), ambas escogen un número aleatorio de *KeyPoints* a dibujar (100 según el enunciado), utilizan las funciones de **OpenCV** ”*drawMatches*” y ”*drawMatchesKnn*” respectivamente, con el flag *DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS*, para que dibujen únicamente aquellas con correspondencia en la otra imagen.

En ambos ejercicios utilizaremos para el ”matcher” **BFMatcher** el parámetro *cv2.NORM_HAMMING*, debido a que AKAZE utiliza la norma de HAMMING para optimizar su cálculo[2].

2.2 BruteForce + crossCheck

En este apartado, vamos a utilizar BFMatcher, es decir un ”Matcher” de Fuerza bruta, eso quiere decir que va a encontrar la mejor correspondencia (no quiere decir que sea la correcta). Además, requiere de un parámetro, *crossCheck=true*, esto quiere decir que para un par (i, j) tal que para un i-ésimo descriptor, el j-ésimo descriptor es el más cercano y vice-versa.

Una vez obtenido los descriptores, los keypoints y el matcher, podemos realizar un match de los descriptores, para ello utilizaremos el método del **BFMatcher**, *match* que necesita la lista de descriptores de ambas imágenes.

Finalmente, se hace uso de la función anteriormente explicada *drawMatches*.

2.2.1 Resultados



Fig. 5: `yosemite1.jpg` matching con `yosemite2.jpg`

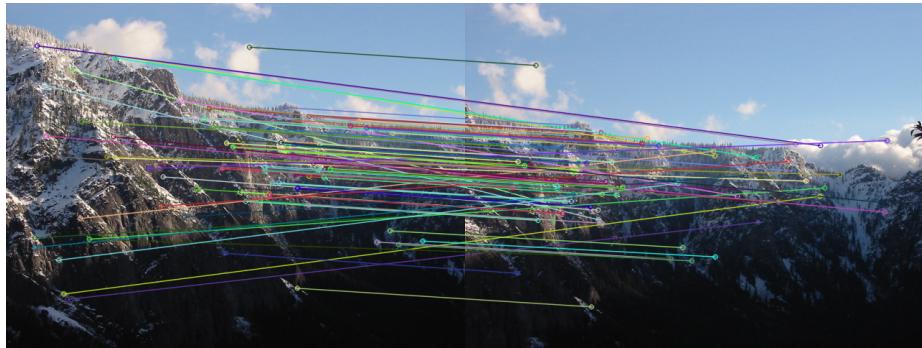


Fig. 6: `yosemite6.jpg` matching con `yosemite7.jpg`

2.2.2 Valoración

Con esta técnica de matching vemos algunas correspondencias claras, por ejemplo en la imagen superior se observa que encuentra las correspondencias del pico y de la colina de la parte derecha de la imagen. De la imagen inferior, vemos que la nube, y los píxeles del bosque también lo hacen.

El problema que tiene este criterio es la gran cantidad de malas correspondencias en relación con las buenas observadas visualmente, esto podemos asegurarlo ya que las líneas se cortan entre ellas.

Aunque elegimos aleatoriamente correspondencias, vemos que aparecen siempre bastantes positivas, solo faltaría un criterio que fuera capaz de eliminar las malas y es esto lo que vamos a ver ahora.

2.3 Lowe-Average-2NN

De la misma forma que el apartado anterior, crearemos el descriptor AKAZE, calcularemos los KeyPoints y sus descriptores utilizando `create_akaze` y crearemos nuestro "matcher" **BFMatcher**, pero esta vez, no usaremos `crossCheck`, por lo que estará a *False*. Ahora realizaremos el matching de los descriptores, pero para poder optar a dos posibles matches, utilizaremos esta vez `knnMatch`, con los descriptores y el parámetro $k = 2$, este nos devolverá una lista de los dos posibles matches con la distancia más cercana al punto posible.

Finalmente es cuando aplicaremos el criterio de Lowe a los pares de matches, este criterio realiza para cada par de matches, el ratio del mas cercano entre el segundo más cercano, si esta fracción sobrepasan un umbral, son descartados, por el contrario, nos quedamos con el primero (el más cercano).

2.3.1 Resultados

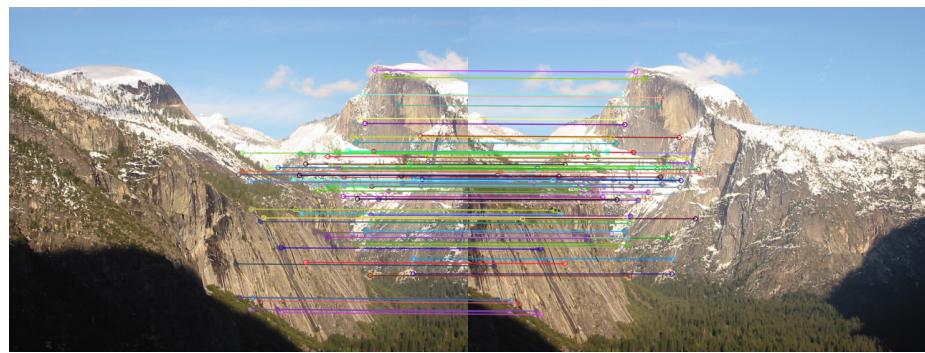


Fig. 7: `yosemite1.jpg` matching con `yosemite2.jpg`

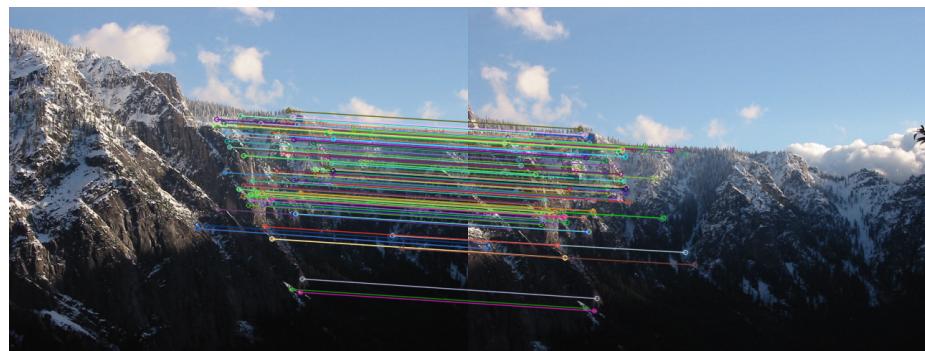


Fig. 8: `yosemite1.jpg` matching con `yosemite2.jpg`

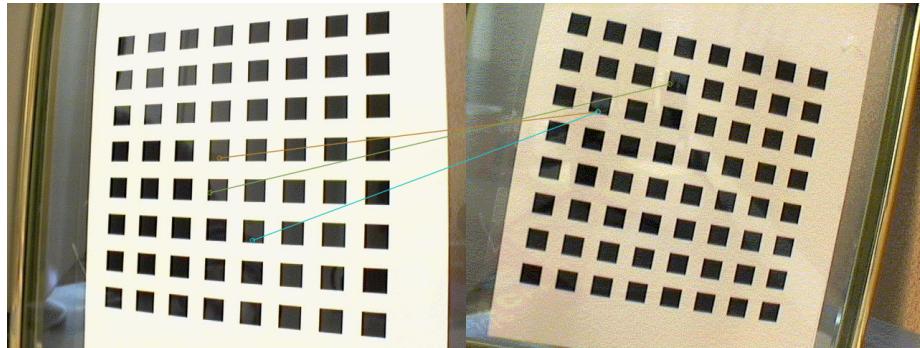


Fig. 9: Tablero1.jpg matching con Tablero2.jpg

2.3.2 Valoración

Los resultados obtenidos arriba han sido obtenidos aplicando un umbral de 0.55. Como vemos, este criterio es muy fuerte, para las imágenes 1 y 2 del apartado anterior, vemos que todas las correspondencias que aparecen son buenas, aunque no podemos asegurar que estén todas las del conjunto de correspondencias. Aseguramos que son buenas ya que las líneas son paralelas entre sí y por inspección ocular, se corresponden.

Cuanto menor sea este umbral, menos correspondencias habrán ya que esta relación de distancias es cada vez más estricta.

Por último comentar que este criterio es bueno pero no perfecto, por ejemplo en la tercera imagen del apartado anterior, falla catastróficamente ya que no tiene en cuenta información total de la imagen.

3 Tercer Apartado

En esta sección aprenderemos a formar un mosaico de dos imágenes, utilizaremos los KeyPoints encontrados en el ejercicio anterior y sobre estos, encontraremos una homografía que sea capaz de "unir" ambas imágenes.

Se ha creado una función básica que devuelve una homografía de translación compuesta con otra de escalado, llamada *homography_matrix* y recibe como parámetros, la translación en "x" e "y" y el scalado *s*.

Por otro lado, otra función llamada *get_homography_matrix* que requiere ambas imágenes como parámetros y si es necesario, el umbral de Lowe. Este método llama a la función creada en el apartado anterior *ejercicio2_lowe*, ya que da buenas correspondencias y genera dos listas de posiciones (otra lista [x, y]) de los matches obtenidos, creando uno para la primera imagen *k_from* y otra para los puntos de la segunda imagen *k_to*. Finalmente, devuelve una homografía a partir de dichas listas, calculada con la función de OpenCV *cv2.findHomography*, además de los parámetros *cv2.RANSAC* para activar el método iterativo con un umbral de error de 1 pixel.

3.1 Mosáico de 2 Imágenes

Este método está declarado con el nombre *ejercicio3*, que recibe como argumentos ambas imágenes. Además, supondremos que existe una Homografía que pueda solapar ambas imágenes.

En primer lugar, vamos a crear una imagen negra donde iremos posicionando cada imagen, de tamaño 1500x700, la primera, irá al centro, para ello vamos a hacer uso de la función anteriormente descrita *homography_matrix*, como argumentos, $tx = \frac{\text{ancho} - 750}{2}$, $ty = \frac{\text{alto} - 250}{2}$ y la escala = H_t , la deseada. Utilizaremos el método de OpenCV *cv2.warpPerspective*, para dibujar la imagen pasada como primer argumento sobre el canvas creado. Los argumentos son: La imagen, La homografía creada de translación y escalado, un par (ancho, alto) del canvas creado, la imagen de destino "*dst*" es decir el canvas y finalmente el flag "*boderMode*" a *cv2.BORDER_TRANSPARENT* que funciona como una máscara, pudiéndose así dibujar una imagen al lado de la otra, **además es importante añadir el parámetro *dst***, si no, aparecerá ruido. Ahora vamos a utilizar *get_homography_matrix* para obtener la homografía de la segunda imagen sobre la primera (*get_homography_matrix(img2, img1)*) = $H_{2,1}$, es importante este orden, aunque si lo hicieramos al revés, deberíamos calcular su inversa. Componemos la homografía obtenida con la homografía anterior utilizando el producto de matrices, $H_t \cdot H_{2,1}$, con Numpy, podemos utilizar el operador @ como multiplicador de matrices. Finalmente, utilizamos de nuevo *cv2.warpPerspective* de la misma forma que antes pero con la composición de las homografías.

3.1.1 Resultados



Fig. 10: mosaico **yosemite1.jpg** y **yosemite2.jpg**

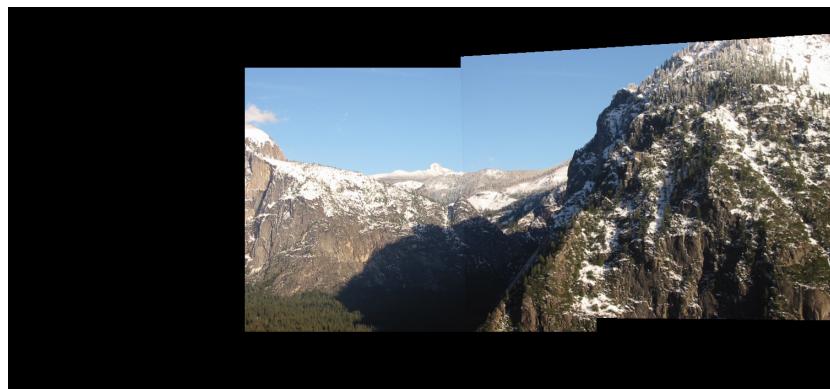


Fig. 11: mosaico **yosemite3.jpg** y **yosemite4.jpg**

3.1.2 Valoración

Vemos que para ambas imágenes las homografías son muy buenas, estas hacen que se solapen todos los puntos de forma muy precisa. El único problema observable a simple vista es la iluminación de cada imagen, pero este problema está fuera de nuestro alcance. Otra cosa que podemos observar en el segundo mosaico que no vemos en el primero, es el efecto de sesgo que sufre la imagen de la derecha que no, mientras que el borde inferior parece mantenerse totalmente horizontal. Aunque a simple vista no es observable, esta imagen que hemos comentado se ha visto estirada lo cual la ha deformado para poder encajar en el mosaico.

4 Cuarto Apartado

En este último apartado, vamos a resolver el problema anterior de forma más general, es decir, vamos a realizar un mosaico de n imágenes, lo importante es que la primera imagen tiene que poder solaparse con la siguiente y así sucesivamente, además, vamos a suponer que todas las imágenes son del mismo tamaño.

Para la realización de este ejercicio, vamos a hacer uso de una función con el nombre *homographies_queue*, recibirá una lista de imágenes, un desplazamiento en x e y, un escalado y el ratio de Lowe para el ejercicio 2. Cabe destacar que el procedimiento es equivalente al anterior, se realizará una homografía de traslación y escalado H_t (*homography_matrix(x, y, s)*) para la inicial y se irá componiendo con la homografía de la imagen actual con la anterior $H_{i,i-1}$ (*get_homography_matrix(imgs[i], img[i - 1])*), es decir para el caso inicial $H_t \cdot H_{1,0}$, para el siguiente $H_t \cdot H_{1,0} \cdot H_{2,1}$ y así sucesivamente hasta acabar con una lista de tantas homografías como imágenes haya.

4.1 Mosáico de N Imágenes

La función es llamada ”*ejercicio4*”. En primer lugar, vamos a crear una imagen de tamaño $1500x700$ de color negro donde iremos posicionando para cada iteración la siguiente imagen del mosaico.

Generaremos de la misma forma que en el ejercicio anterior, la homografía de traslación en el centro de la imagen. Haremos uso de la función anteriormente definida como *homographies_queue*, esta nos devolverá la lista ya mencionada de homografías, finalmente solo debemos iterar para cada imagen e ir aplicando la homografía i-ésima, ya que cada una ha sido compuesta con la anterior.

Utilizaremos *cv2.warpPerspective* cuyos argumentos son *imgs[i]*, *homographies[i]* y ”*dst=compose_image*” y los demás argumentos como el ejercicio anterior. El valor que nos retornará será también *compose_image* para así combinar las imágenes de forma recursiva.

4.1.1 Resultados

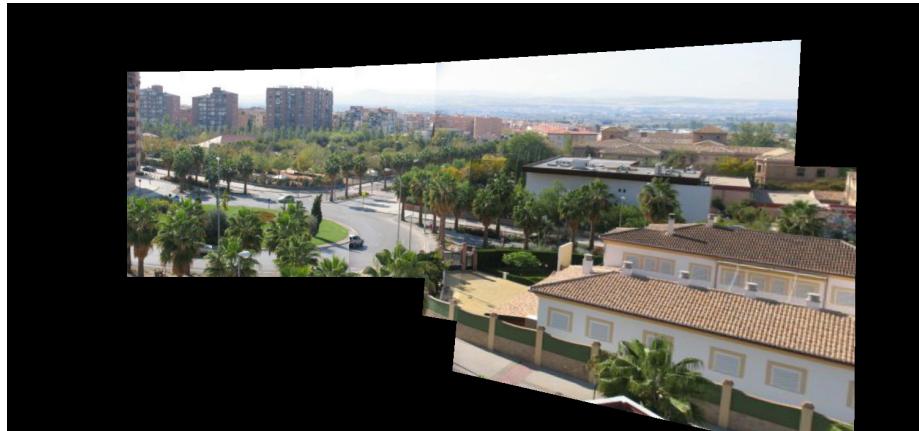


Fig. 12: mosaico mosaico(2-11).jpg

4.1.2 Valoración

En esta mosaico podemos observar claramente algunos desperfectos, pero que también están fuera de nuestro alcance para solucionar, como por ejemplo, el coche aparece incompleto, el edificio rojo del fondo aparece también recortado y algunas imágenes del mosaico tienen diferente iluminación.

Otra aspecto que hemos tratado antes pero no era tan perceptible era como se estiraban las imágenes, aquí claramente se observa que el edificio de la derecha parece que tiene una mayor escala que los demás elementos de la imagen, además, la imagen encima tiene forma de reloj de arena, realmente la imagen tiene el lado derecho fijo y el otro lado está rotado hacia dentro.

5 Bonus

Ransac es un método iterativo capaz de realizar una estimación robusta de los parámetros de un modelo y con un alto grado de precisión.

5.1 Ransac

Vamos a utilizar la misma plantilla que para el ejercicio 3, pero esta vez, vamos a estimar nuestra propia homografía H utilizando la técnica de RANSAC. Para ello se ha creado la función `ransac` que recibe como argumentos los keypoints x de la imagen 1 y los keypoints x' de la imagen 2 y el threshold, es decir, la distancia máxima hasta la cual se aceptará un punto. Formalmente $H \cdot x = x'$. La función `ransac` va a realizar un pre-procesado de los datos, transformando las coordenadas 2D en una coordenada del plano afín $(x, y, 1)$.

El algoritmo en pocas palabras: toma 4 puntos aleatorios `random.sample(n, 4)`, calcula una homografía `find_homography` y cuenta cuantos puntos del conjunto total aplicando dicha homografía están cerca (según un umbral) del punto deseado, se sumará 1 por cada "punto cercano", como es un proceso que se repite varias veces, se quedará con la homografía actual si suma más 1's que la mejor hasta la iteración actual, cuando acaba la última iteración, se devuelve la mejor homografía encontrada.

La función `find_homography` tiene como argumentos, la lista de puntos $X = [(x_0, y_0), \dots]$ y la lista de puntos en correspondencia $X' = [(x'_0, y'_0), \dots]$. Se genera la matriz en python.

$$H = \begin{pmatrix} -x_0 & -y_0 & -1 & 0 & 0 & 0 & x'_0 \cdot x_0 & x'_0 \cdot y_0 & x'_0 \\ 0 & 0 & 0 & -x_0 & -y_0 & -1 & y'_0 \cdot x_0 & y'_0 \cdot y_0 & y'_0 \\ -x_1 & -y_1 & -1 & 1 & 1 & 1 & x'_1 \cdot x_1 & x'_1 \cdot y_1 & x'_1 \\ 1 & 1 & 1 & -x_1 & -y_1 & -1 & y'_1 \cdot x_1 & y'_1 \cdot y_1 & y'_1 \\ -x_2 & -y_2 & -1 & 2 & 2 & 2 & x'_2 \cdot x_2 & x'_2 \cdot y_2 & x'_2 \\ 2 & 2 & 2 & -x_2 & -y_2 & -1 & y'_2 \cdot x_2 & y'_2 \cdot y_2 & y'_2 \\ -x_3 & -y_3 & -1 & 3 & 3 & 3 & x'_3 \cdot x_3 & x'_3 \cdot y_3 & x'_3 \\ 3 & 3 & 3 & -x_3 & -y_3 & -1 & y'_3 \cdot x_3 & y'_3 \cdot y_3 & y'_3 \end{pmatrix}$$

Según la diapositiva, se calcula la descomposición en valores singulares UVW^t `np.linalg.svd` y nos quedamos con la columna cuyo autovalor es menor, `np.linalg.svd` nos devuelve los autovalores de mayor a menos. Buscamos la dicha columna que represente a dicho autovalor, con Numpy, solo bastaría con coger la última fila, la octava, ya que W^t . Como el resultado está en filas, hay que convertirla en una matriz, basta con utilizar el método `np.reshape`, finalmente normalizamos la matriz dividiendo toda la matriz por la última celda.

Finalmente, con la homografía obtenida, para todo punto p en X y su correspondencia p' en X' , se calcula $\|Hp - p'\|$, esta es la distancia entre p' y hacia donde ha llevado dicha homografía el punto p , si dicho valor es menor a un umbral (Puede ser 1px, por ejemplo). Incrementamos en 1 por cada punto

dentro del umbral, si la cantidad total es mayor a la alcanzada hasta el momento, ponemos como nueva homografía la calculada en dicha iteración. Una vez acabado, devolvemos la homografía final.

5.1.1 Resultados

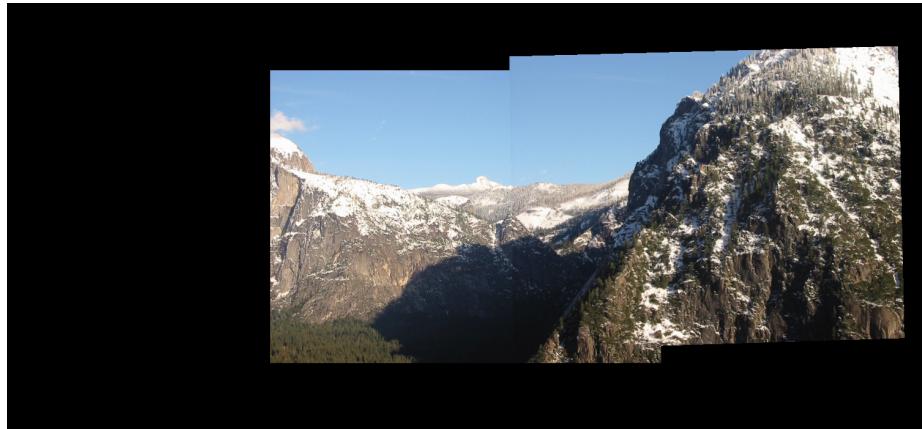


Fig. 13: mosaico bonus ”yosemite3.jpg” y ”yosemite4.jpg”

5.1.2 Valoración

No se observa ninguna diferencia visual respecto del ejercicio3, lo que significa que la homografía calculada es correcta. La única diferencia significativa es el tiempo de cómputo.

Referencias

- [1] Matrices de transformación
https://en.wikipedia.org/wiki/Transformation_matrix#Other_kinds_of_transformations
- [2] AKAZE and Hamming Distance
<https://stackoverflow.com/questions/43614497/orb-bfmatcher-why-norm-hamming-distance>
- [3] Homography stimation
https://cseweb.ucsd.edu/classes/wi07/cse252a/homography_estimation/homography_estimation.pdf