



UNIVERSIDAD
DE GRANADA



Práctica 2: Detección de Puntos Relevantes y Construcción de Panoramas

Visión por Computador

Autor:

Lugli, Valentino Glauco · YB0819879

Índice

1 Notas sobre la implementación	2
2 Extracción de regiones relevantes en un espacio de escalas	3
2.1 ¿Qué operaciones sobre la imagen original permiten fijar la imagen semilla en $\sigma = 1.6?$	3
2.2 Función para el cálculo de una escala cualquiera	4
2.3 Cálculo de las octavas	4
2.4 Detección de Puntos	9
3 Correspondencias	11
4 Construcción de un panorama rectangular	14
5 Bonus	16
5.1 Obteniendo la respuesta más alta por octava	16
5.2 Obteniendo puntos mejorados por interpolación	20
5.3 Panorama de Proyección Plana Completo	21

1. Notas sobre la implementación

- La práctica fue realizada completamente en Python haciendo uso del IDE “Spyder”, este IDE permite tener “celdas” de código que se pueden ejecutar independientemente del código que tienen antes o después, es decir, es como una celda de código de un Colab Notebook: no es necesario ejecutar todo el programa entero, se puede ir paso a paso.
- Al igual que un Colab Notebook, se han de ejecutar las celdas en orden la primera vez que se carga el fichero para tener las funciones y variables de celdas anteriores en memoria. Una vez realizado esto se pueden ejecutar las celdas en cualquier orden.
- Las celdas se delimitan con un comentario de la forma “`# %%`” y pueden ejecutarse haciendo `Ctrl+Enter` en una celda resaltada o bien haciendo clic en el ícono en la Figura 1 que está a la derecha del ícono de “Play” que ejecuta el código entero secuencialmente.
- Las celdas están organizadas de manera que la primera celda abarca todas las funciones implementadas, y luego existe una celda por cada ejercicio, de esta manera solamente es necesario ejecutar la celda inicial, denominada la celda 0 y la celda del ejercicio que se desee ejecutar, el cual se encuentra apropiadamente identificada en el código.
- También se muestra donde comienzan y terminan las funciones de cada ejercicio para facilitar la legibilidad de la práctica.



Figura 1: Ícono para ejecutar la celda seleccionada

2. Extracción de regiones relevantes en un espacio de escalas

Para la realización de este ejercicio se realizaron dos funciones principales, `lowePyramid(p_img, numOctaves, numScales, sigma0)` que recibe una imagen, un entero indicando el número de octavas a generar, el número de escalas por octava y el sigma inicial de las escalas y `getExtrema(DoG)` que recibe únicamente la pirámide de diferencias de gaussianas generadas en `lowePyramid`.

Entre las dos funciones se estaría implementando una versión básica del detector de SIFT, pero se ha separado por legibilidad.

Se utilizaron las dos imágenes provistas para el ejercicio, `Yosemite1.jpg` y `Yosemite2.jpg`.

Dentro de estas funciones se utilizaron múltiples funciones auxiliares que se describirán cuando sea pertinente hacerlo, cabe notar que quitando la operación de OpenCV para aumentar el tamaño de la imagen con `resize()`, el resto de funciones para manipular la imagen, como la convolución o el submuestreo, fueron implementadas por el autor, copiadas directamente de la práctica anterior.

2.1. ¿Qué operaciones sobre la imagen original permiten fijar la imagen semilla en $\sigma = 1.6$?

Asumiendo que una imagen de entrada por efectos de la discretización digital posee un sigma de 0.8, para fijar la imagen semilla a un sigma de 1.6 sin convolucionarla directamente se tiene que primero remuestrear la imagen al doble de tamaño, es decir, aumentarla de tamaño al doble por interpolación; por el efecto de remuestreo al aumentar el tamaño de una imagen también se aumenta en esa misma proporción el grado de alisamiento, por lo tanto el sigma de esta imagen ampliada es de 1.6, ahora esta imagen se debe de convolucionar con varias máscaras o con una más grande equivalente (Esto es en teoría, naturalmente en este detector necesitamos generar escalas con incrementos de sigma) hasta obtener un sigma de 3.2, ahora la imagen se reduce a la mitad, volviendo al tamaño original y también se reduce a la mitad el sigma, teniendo un sigma de 1.6; esto en esencia es la generación de la escala 0 de la pirámide de Lowe.

A nivel de implementación, se llama a la función `lowePyramid()` indicándole 3 octavas reales o canónicas –la octava 0 ya se toma en cuenta internamente– con un sigma inicial de 1.6 para las escalas.

Dentro de la función, luego de inicializarse los vectores que contendrán las octavas gaussianas y las octavas laplacianas se aumenta esta imagen de entrada, siendo esta la primera parte del procedimiento descrito en el párrafo anterior.

Luego, se entra en un bucle que va a generar las octavas, y se llama a la función apropiadamente nombrada `genOctave(p_img, scales, sigma0, extra = 3)` que generará una escala con una imagen semilla, una cantidad de escalas “canónicas” –aquellas donde realmente interesa detectar los puntos característicos–, en este caso son 3, un sigma inicial y las escalas auxiliares, que en este caso también son 3 por defecto.

En este caso, la primera iteración generará la octava 0, internamente la función genera dos vectores vacíos, uno para las escalas gaussianas y otro para las DoG de esas escalas; el primer miembro del vector de escalas es la imagen semilla, se añade antes de comenzar el bucle.

Dentro de este bucle se generan las siguientes escalas, primero se obtiene el sigma necesario para avanzar de la escala actual a la siguiente en nivel de alisamiento, se utiliza la función `getSigmaOct(sigma0, s, ns)` que en esencia es una adaptación de la fórmula 1 provista en la presentación de la práctica, luego se utiliza la función de la práctica previa `gaussianMask()` para obtener una máscara gaussiana de dicho sigma y luego se convoluciona la imagen con la función `convolveImage()` también de la práctica pasada para obtener la siguiente escala.

$$\sigma_s = \sigma_0 \times \sqrt{2^{\frac{2 \cdot s}{ns}} - 2^{\frac{2(s-1)}{ns}}} \quad (1)$$

Ahora, esta imagen que es la nueva escala se almacena en el vector de escalas de la octava, se realiza la diferencia entre la escala anterior y la nueva escala para obtener la diferencia de gaussianas y se almacena también; finalmente, se actualiza una variable para que ahora esta escala permita producir la escala siguiente de manera iterativa.

Una vez obtienen todas las escalas, tanto las “canónicas” como las auxiliares, se devuelven los vectores que contienen las escalas gaussianas y las escalas laplacianas.

Este paso es donde se va aumentando el sigma iterativamente, en la octava 0, se ha pasado de un sigma 1.6 inicial a un sigma 3.2 en la escala número 3.

Ahora, volviendo de esa función al bucle en `lowePyramid()`, luego de esto se utiliza la última escala canónica, que en este caso es la número 3 –contando desde 0– y se reduce a la mitad, generando ahora la imagen semilla de la siguiente escala.

Este es el paso final si fuera la octava 0, pues esta escala se reduce a la mitad, pasando de sigma 3.2 a sigma 1.6 y por lo tanto llegando al sigma deseado en el tamaño de la imagen original.

2.2. Función para el cálculo de una escala cualquiera

Como se puede notar, la función `genOctave()` funciona para cualquier escala, lo que difieren es en la imagen semilla, ya que internamente los sigmas no van a variar, pues en relación al tamaño de la imagen, el sigma inicial se mantendrá idéntico en 1.6 y las máscaras generadas serán siempre las mismas, pero en relación a la imagen original, el sigma real estará creciendo aunque las máscaras se mantengan constantes, justamente el funcionamiento esperado para esta parte del detector SIFT.

2.3. Cálculo de las octavas

Volviendo a la función `lowePyramid()` ya de manera más general se encarga de generar tanto las octavas gaussianas como las laplacianas por medio de la función `genOctave()`.

Al recibir una imagen la función `lowePyramid()`, que primero es ampliada, al entrar al bucle se le calculan las octavas gaussianas y las octavas laplacianas, siendo esta la escala 0 en la primera iteración, luego se selecciona la última escala canónica obtenida en esta iteración, se reduce a la mitad y se convertirá en la imagen semilla de la siguiente octava, esto se repetirá hasta llegar al número especificado de octavas pasado por parámetro.

Se pueden observar las distintas escalas canónicas generadas para las dos imágenes de Yosemite en las Figuras 2 y 3, para las versiones laplacianas ver las Figuras 6 y 5.

Octava 0, Escala 1, 2 y 3



Octava 1, Escala 1, 2 y 3



Octava 2, Escala 1, 2 y 3



Octava 3, Escala 1, 2 y 3

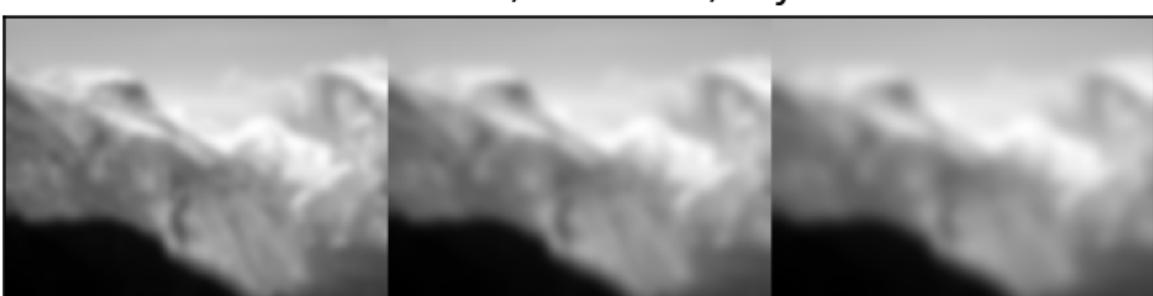


Figura 2: Octavas y Escalas Gaussianas, Yosemite1.jpg

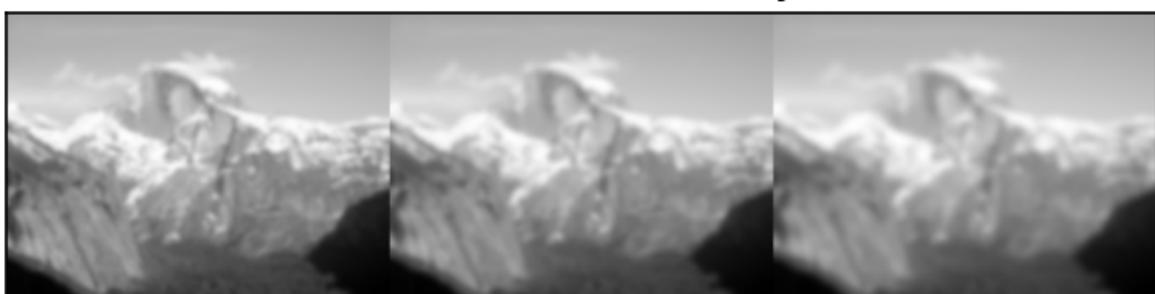
Octava 0, Escala 1, 2 y 3



Octava 1, Escala 1, 2 y 3



Octava 2, Escala 1, 2 y 3

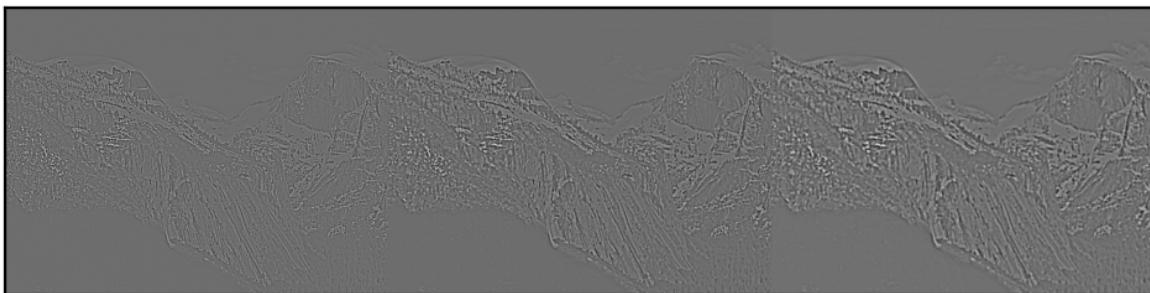


Octava 3, Escala 1, 2 y 3

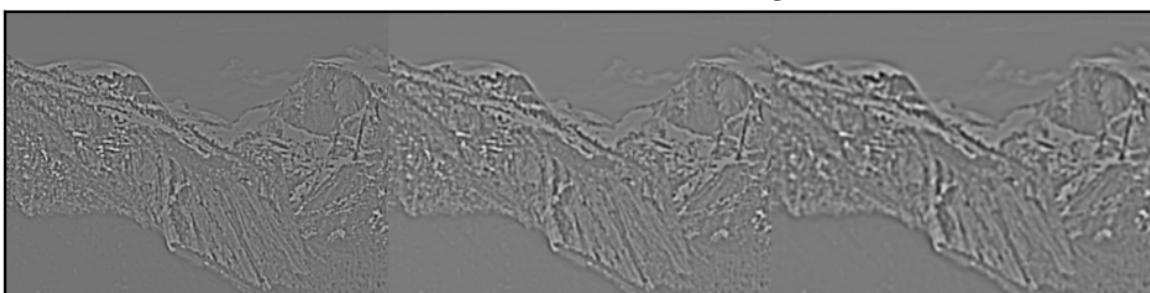


Figura 3: Octavas y Escalas Gaussianas, Yosemite2.jpg

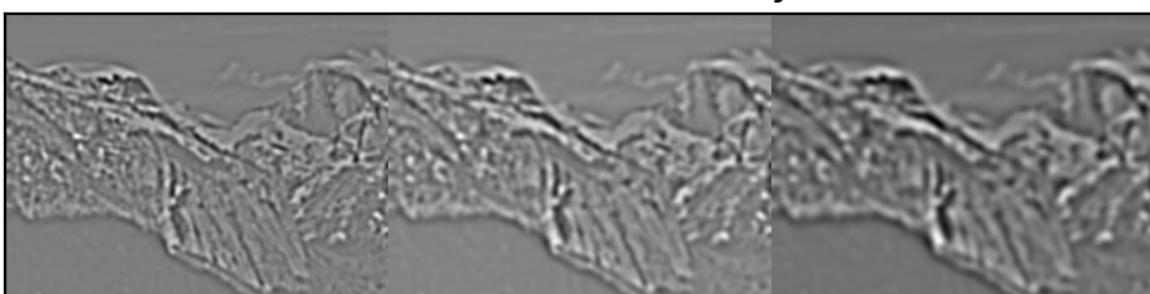
Octava 0, Escala 1, 2 y 3



Octava 1, Escala 1, 2 y 3



Octava 2, Escala 1, 2 y 3



Octava 3, Escala 1, 2 y 3

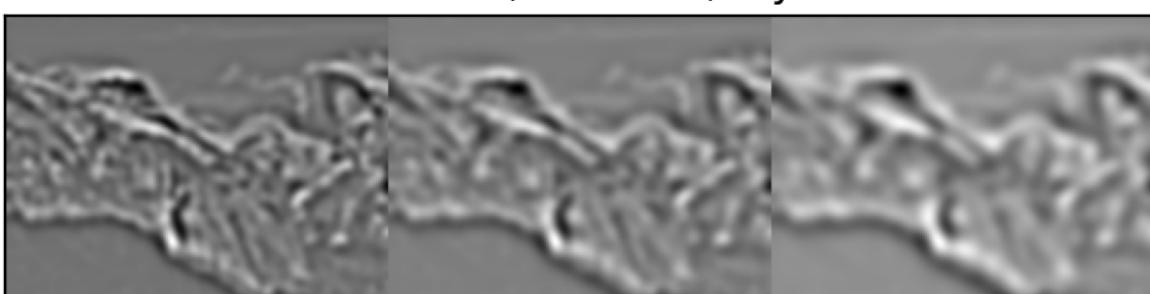
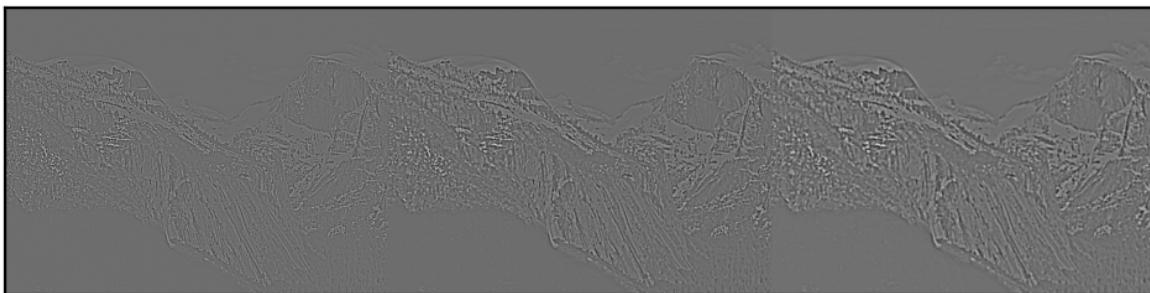
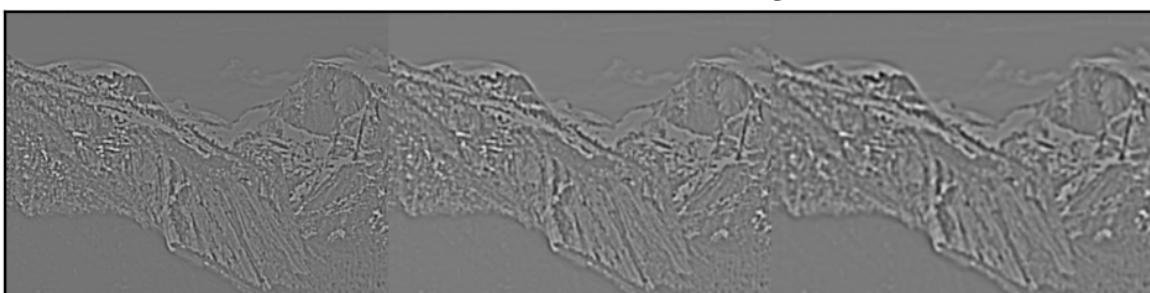


Figura 4: Octavas y Escalas Laplaciadas, Yosemite1.jpg

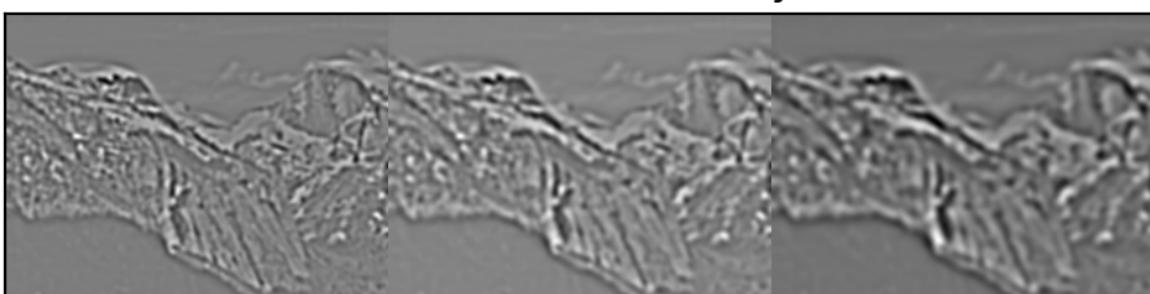
Octava 0, Escala 1, 2 y 3



Octava 1, Escala 1, 2 y 3



Octava 2, Escala 1, 2 y 3



Octava 3, Escala 1, 2 y 3

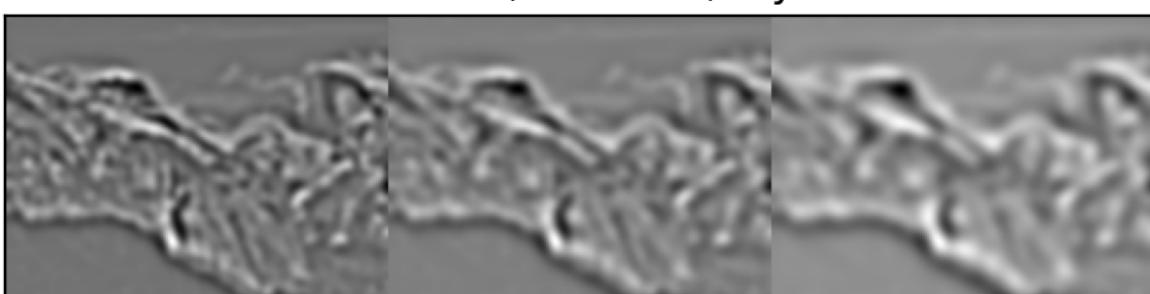


Figura 5: Octavas y Escalas Laplaciadas, Yosemite2.jpg

Se puede apreciar que, naturalmente, las imágenes empiezan a volverse más y más desenfocadas aunque se esté aplicando siempre las mismas máscaras gaussianas, validando uno de los puntos de la rapidez de SIFT, el hecho de que se pueden obtener alisamientos más y más fuertes que de otra manera requerirían unas máscaras gaussianas bastante más grandes, pero que por las propiedades de la gaussianas pueden conseguirse de una manera más eficiente y también que obtener la Laplaciana por medios de la diferencia de Gaussianas aporta a la velocidad del detector.

2.4. Detección de Puntos

Para realizar la detección de puntos se tiene una función llamada `getExtrema (DoG)` que toma como único parámetro la lista de listas que comprende la pirámide laplaciana de Lowe para poder calcular los puntos característicos, como se ha mencionado antes esto también comprende parte del detector de SIFT, que se ha separado por legibilidad.

Para realizar esto, se siguió como referencia el pseudocódigo que se encuentra en el artículo “Anatomy of the SIFT method”.

Dentro de la función, primero se generan las listas que se utilizarán para almacenar los puntos característicos, se tienen varios bucles anidados porque se tiene que recorrer cada octava, y por cada octava, las escalas canónicas (1, 2 y 3) y por cada escala, que es una imagen, se tiene que recorrer con una ventana deslizante 3×3 todos los píxeles, salvo los píxeles de los bordes, porque no encajaría la imagen: esto no afecta mucho pues los píxeles en los bordes no aportan mucha información.

¿Por qué se realiza esto? Para detectar puntos característicos, se tiene que comparar un píxel de una escala con los píxeles que tiene inmediatamente alrededor y también con los píxeles en la escala siguiente y anterior en la misma ventana 3×3 ubicada en la misma posición, es decir que se está comparando un píxel con sus 26 vecinos en una especie de “cubo” conformado por tres escalas, según el método SIFT, si este píxel es mayor o menor que todos los puntos en esta ventana $3 \times 3 \times 3$ se considera un punto “interesante” o característico y se almacena.

Por este hecho es que se han realizado las 3 escalas adicionales: esto resulta en 6 imágenes por cada octava, 3 escalas canónicas y 3 auxiliares, cuando se realiza la diferencia de Gaussianas, se tienen ahora 5 escalas teniendo dos auxiliares, ahora bien como se necesita realizar esta comparación por medio del “cubo”, para la escala 1, se tiene que comparar con la escala 0 y la escala 2, la escala 2 con la escala 1 y la 3, y la 3 con la 2 y la 4, por eso se realizaron esas escalas y por eso el bucle recorre solamente las escalas 1, 2 y 3, no tiene sentido recorrer las otras dos escalas auxiliares pues no faltaría en ambos casos otra escala con que comparar.

Dicho esto, la comparación la realiza una función llamada `isLocalExtrema(x, y, localLayer, backLayer, frontLayer)` la cual toma la posición actual del píxel, y se le pasa la escala actual, la anterior y la frontal, internamente calcula los máximos y mínimos de cada ventana 3×3 alrededor del punto y retorna un valor verdadero si el píxel que se tiene como actual es el mayor o el menor de esos valores.

Si es así, se almacena en una tupla su posición actual (x, y), la escala, la octava, el valor absoluto que tiene el píxel como tal, que vendría siendo la “respuesta” y el sigma real del mismo, el cual se calcula con la función `getRealSigma(k, octave, sigma_0=1.6)` el cual obtiene el sigma en base

a la escala y la octava en la que se encuentra el punto, específicamente, se utiliza la ecuación 2, que ha sido adaptada de la teoría, variando en el cálculo de k' , que se define como la escala k de una octava l , la expresión 3 permite obtener su valor real de manera que concuerda con los valores sigma de las escalas canónicas que se observan en la famosa gráfica del método SIFT.

$$\sigma_{\text{real}} = \sigma_0 \times 2^{\frac{k'}{ns}} \quad (2)$$

$$k' = k + (l - 1) \times 3 \quad (3)$$

Se toma el valor absoluto pues, se razona, independientemente que sea el signo positivo o negativo, lo que importa es que tan grande es el valor en sí, su respuesta.

Esta primera lista se considera en el código como una lista “plana” de puntos, plana en el sentido de texto plano porque puede verse directamente en Python, el porqué se menciona explícitamente esto tendrá sentido pronto.

Una vez finalizada la búsqueda por todas las escalas, esta lista de tuplas de ordena de mayor a menor en base al valor de la respuesta y luego se obtienen los 100 puntos con mayor respuesta.

A continuación se construye un vector de objetos `cv.KeyPoint(y, x, size)` para poder luego ser mostrados por pantalla, en la construcción del objeto se calcula cual sería la coordenada correcta en la imagen original, para ello se utiliza la función `realCoords(i, 1)` la cual dependiendo de la octava, calcula una nueva coordenada y por defecto la redondea. Notar que se escribe el constructor como si tomase las coordenadas traspuestas (y, x) , esto se debe a que lo que el autor toma como (x, y) , para el objeto es (y, x) , es un detalle de implementación que permite que los puntos se dibujen correctamente donde han sido detectados.

También se multiplica por 12 el tamaño del punto, puesto que se pide mostrar los puntos con 6σ de radio, por lo tanto, 12 de diámetro.

Esta lista se denomina la lista de objetos `KeyPoint` y como se observa, es idéntica la lista plana anterior salvo que los puntos y el sigma han sido tratados para que sean los valores reales y no los relativos, también no permite visualizar directamente los puntos, y por eso se utilizan ambas listas: la lista plana tuvo un papel fundamental a la hora de verificar y validar que los puntos fueron detectados correctamente y la lista de `KeyPoints` permite dibujarlos sobre la imagen, y en cierta forma también validar que se realizó correctamente la detección.

Una vez realizado esto se devuelve la lista de tuplas de puntos ordenados y la lista de objetos `cv.KeyPoints` que contienen los mismos puntos pero formateados para que OpenCV los entienda.

Las imágenes se dibujan con los `KeyPoints` con la función `cv.drawKeypoints(image, keypoints, outImage, flags)`, con la imagen indicada, la lista de `KeyPoints`, `None` para la imagen de salida –detalle de implementación–, y el `flags = 4` para indicar que dibuje los `Keypoints` tomando en cuenta el tamaño que poseen, lo que se denomina `RICH_KEYPOINT`.

Se pueden visualizar los puntos en las Figura 6, se puede notar que los 100 puntos con mayor respuesta tienen sentido debido a que se han detectado por lo general en áreas donde hay un fuerte cambio de iluminación, por ejemplo: en el cambio de la sombra de la montaña a una parte con iluminación más fuerte se ha detectado un punto, también en zonas que se encuentran rodeadas

de un nivel de gris contrastante, por ejemplo, puntos en la montaña que son rocosos y luego alrededor están rodeados por nieve blanca o también lo opuesto: parches de nieve rodeada de rocas más ocurras, ese cambio de iluminación que es una frecuencia alta a nivel del espacio de escala Laplaciano ha sido detectado como un punto interesante, por lo que concuerda con los resultados teóricos que deberían esperarse. También notar que de estos 100 puntos con mayor respuesta no hay ninguno en zonas de frecuencia baja, es decir, enteramente en la sombra de la montaña, o en el cielo, si bien es posible que de la cantidad total detectada existan puntos anómalos por una diferencia ligera entre píxeles vecinos que es esperable, en el top 100 no es el caso, por lo que se toma como una buena señal que el detector funciona como debe de funcionar.

Otra cosa a notar es que en general los puntos con mayor valor han sido detectados en las primeras octavas, siendo los círculos de un tamaño más pequeño, aunque si hay un par que han sido detectados en octavas mayores; esto también indica que la detección está sucediendo en todas las escalas de la pirámide.

3. Correspondencias

Se realizó el procedimiento que se describe el enunciado, se cargan dos imágenes de una montaña del parque Yosemite y, por medio de los dos detectores de fuerza bruta, uno utilizando CrossCheck y otro utilizando el criterio de Lowe, se obtienen los puntos correspondientes entre las mismas imágenes.

Para realizar eso, y por motivos de modularidad, se crearon dos funciones: `getKeyPoints_BF(img1, img2)` para utilizar el algoritmo de fuerza bruta junto con CrossCheck y `getKeyPoints_Lowe2NN(img1, img2, p_k=2)` para utilizar el criterio de Lowe con las dos mejores correspondencias de un punto.

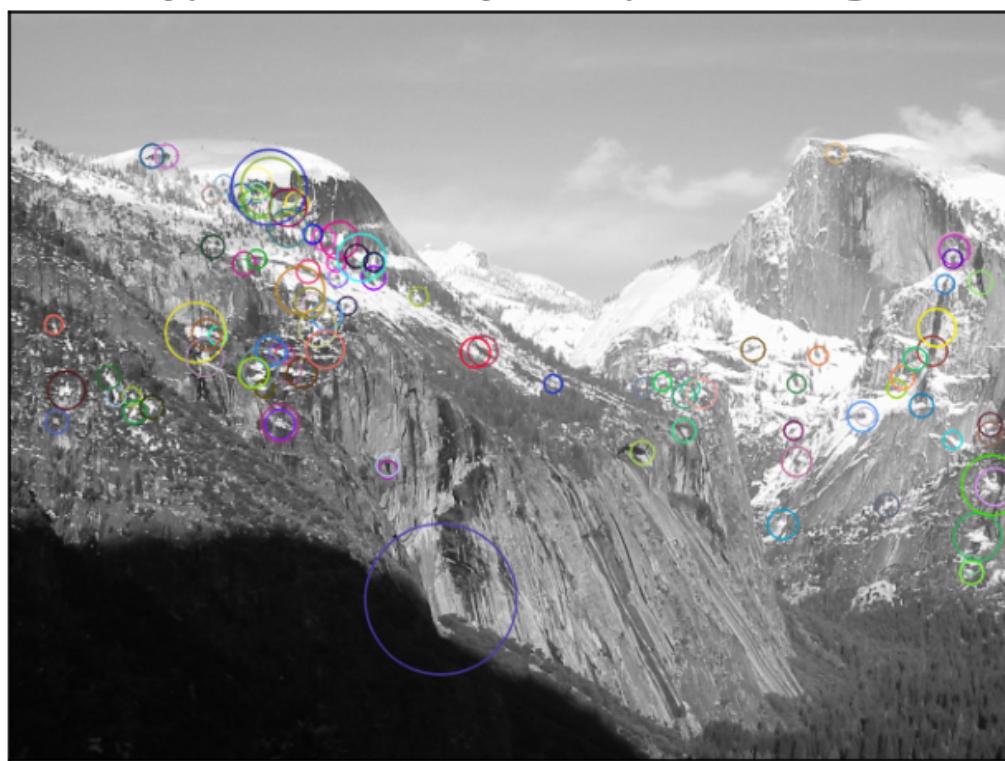
Siendo más específicos, en la función `getKeyPoints_BF()` luego de crear un objeto del algoritmo SIFT con `cv.SIFT_create()` y un objeto del matcher de fuerza bruta con `BFMatcher_create(cv.NORM_L2, True)` donde se le indica que utilice normalización L2, que es distancia euclídea, para calcular la similitud entre un punto en correspondencia de una imagen y la lista de puntos de la otra imagen, se indica `True` para indicar que se utilice CrossCheck, es decir, que solamente se obtengan aquellas parejas de puntos (i, j) tal que el punto i tiene como punto más cercano/similar al punto j y que el punto j tenga como punto más cercano al punto i .

Luego, se utiliza el método de SIFT `cv.detectAndCompute(img, None)` para obtener los puntos característicos de las dos imágenes y sus descriptores, dos llamadas por cada imagen y se utiliza `None` para indicar que no se están utilizando máscaras. Esta función implementa la versión completa de SIFT, su detector (lo que se ha realizado en el ejercicio 1, pero mejor) y los descriptores para esos puntos característicos.

Finalmente, se utiliza `BFMatcher.match(Dimg1, Dimg2)` para calcular las correspondencias entre los puntos utilizando sus descriptores y se devuelven aquellos en correspondencia dentro de un objeto `DMatch`.

En la función `getKeyPoints_Lowe2NN()`, el procedimiento inicial es similar a diferencia de que se crea el matcher sin el CrossCheck, se llama nuevamente a la función de SIFT para detectar y computar los puntos en correspondencia y sus descriptores.

100 Keypoints con mayor respuesta en general



100 Keypoints con mayor respuesta en general

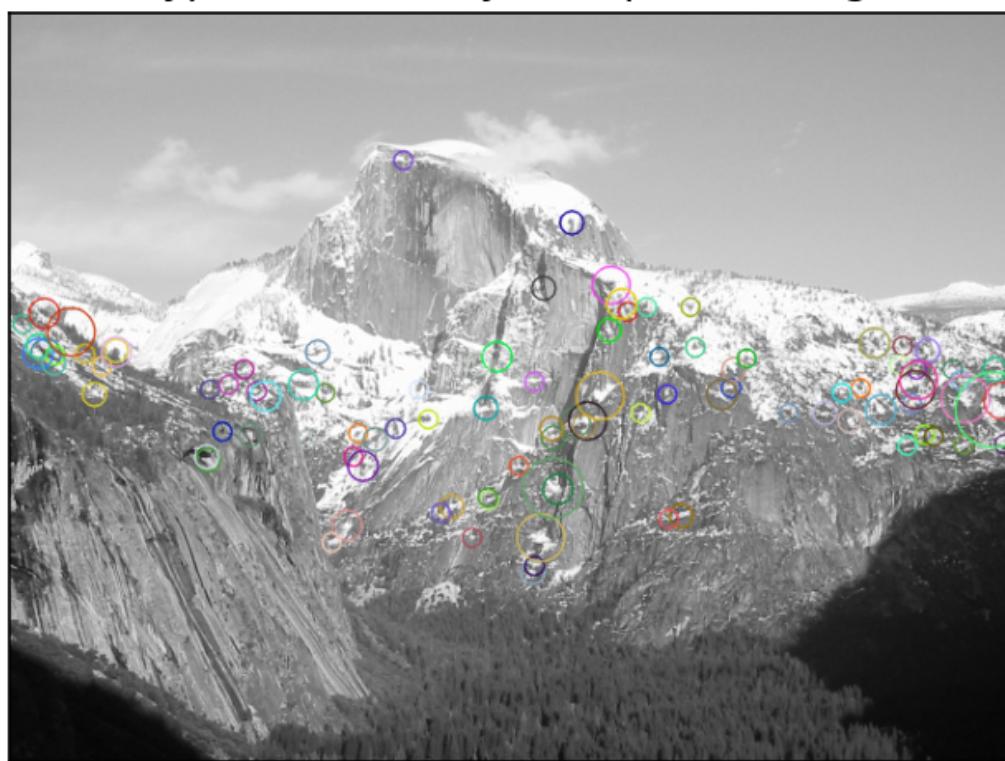


Figura 6: Puntos característicos detectados en Yosemite1.jpg y Yosemite2.jpg

SIFT: BruteForce + CrossCheck



Figura 7: Puntos en correspondencia, BruteForce+CrossCheck

Ahora, en vez de utilizar `BFMatcher.match(Dimg1, Dimg2)` se utiliza `BFMatcher.knnMatch(img1, img2, k)` donde se obtiene para un punto en correspondencia en una imagen, los dos puntos que tienen la distancia/similitud más cercana a cada punto, esto se utilizará el método que Dr. Lowe especifica: si la distancia de un punto a su mejor correspondencia es más pequeña que la distancia de un punto a su segunda mejor correspondencia multiplicado por 0.8, entonces el punto no es ambiguo pues la distancia entre esos dos es más cercana en proporción y se considera que es válido, se añade en otra lista que es la que se retorna de la función.

Una vez realizado esto, en el código propio del ejercicio de obtienen aleatoriamente 100 puntos, tal como se especifica en el guión y por solamente estar seguros, si se diera el caso de que hay menos de 100 puntos, entonces se incluyen solo esos.

Finalmente, se utiliza `drawMatches` y `drawMatchesKnn` que toman como parámetros las dos imágenes —que deben convertirse a `uint8` sino OpenCV se queja— sus puntos y la lista de descriptores de los puntos que están en correspondencia para pintar en las imágenes líneas que corresponden de un punto en una imagen a su correspondiente en la otra imagen, esto se logra con `flag=2` que indica pintar líneas en diferentes colores entre cada punto.

Los resultados de este procedimiento anteriormente descrito pueden observarse en las Figuras 7 y 8

Se puede observar que, ambos procedimientos en general están realizando el objetivo de obtener los puntos que en dos imágenes corresponden a la misma característica de la escena, pero hay una clara diferencia en la calidad de las soluciones.

En el método de BruteForce+CrossCheck de 7 se puede observar que las líneas por lo general se encuentran paralelas lo que tiene sentido en estas imágenes que son de un panorama horizontal y por lo tanto los puntos que corresponden deberían estar a la misma altura en las dos imágenes aproximadamente, aún así, varias correspondencias se dan en zonas de las imágenes donde no hay solapamiento de las mismas pero que si poseen una “textura” similar y también hay líneas que

SIFT: Lowe-Average-2NN

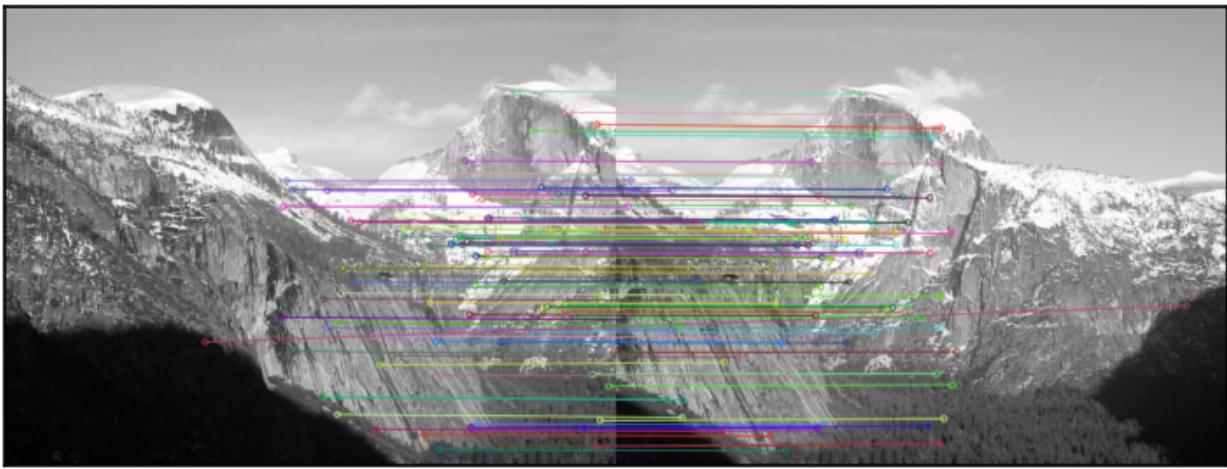


Figura 8: Puntos en correspondencia, Criterio de Lowe

se cruzan y van en diagonal: esto es señal de que esos puntos son “outliers” puesto que se han clasificado como correspondientes cuando no lo son realmente. Esto confirma lo visto en teoría, que aunque se introduzca el CrossCheck para cerrar más la búsqueda de puntos correspondientes, surge el problema de los puntos ambiguos, dado porque en la imagen pueden haber zonas que son muy parecidas a otras zonas y no necesariamente son la misma: existe una ambigüedad debido a que un punto “no sabe” del contexto de donde está en la imagen más allá de lo que tiene su descriptor.

Aquí es donde entra el método de Lowe, dados estos puntos ambiguos se puede obtener una proporción de las distancias entre la primera mejor correspondencia y la segunda; este método puede verse es superior debido a que –dependiendo de la ejecución– muy pocas líneas diagonales se generan y la mayoría de los puntos yacen en el centro de cada imagen que tiene sentido pues es la zona donde se solapa la escena de la montaña en las dos fotos, por lo tanto también valida los resultados obtenidos por el Dr. Lowe de que este método reduce substancialmente el porcentaje de outliers, aún así, pueden seguir apareciendo, por lo tanto se ha de realizar otro procesamiento de filtrado superior, en nuestro caso utilizando el método RANSAC en el ejercicio siguiente.

4. Construcción de un panorama rectangular

Se realizó una función tal como se pide en el enunciado denominada `genSimplePanorama(centre, left, right, canvas)` que calcula la homografía entre 3 imágenes; como se indica en los parámetros toma una imagen previamente seleccionada como la central, una imagen que va a su izquierda y una imagen que va a su derecha y el canvas de un tamaño adecuado al donde se pintará el mosaico.

En el ejercicio, se tomó como imagen central la imagen `IMG_20211030_110415_S.jpg`, imagen que tiene de centro la escena a La Alhambra, se toma la imagen directamente a su izquierda, `IMG_20211030_110413_S.jpg` y a su derecha `IMG_20211030_110417_S.jpg`.

En primer lugar se cargan las tres imágenes a memoria, y se genera un canvas con `np.zeros()` de un tamaño que se ha determinado por prueba y error y se llama la función `genSimplePanorama()`.

Dentro de la función primero se determinan los puntos en común que tienen las imágenes, esto se realiza con la función previamente definida para el ejercicio anterior, `getKeyPoints_Lowe2NN(img1, img2, p_k=2)` la cual obtiene los keypoints y los descriptores utilizando `knnMatch` que luego se mejora por medio del criterio de Lowe.

Luego de esto, con los puntos obtenidos por el matching se separan, `queryIdx` se refiere a los descriptores de los puntos de la imagen primera y `trainIdx` a los de la segunda imagen.

Una vez se tienen los puntos separados, se llama a `cv.findHomography(srcPoints, dstPoints, method)`, donde los `srcPoints` se refieren a los puntos de la imagen a la que se quiere obtener una homografía para que encajen con los puntos `dstPoints` y el método para quitar outliers, como se indica en el guión es por medio del algoritmo RANSAC, utilizando `cv.RANSAC` para ello.

En este caso sencillo de tres imágenes, los puntos referentes a `srcPoints` son aquellos de las imágenes derecha e izquierda, y los puntos `dstPoints` son aquellos de la imagen central dado que se quiere realizar una homografía de esas imágenes hacia el centro.

Una vez se realiza esto para la imagen de la izquierda con el centro y de la derecha con el centro se calcula una homografía de traslación de la imagen central hacia el canvas, esto se realiza generando a mano una matriz 3×3 con diagonal 1, donde se indica que píxeles hay que moverse, en este caso para centrar la imagen central en el centro del canvas, se obtiene la diferencia entre la mitad de los tamaños del canvas y de la imagen, la lógica es que se tiene que mover cada píxel esa traslación para mover la imagen de la esquina superior izquierda hacia el centro del canvas, un ejemplo de esta homografía puede verse en la ecuación 4, con i, j representando cada lado de la imagen `img` y el `canvas`, x, y cada píxel de la imagen original.

$$\begin{bmatrix} 1 & 0 & i = T_{\text{canvas}_i}/2 - T_{\text{img}_i}/2 \\ 0 & 1 & j = T_{\text{canvas}_j}/2 - T_{\text{img}_j}/2 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + i \\ y + j \\ z \end{bmatrix} \quad (4)$$

Una vez realizado esto, se llama a `cv.warpPerspective(src, M, dsize, dst, borderMode)` tres veces, en cada caso `src` indica la imagen a “pegar” en el canvas, `M` es la homografía que se aplicará a la imagen, `dsize` es una tupla que indica las dimensiones de la imagen destino, `dst` indica la imagen como tal a la que se quiere “pegar” la imagen `src` y `borderMode` indica el tipo de borde.

La primera aplicación de esta función se realiza sin `borderMode` puesto que en el ordenador del autor, si se aplica `borderMode=cv.BORDER_TRANSPARENT` la imagen entera tendrá ruido en los píxeles “vacíos”.

En cualquier caso, como es la primera imagen que se va a pegar no es realmente necesario que los bordes sean transparentes; y esta imagen en la implementación es la imagen a la derecha del centro.

La homografía que se utiliza es el producto punto de la homografía de traslación de la imagen del centro hacia el canvas y la homografía entre la imagen derecha y la del centro.

La segunda aplicación de `warpPerspective()` sí utiliza los bordes transparentes ya que aquí si se quiere que lo que no se está pegando se mantenga igual, esta segunda imagen que se aplica es la imagen a la izquierda del centro.

Nuevamente la homografía es el producto punto de la homografía de traslación de la imagen hacia el canvas y la homografía entre la imagen izquierda y la del centro.

Esto es posible realizarlo debido a que las homografías de dos transformaciones o más pueden unificarse en una sola homografía que es el producto de las homografías anteriores.

Finalmente, la última llamada a la función se realiza con la imagen central y únicamente con la homografía de traslación previamente definida a mano, ya que esta imagen no tiene que transformarse porque se está tomando como la base para que el resto de imágenes se transformen a su perspectiva de la escena.

En este ejercicio se tomaron las imágenes que se pueden observar en la Figura 9, y su panorama puede visualizarse en la Figura 10.

Puede observarse que efectivamente se comprueba la propiedad que se estudió en teoría sobre el producto entre homografías pues es fácil notar que las imágenes encajan entre sí formando un único panorama ininterrumpido además de que también comprueban que la manera de transformar una imagen de manera proyectiva o afín es por medio de una homografía, ya que si este no fuera el caso no se podría utilizar este método para generar panoramas entre imágenes que no se sabe que clases de transformaciones han sufrido entre sí.

Esto también confirma que RANSAC es capaz de eliminar aquellos puntos denominados “outliers” puesto que la calidad de la homografía depende también de que los pares de puntos sean realmente correspondientes, puesto que en caso contrario puede aumentar el error y por lo tanto la homografía se vería con errores, como por ejemplo que no encajasen bien las imágenes al montarlas en el canvas, cosa que no ocurre en el resultado obtenido.

5. Bonus

5.1. Obteniendo la respuesta más alta por octava

Se ha realizado una función adicional, `getBestPercentage(keyPoints)` que tiene como único parámetro la lista de puntos plana; se ha añadido código en la función `getExtrema()` de manera que se puedan obtener ahora los puntos de mayor respuesta por cada octava de la pirámide de Lowe, la proporción que se ha pedido es: 50 en la octava 0, 25 en la octava 1, 15 en la octava 2 y 10 en la octava 3.

Para realizar esto, en `getExtrema()` se declara una nueva lista `keyPerOct` que almacenará listas de puntos por cada octava, de esta forma se podrán luego ordenar de una manera más cómoda, se añade una nueva lista en cada iteración de la octava, se añade un punto a la lista de la octava correspondiente cuando es detectado por `isLocalExtrema()`, se devuelve esta lista junto con las originales y luego se utiliza en la función `getBestPercentage()`.

Esta función sencilla consta de dos bucles, el primero se utiliza para ordenar una lista de puntos de una octava en particular y directamente en ese mismo bucle añadir a una lista de listas nueva la proporción necesitada de puntos por cada escala.

Se tiene un segundo bucle que lo que hace es generar los objetos `cv.KeyPoint` por cada punto y almacenarlo en una lista normal de Python, la cual se devuelve junto con la lista “normal” de

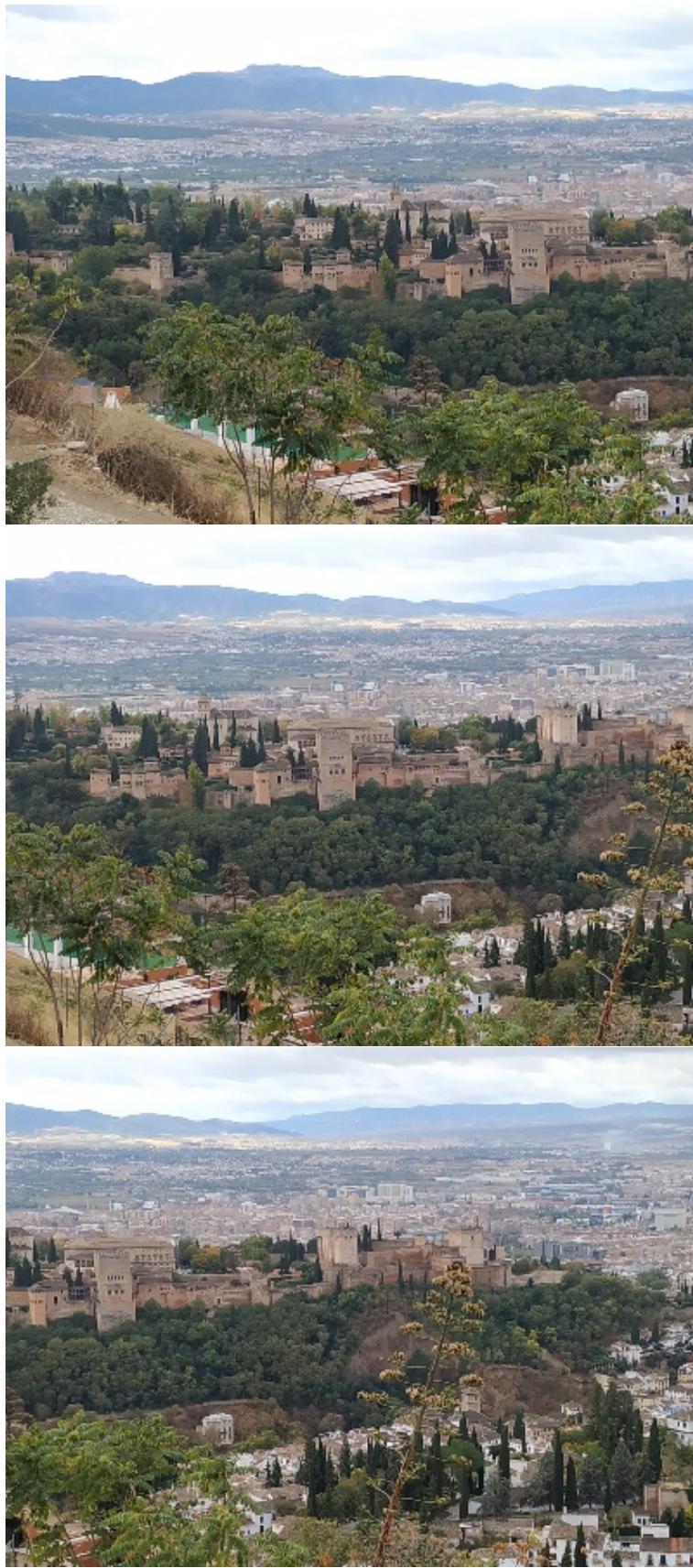


Figura 9: Imágenes utilizadas para el ejercicio, en orden de izquierda a derecha.

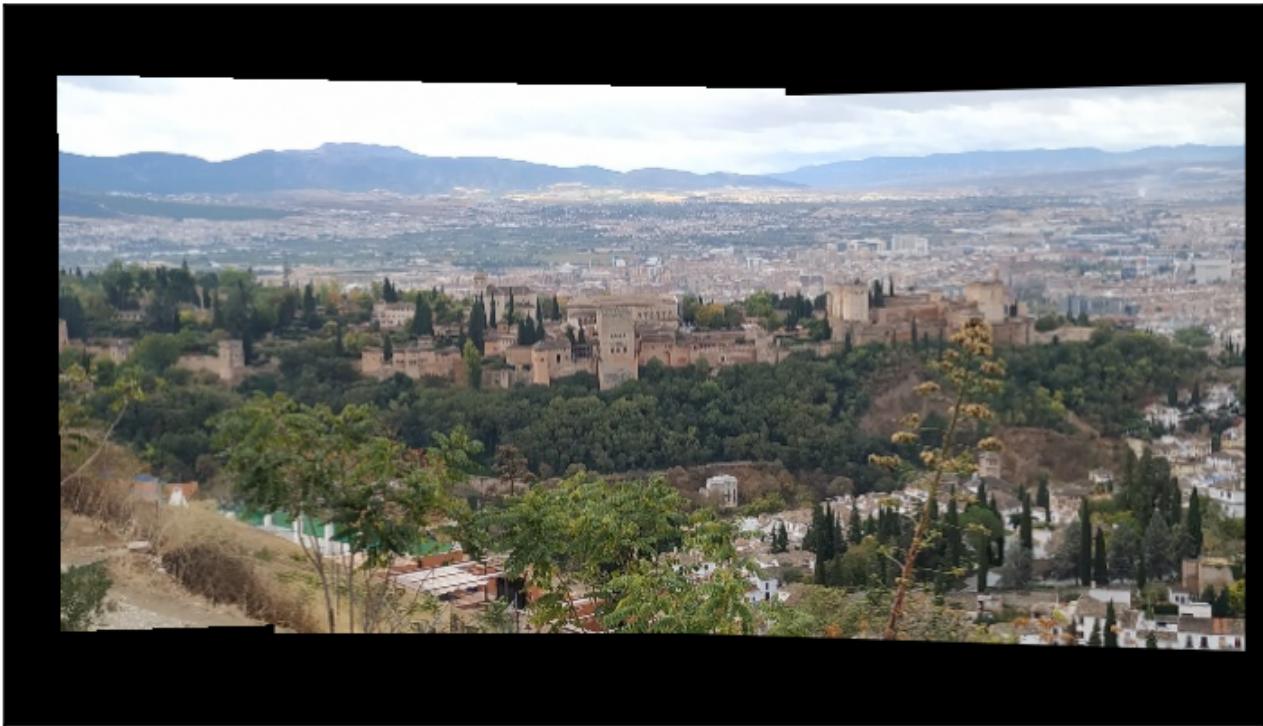


Figura 10: Panorama resultante de la Figura 9

puntos.

Una vez en el código del ejercicio se pintan las imágenes luego de superponer los puntos con `drawKeypoints()`.

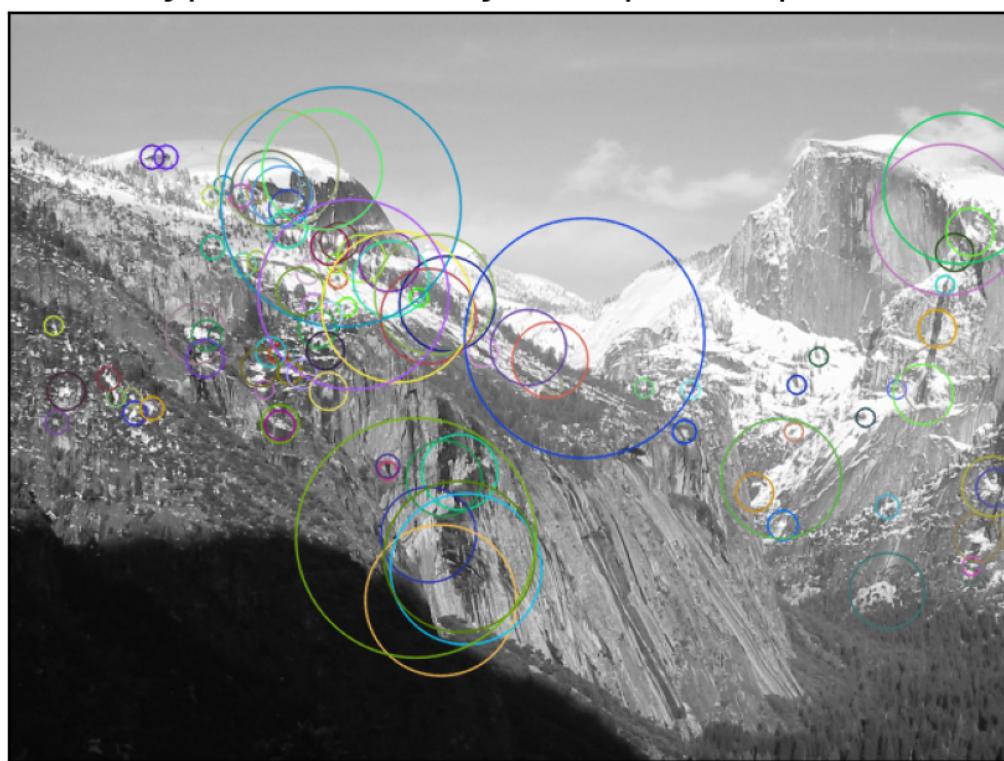
Se realizó de esta manera pues se pensó es más sencillo al detectar puntos, tenerlos ya separados por escala que luego tener una única lista y tener que procesarla para determinar que puntos pertenecen a que escala, aunque también es una opción válida.

La función se tiene para tener el código más organizado, también era posible llamar a la función dentro de la función `getExtrema()` o incluso ahí directamente reordenar los puntos pero por mantener separado lo que es ejercicio y bonus, está fuera de la misma, en todo caso esto no afecta la funcionalidad.

Se pueden visualizar los resultados en la Figura 14.

Se observa que naturalmente ahora hay círculos de mayor diámetro que antes por la presencia de octavas mayores, pero siguen teniendo sentido: no hay puntos que están en el cielo o en lugares planos, por lo que se mantiene las conclusiones del ejercicio 1, que el detector está funcionando, se obtienen puntos en todas las octavas; afirmando nuevamente que el algoritmo implementado está logrando el objetivo deseado.

100 Keypoints con mayor respuesta por octava



100 Keypoints con mayor respuesta por octava

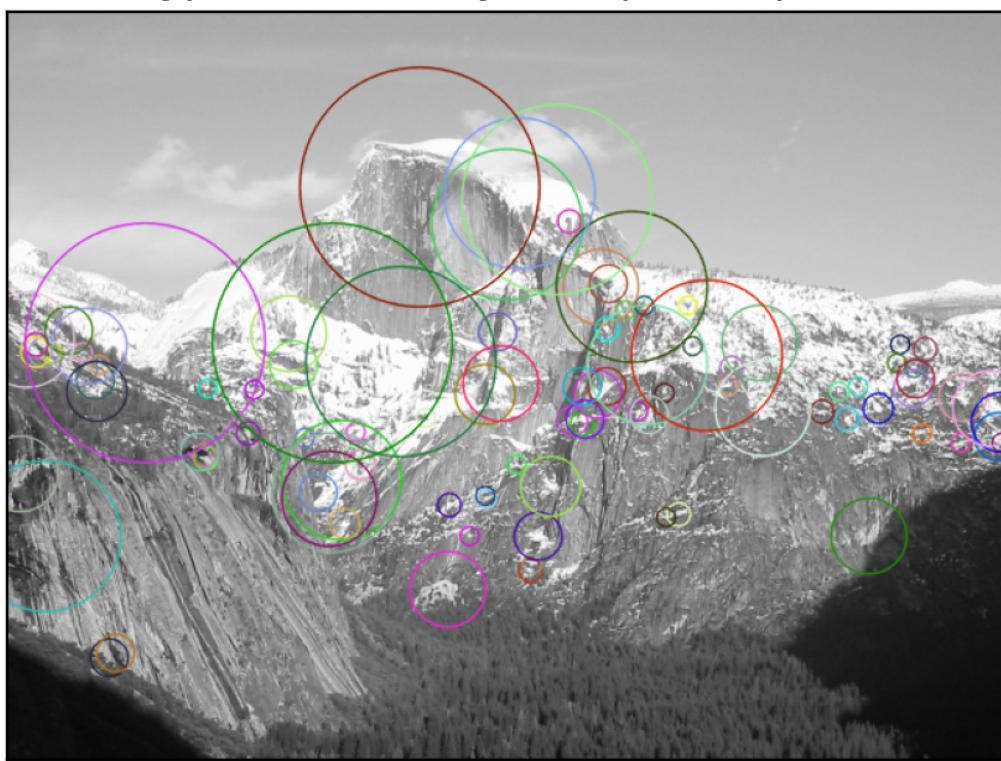


Figura 11: Puntos por octava característicos en Yosemite1.png y Yosemite2.png

5.2. Obteniendo puntos mejorados por interpolación

Se ha realizado una función denominada `extremaInterpolation(keyList, DoG)` que toma como parámetros una lista de puntos plana y la pirámide laplaciana obtenida anteriormente.

Dentro de esta función se implementan los algoritmos que se describen en el artículo científico “Anatomy of the SIFT Method”, en la sección 3.2 del mismo.

Se utilizan los 100 mejores puntos en general para la comparación por seleccionar aquellos que mejor representan la detección de los puntos, y también resultan una cantidad manejable de puntos para poder comparar como varían. El algoritmo funciona para cualquier cantidad de puntos.

En esencia, dada la lista de puntos se tienen que procesar para obtener su localización refinada, pero también es un proceso de filtración: no todos los puntos pueden ser válidos, según explica el artículo, esto se realiza para evitar inestabilidades numéricas.

La función es un bucle que va recorriendo la lista de puntos, en una iteración cualquiera se toma un punto y se realiza interpolación cuadrática, en este caso utilizando la función `localQuadratic(kp, DoG)` que toma como parámetros un punto y el espacio de escalas laplaciano ya que se tendrá que acceder a las diferentes escalas de una octava.

Dentro de esta función se obtienen primero las escalas que conforman el cubo donde fue detectado el punto y con ellas se realizan los procesos descritos en el artículo de manera que se generan dos matrices 3×3 con la gradiente y hessiana discretizadas, que en esencia son diferencias entre los diferentes píxeles alrededor del punto detectado en la misma escala y en las escalas anterior y posterior que forman el cubo.

De esta función se generan dos elementos: un vector de 3 elementos denominado α^* que contiene el sesgo que debe añadirse la escala y las coordenadas del punto y también un valor ω que contiene el valor de la respuesta del punto interpolada.

Estos valores son devueltos a la función `extremaInterpolation()` donde se utilizan de dos maneras, como se describe en el artículo:

Primero, se calcula el punto real, es decir, se convierten las coordenadas relativas a la escala y octava a aquellas de la imagen, junto la escala real, se utilizan las funciones previamente definidas, `realCoords()` y `getRealSigma()` y también se actualiza el punto “relativo”, aquel que fue seleccionado de la lista y que posee las coordenadas en relación a la escala, en este caso se redondean los valores al entero más próximo y ahora se tiene que determinar si el punto es válido o no: si alguno de los miembros de α^* tiene un valor absoluto máximo $< 0,6$ se considera que es un punto válido, sino, se vuelve a intentar la interpolación con el punto actualizado con los valores actuales, esto así hasta un máximo de 5 intentos, si no se ha logrado que la interpolación esté a menos que ese valor, el punto cae fuera del rango de validez y por lo tanto es descartado.

Se ha realizado de esta manera pues así está dispuesto en el artículo anteriormente mencionado, lo que se ha tenido es que, en primer lugar entender qué se está haciendo a nivel intuitivo y posteriormente se tuvo que adaptarlo para utilizar funciones tanto de Numpy como de Python, por ejemplo para realizar los cálculos de matrices de la función `localQuadratic()`.

Los puntos obtenidos son añadidos a una lista plana, para poder ser ver su valor de una manera más clara y a una lista de objetos `KeyPoint` para poderlos visualizar por la función `drawKeypoints()`.

Se pueden observar en la diferencia visualmente en las imágenes de la Figura 12, donde los puntos originales están pintados en rojo y los puntos interpolados que han sido validados por la interpolación en verde, cabe notar que las diferencias entre los puntos a veces son muy muy pequeñas pero un buen ejemplo es el punto detectado en `Yosemite1.jpg`, en una escala grande justo en la sombra de la montaña, puede verse que la escala se ha reducido ligeramente y el punto se ha movido, hay varios puntos que también han sido eliminados y otros que prácticamente se solapan, para estas dos imágenes aproximada 20 % de los puntos no se consideraron estables y fueron descartados.

Un ejemplo más claro es el siguiente: puede verse en el punto de índice 47 en los 100 mejores puntos e índice 44 en los interpolados, se muestra como se ha movido su versión interpolada para encajar mejor con el punto en la imagen real, se puede ver en la Figura 13.

Numéricamente también se nota la diferencia:

Punto Original		Punto Interpolado	
(x, y)	σ	(x, y)	σ
(242, 293)	$\frac{19,2}{12} = 1,6$	(242.2254, 290.1786)	$\frac{16,1321}{12} = 1,3443$

De esto puede concluirse que, la interpolación se está realizando de una manera que concuerda con lo que se establece en el artículo y queda validado que los pasos a realizar efectivamente producen un resultado que es posible notar a simple vista, se puede ver que la detección del punto se ha centrado más en punto de la imagen que le corresponde y se han ajustado también las escalas para compensar por la discretización de la escena al ser pasada a una imagen, que es la razón de ser de esta interpolación, poder tener una precisión subpíxel de los puntos detectados.

5.3. Panorama de Proyección Plana Completo

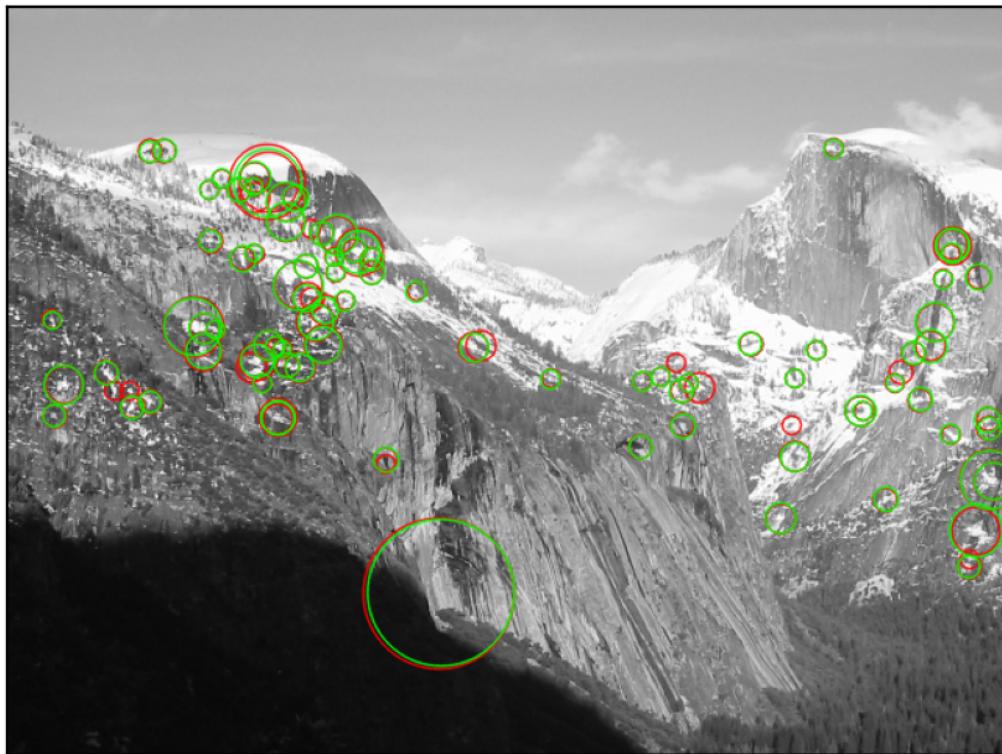
Se ha realizado una función adicional denominada `genPanoramaFlat` (`center`, `left`, `right`, `canvas`) la cual es una evolución de la función anteriormente declarada para el Ejercicio 3, esta función calcula las homografías entre varias imágenes consecutivas automáticamente y las va añadiendo al canvas que se pasa por parámetro.

El parámetro `center` sigue siendo una imagen, junto con `canvas` que es un lienzo de un tamaño suficientemente grande cuyo tamaño ideal se obtuvo medio de prueba y error, pero ahora, `left` y `right` son listas de imágenes que deben de estar en un orden específico para que el cálculo de las homografías se realice correctamente.

El orden de las imágenes deben ser tal que, la primera imagen de cada lista debe de ser la imagen que está directamente a la izquierda o a la derecha respectivamente de la imagen central, y cada imagen siguiente debe de ser la imagen directamente a la izquierda o a la derecha de la imagen anterior.

También, se tuvo que considerar que imagen tomar como la imagen central para mitigar la distorsión creada por este tipo de proyección, en este caso se seleccionó la imagen `IMG_20211030_110420_S`, la 6^a imagen del panorama si se inicia a contar por la izquierda puesto que viendo todas las imágenes el centro del panorama se encuentra por esa parte de la escena que capturó esa imagen.

85 Keypoints con mayor respuesta interpolados



82 Keypoints con mayor respuesta interpolados

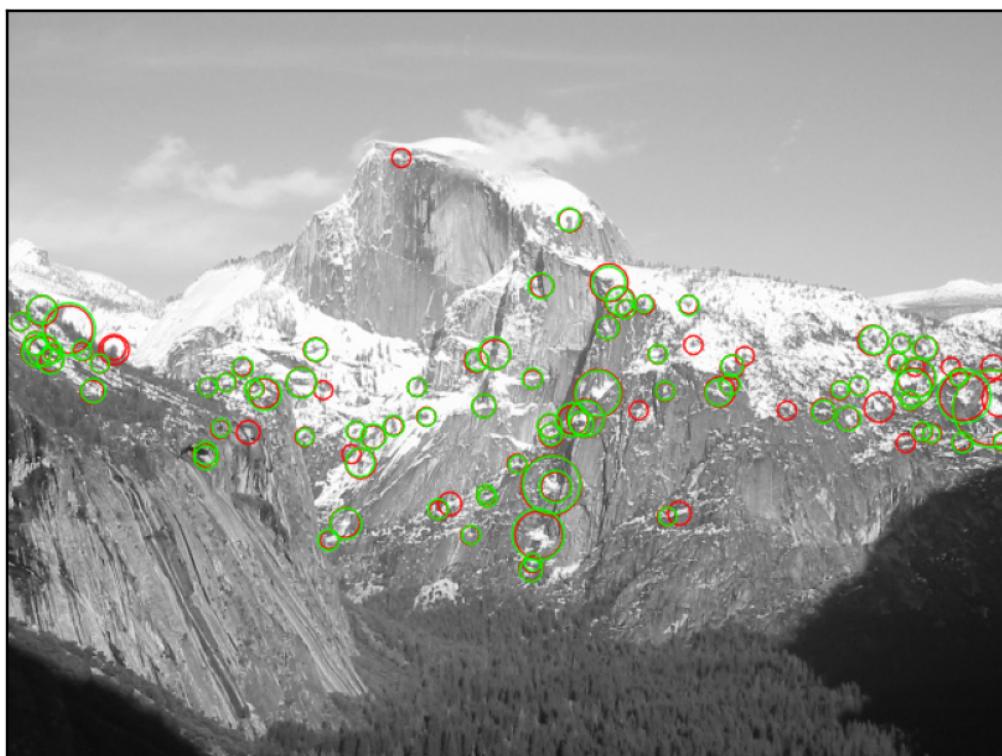


Figura 12: Comparación de los puntos originales (rojo) y los puntos interpolados (verde) para Yosemite1.jpg y Yosemite2.jpg

Zoom: Ejemplo de KeyPoint Interpolado

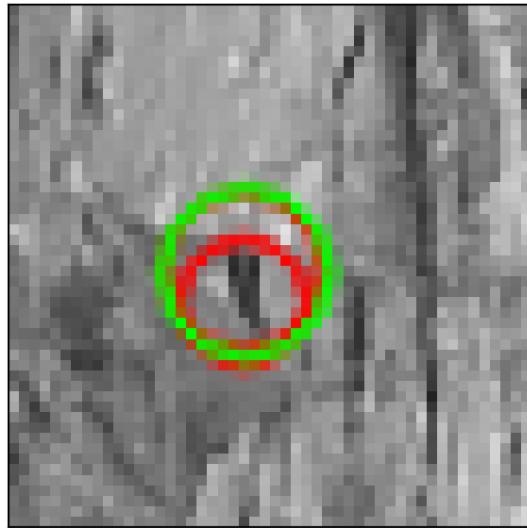


Figura 13: Comparación con más detalle de un punto original, rojo, y su versión interpolada en verde, sacado de `Yosemite1.jpg`, punto ubicado alrededor del centro de la imagen en la parte inferior, ligeramente a la izquierda y arriba del punto a mayor escala debajo.

Internamente, la función `genPanoramaFlat()` expande lo realizado en `genSimplePanorama()`, igual que en su antecesor se genera una homografía de traslación del centro de la imagen al canvas.

Luego se declaran dos listas las cuales contendrán las homografías de las imágenes a la izquierda y a la derecha de la imagen central.

En primer lugar, se genera la homografía de transformar la primera imagen de la izquierda a la perspectiva de la imagen central con `getHomography(img1, img2)` que encapsula el procedimiento que se aplicó en `genSimplePanorama()`, se realizó de esta manera para mantener el código más legible y evitar código repetido; la intuición de la función es que se quiere obtener la homografía para transformar la `img1` para que encaje con `img2`.

Directamente se obtiene el producto punto de la homografía de translación general y esta homografía, se almacena como el primer miembro de la lista de homografías de las imágenes a la izquierda y también se tiene como una variable llamada `prev` que será útil luego.

Ahora, se entra en un bucle que recorre el resto de imágenes que se encuentran a la izquierda de la homografía y se empieza a obtener la homografía entre cada par de imágenes consecutivas, y directamente se calcula la homografía que tendrá que aplicarse en general gracias a la variable `prev` que va acumulando todas las transformaciones que se realizan entre las imágenes, esto se debe a que para que se proyecte bien, por ejemplo, la última imagen a la izquierda tiene que aplicarse la transformación dada por homografía entre esa imagen y su vecina, y también la transformación que se le aplicaría a su vecina con la vecina de la vecina hasta la imagen central y luego la translación al canvas, todo esto se encuentra ya calculado en una homografía por la manera en que se acumula en `prev` las transformaciones de cada par de imágenes.

Se repite el mismo proceso ahora con las imágenes a la derecha del centro.

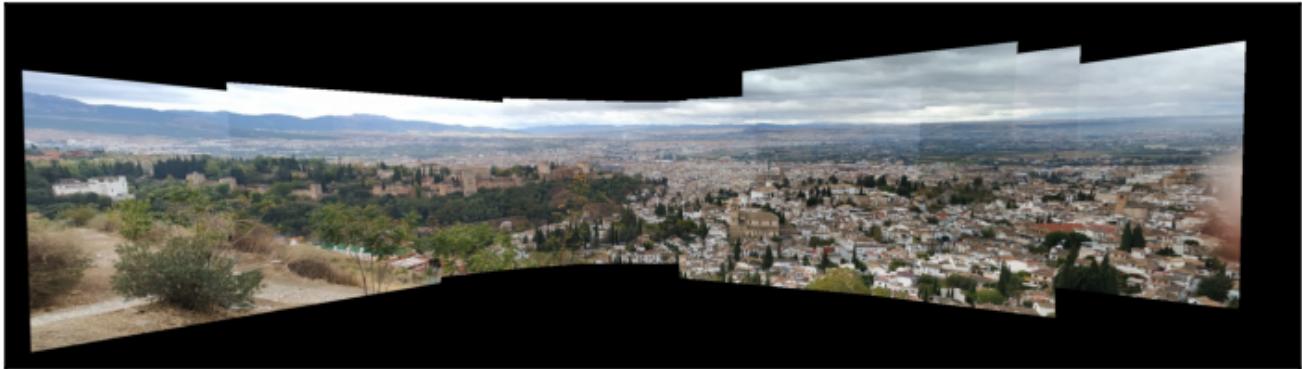


Figura 14: Panorama plano con todas las imágenes

Notar que esto funciona debido a que las imágenes están ordenadas de manera que mientras más alto es su índice en la lista, están más alejadas del centro de la imagen y permite que la función `genHomography()` obtenga las homografías de la manera correcta puesto que se obtiene siempre la homografía de la imagen actual con la imagen inmediatamente anterior, que es la que está más cerca del centro.

Una vez realizado esto, por comodidad se invierten las listas de imágenes a los lados y sus homografías, se pega la primera imagen al canvas, al igual que en el Ejercicio 3, la primera imagen que se pega no posee los bordes transparentes por detalles de implementación. Esta imagen es la imagen más a la izquierda del panorama, se decidió de manera arbitraria, pues sería lo mismo pegar la imagen más a la derecha del panorama. Si es importante que sea la más alejada pues el resto de las imágenes se empezarán a pegar –utilizando `warpPerspective()` y la homografía correspondiente– en unos bucles de manera que las imágenes más alejadas quedarán por debajo de las imágenes más cerca del centro, así se puede mantener más la calidad del panorama pues las imágenes más cercanas al centro habrán sufrido menos deformaciones y tendrán mejor calidad.

Una vez estos bucles finalizan, se pega encima de todo la imagen central con la homografía de traslación y se devuelve el resultado.

El resultado se puede observar satisfactoriamente en la Figura 14, se puede observar que nuevamente se confirma que las homografías pueden combinarse en una sola homografía que realiza la transformación de sus partes, así mismo puede observarse y verificar lo aprendido en teoría, las transformaciones planas empiezan a acumular deformaciones mientras más se aleja del centro de la imagen pues se está intentando proyectar en un plano otro plano que no es totalmente paralelo al mismo, y mientras esta diferencia crece, la proyección se deforma más y más.

Se puede observar que la imagen no está tan deformada por haber elegido la 6^a imagen, pues la deformación a los lados es más o menos simétrica, no se da que hay más deformación de un extremo que el otro.

También se puede notar que por la ausencia de blending entre las imágenes, hay diferentes tonos en el panorama que revelan donde las imágenes han sido cosidas, pero puede verse que salvo este detalle la homografía en efecto hace que encajen las imágenes donde deberían estar.