

Visión por Computador

Práctica 3: Detección de puntos relevantes y construcción de panoramas

Antonio Coín Castro

Curso 2019-20

Estructura del código

Se han desarrollado una serie de funciones genéricas de representación de imágenes, extraídas en su mayoría de prácticas anteriores (aunque con ligeras modificaciones). Hay una serie de consideraciones a tener en cuenta:

- En general, todas las imágenes se convierten a números reales en el momento en que se leen, y solo se normalizan a $[0, 1]$ cuando se vayan a pintar.
- Tras mostrar una ventana de imagen, el programa quedará a la espera de una pulsación de tecla para continuar con su ejecución (incluso si se cierra manualmente la ventana del *plot*).
- Hay una serie de parámetros globales (editables) al inicio del programa para modificar ciertos comportamientos de algunas funciones.
- Las imágenes deben estar en la ruta relativa `imagenes/`.
- Todas las funciones están comentadas y se explica el significado de los parámetros.

El programa desarrollado va ejecutando desde una función `main` los distintos apartados de la práctica uno por uno, llamando en cada caso a las funciones que sean necesarias para mostrar ejemplos de funcionamiento.

Nota: Es posible que al ejecutar el programa aparezca el siguiente error: *Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers)*. Esto se debe a errores de redondeo en los cálculos, que hacen que las imágenes se salgan del rango $[0, 1]$ por valores despreciables del orden de 10^{18} .

Ejercicio 1: detección de puntos Harris

En este ejercicio perseguimos utilizar el detector de Harris para detectar esquinas sobre la pirámide Gaussiana de una imagen. Mostraremos los resultados obtenidos sobre las imágenes `Yosemite1.jpg` y `Yosemite2.jpg`.

Disponemos de funciones ya utilizadas en prácticas anteriores, cuya implementación no comentaremos, para realizar las siguientes tareas específicas: supresión de no máximos, pirámide gaussiana, filtro gaussiano, cálculo de derivadas, y completar una imagen con 0s hasta tener tamaño potencia de 2 en ambas dimensiones.

Extracción de puntos

En primer lugar, completamos nuestra imagen en ambas dimensiones hasta que tengan tamaño la potencia de 2 más cercana. Esto será útil para no perder información de la imagen en las sucesivas etapas de la pirámide Gaussiana, pero al mostrar las imágenes las mostraremos sin el borde negro. Para cada nivel de la pirámide, repetimos el siguiente proceso.

Comenzamos llamando a la función `cornerEigenValsAndVecs` para extraer para cada punto p información sobre la matriz de covarianzas de las derivadas en un entorno $S(p)$ del mismo, a saber:

$$M(p) = \begin{pmatrix} \sum_{S(p)} (dI/dx)^2 & \sum_{S(p)} (dI/dxdI/dy) \\ \sum_{S(p)} (dI/dxdI/dy) & \sum_{S(p)} (dI/dy)^2 \end{pmatrix}$$

Tras la llamada obtenemos una matriz con 6 canales, pero solo nos interesan los dos primeros, que contienen para cada punto los valores propios $\lambda_1(p)$ y $\lambda_2(p)$ de la matriz anterior:

```
dst = cv2.cornerEigenValsAndVecs(im_scale, blockSize = 3, 3)
```

El segundo parámetro define el tamaño del vecindario, que fijamos a 3×3 siguiendo las indicaciones de [1]. El tercer parámetro es el tamaño de los *kernels* usados anteriormente para calcular derivadas.

Ahora calculamos para cada punto la función *corner strength*, que no es más que la media armónica (reescalada) de los valores propios asociados:

$$f(p) = \frac{\lambda_1(p)\lambda_2(p)}{\lambda_1(p) + \lambda_2(p)}$$

Notamos que esta función mide de alguna forma la intensidad de las direcciones propias de la matriz anterior. Nos interesaría cuando el valor de f sea elevado, pues esto es indicativo de dos direcciones pronunciadas en el entorno del punto (una esquina).

```
for x in range(nrows):
    for y in range(ncols):
        11 = dst[x, y, 0]
        12 = dst[x, y, 1]
        f[x, y] = (11 * 12) / (11 + 12) if 11 + 12 != 0.0 else 0.0
```

Después realizamos supresión de no máximos sobre la matriz de *corner strength*, para quedarnos solo con aquellos puntos que superan un umbral y son máximos locales en un entorno de tamaño `window`, ordenados por intensidad. El tamaño de ventana comienza siendo de 7×7 para la escala más grande, y desciende hasta 3×3 para las escalas más pequeñas.

```
max_index = non_maximum_supression(f, window)
max_index = sorted(max_index, key = lambda x: f[x], reverse = True)
```

Posteriormente nos quedamos con un número prefijado de puntos, controlado por la variable `num_points`, que va disminuyendo en cada nivel (nos quedamos con los mejores, pues los tenemos ordenados). Ahora, calculamos una estructura `KeyPoint` para cada punto que sobrevive, donde guardamos la columna y la fila (en ese orden), el tamaño y el ángulo.

Para calcular el ángulo o la orientación de cada punto consideramos el vector unitario $(\cos \theta, \sin \theta) = u/|u|$, donde u proviene del gradiente (local) alisado: $u = \nabla_\sigma I$, con $\sigma = 4.5$. Antes de hacer la pirámide Gaussiana de la imagen original, le aplicamos el alisamiento Gaussiano mencionado, calculamos sus derivadas en x y en y , y realizamos la pirámide Gaussiana de cada una de ellas

```
im.blur = cv2.GaussianBlur(im_pad, ksize = (0, 0), sigmaX = 4.5)
im_dx = gaussian_pyramid(cv2.Sobel(im.blur, -1, 1, 0), levels)
im_dy = gaussian_pyramid(cv2.Sobel(im.blur, -1, 0, 1), levels)
```

Ahora, para cada punto normalizamos el vector $u(p)$, calculamos el seno y el coseno, y con ello la arctangente para obtener el ángulo en grados. Es importante notar que sumamos 180 grados al resultado para obtener valores en el intervalo $[0, 360]$.

```
# En el nivel s
norm = np.sqrt(im_dx[s][p] * im_dx[s][p] + im_dy[s][p] * im_dy[s][p])
angle_sin = im_dy[s][p] / norm if norm > 0 else 0.0
angle_cos = im_dx[s][p] / norm if norm > 0 else 0.0
angle = np.degrees(np.arctan2(angle_sin, angle_cos)) + 180
```

Estimamos el tamaño como `blockSize * (levels - s + 1)`, pues posteriormente la función usada para dibujar necesita que vaya disminuyendo el tamaño conforme aumentamos el nivel (debido al uso de un `flag` concreto). Aprovechamos y guardamos también los `keypoints` con sus coordenadas respecto a la imagen original, multiplicando las que tenemos por 2^s .

Por último, dibujamos los puntos detectados en el nivel actual con la función `drawKeypoints`. El `flag` que le pasamos permite mostrar también la orientación de cada punto. Finalmente dibujamos en la imagen original todos los puntos detectados en todos los niveles, donde los puntos más pequeños corresponden a niveles más altos.

```
im_kp = cv2.drawKeypoints(im_rgb, keypoints, np.array([]),
                           flags = cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
```

Mostramos ahora los resultados obtenidos con las imágenes de prueba.

```
Yosemite1.jpg
Puntos detectados en la octava 0: 1100
Puntos detectados en la octava 1: 733
Puntos detectados en la octava 2: 488
Puntos detectados en la octava 3: 312
Puntos totales detectados: 2633
```

Para la supresión de no máximos hemos fijado el umbral en `THRESHOLD = 10`, obteniendo un conjunto de más de 3000 puntos en total. En la primera octava hemos impuesto que nos quedamos con los 1100 mejores puntos, y a partir de ahí se disminuye la cantidad en

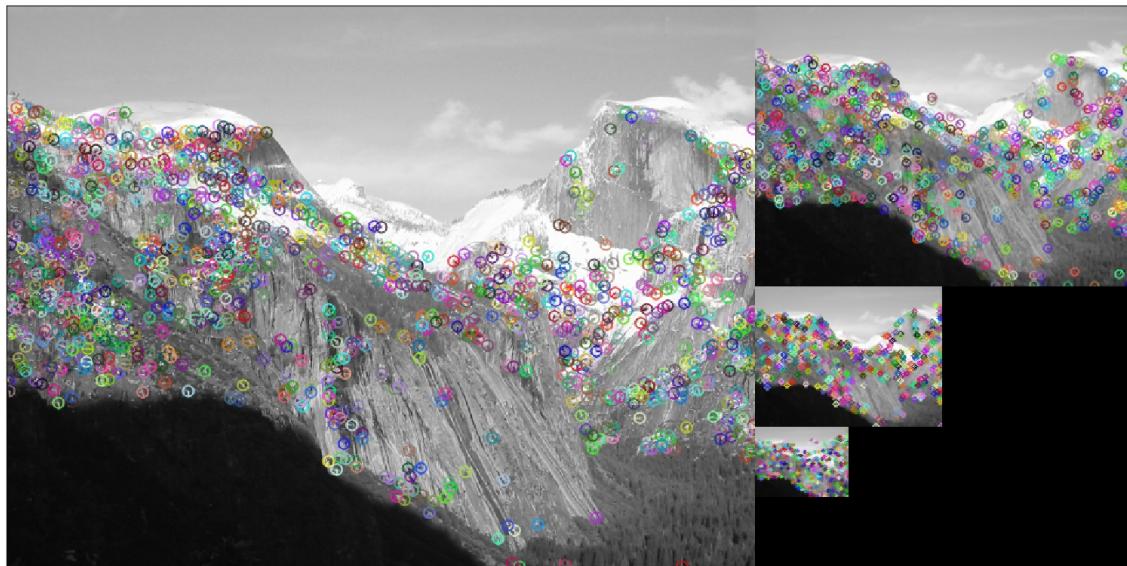


Figura 1: Puntos detectados en cada octava en Yosemite1.jpg

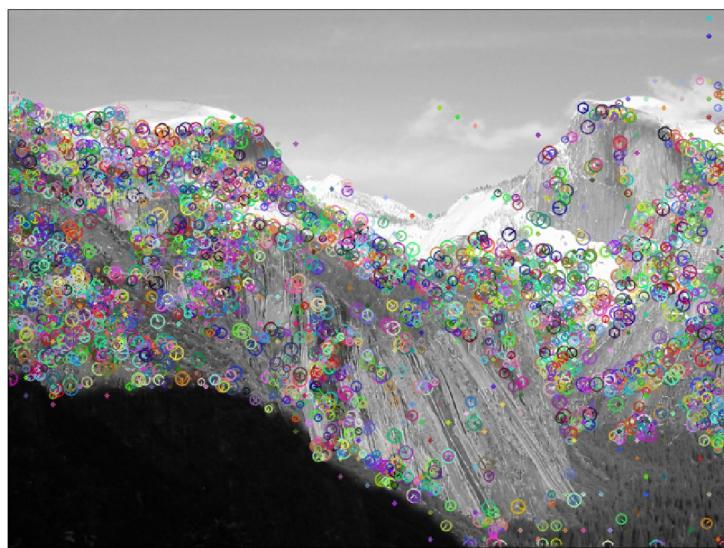


Figura 2: Puntos totales detectados en Yosemite1.jpg

un factor de 1.5 por octava. Obtenemos finalmente un conjunto numeroso, de más de 2000 *keypoints*.

Podemos observar que los puntos detectados son representativos de cada escala, pues marcan las esquinas existentes en toda la imagen y no solo en una parte. Sin embargo, muchos más puntos saturarían la imagen y no permitirían apreciar las esquinas más destacadas, sobre todo en los niveles más altos. En conjunto podemos ver que se obtiene un número suficiente de puntos.

Los resultados obtenidos y la valoración son similares para la imagen `Yosemite2.jpg`. Vemos que conforme aumentamos de octava detectamos esquinas correspondientes a lugares más detallados de la imagen, como por ejemplo las que aparecen en las nubes.

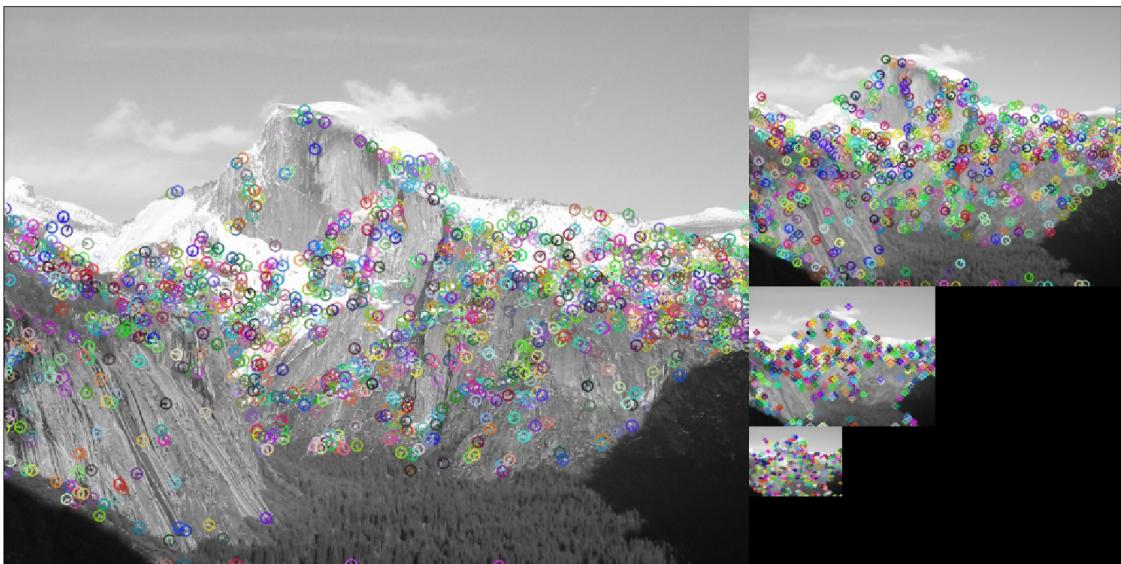


Figura 3: Puntos detectados en cada octava en `Yosemite2.jpg`

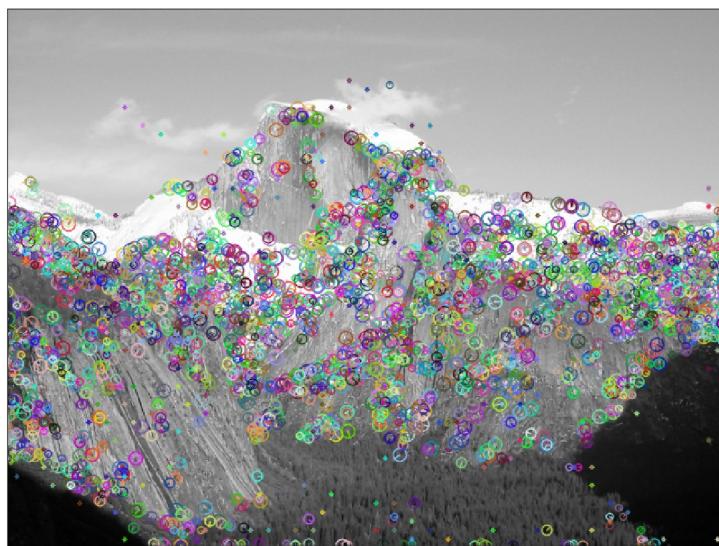


Figura 4: Puntos totales detectados en `Yosemite2.jpg`

Refinamiento de coordenadas

Una vez hemos detectado los puntos de interés, queremos refinar las coordenadas hasta niveles *subpixel* (es decir, en precisión decimal). Esto es así porque en aplicaciones posteriores estos puntos de interés deben ser muy precisos si queremos obtener un resultado estéticamente agradable.

Para ello disponemos de la función `cornerSubPix`, a la que le pasamos una lista con las coordenadas detectadas y un tamaño de ventana, y se encarga de refinar las coordenadas de las esquinas en un entorno definido por dos veces el tamaño proporcionado, hasta cumplir un cierto criterio de parada. En nuestro caso, el criterio de parada es superar las 100 iteraciones, o que no difieran las coordenadas en más de 0.01 en una iteración.

```
win_size = (3, 3)
zero_zone = (-1, -1)
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 100, 0.01)
points = np.array([p.pt for p in keypoints], dtype = np.uint32)
corners = points.reshape(len(keypoints), 1, 2).astype(np.float32)
cv2.cornerSubPix(im, corners, win_size, zero_zone, criteria)
```

El tamaño de ventana elegido es lo suficientemente grande como para tener margen de maniobra al refinar los puntos, y lo suficientemente pequeño como para no englobar a muchos puntos de interés en un mismo vecindario y confundir las esquinas. El parámetro `zeroZone = (-1, -1)` indica que no debemos ignorar ninguna región de búsqueda.

Para mostrar los resultados, elegimos aleatoriamente 3 puntos donde no coincidan las coordenadas originales y las corregidas, y mostramos una ventana 11×11 interpolada a un zoom de $5\times$.

```
# Recuperamos las coordenadas originales e interpoladas
y, x = points[index][:2]
ry, rx = corners[index][0][:2]

# Pasamos la imagen original a color para dibujar sobre ella
im_rgb = gray2rgb(im).astype(np.float32)

# Seleccionamos una ventana  $11 \times 11$  alrededor del punto original
t = x - 5 if x - 5 >= 0 else 0
b = x + 5 + 1
l = y - 5 if y - 5 >= 0 else 0
r = y + 5 + 1
window = im_rgb[t:b, l:r]

# Interpolamos con zoom de  $5\times$ 
window = cv2.resize(window, None, fx = ZOOM, fy = ZOOM)

# Dibujamos en rojo el punto original en el centro
window = cv2.circle(window, (ZOOM * 5 + 1, ZOOM * 5 + 1), 3, (255, 0, 0))

# Dibujamos en verde el punto corregido
```

```

window = cv2.circle(window, (int(ZOOM * (5 + ry - y) + 1),
                             int(ZOOM * (5 + rx - x) + 1)), 3, (0, 255, 0))

```

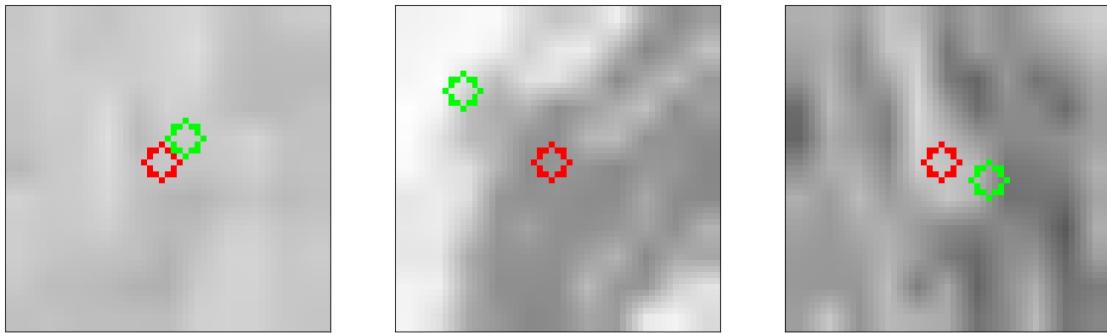


Figura 5: Coordenadas detectadas (rojo) y corregidas (verde) en Yosemite1.jpg

En las fotos de Yosemite no se aprecia del todo bien el resultado, pues hay pocas esquinas claramente diferenciadas. Para ilustrar mejor el funcionamiento, consideramos la imagen Tablero1.jpg, donde sí se encuentran las cuatro esquinas de cada cuadrado negro claramente destacadas.

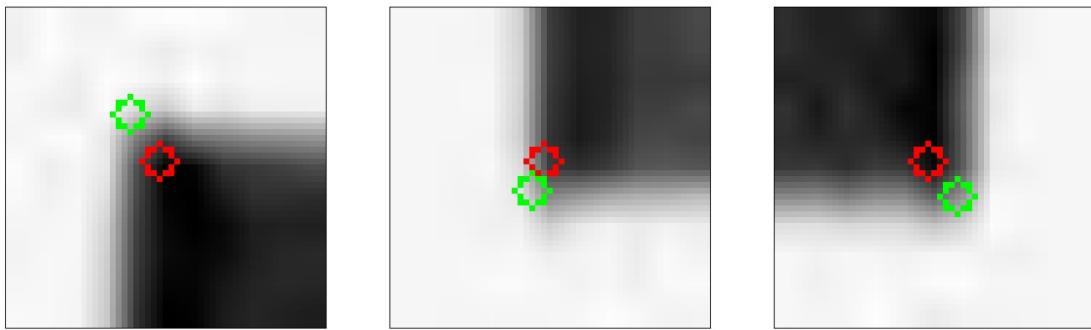


Figura 6: Coordenadas detectadas (rojo) y corregidas (verde) en Tablero1.jpg

Vemos como las coordenadas corregidas se aproximan mejor a las esquinas, aunque sea por poco. Recordamos que aquí la precisión es esencial.

Ejercicio 2: correspondencias

Este ejercicio pretende estudiar las correspondencias entre puntos extraídos de dos imágenes usando el descriptor AKAZE. Para ello, detectamos y extraemos los *keypoints* y sus descriptores con `detectAndCompute`:

```
kp, desc = cv2.AKAZE_create().detectAndCompute(im, None)
```

Para establecer las correspondencias utilizaremos dos criterios distintos. El primero se trata de `BruteForce+crossCheck`, en el que un punto se corresponde con el punto con descriptor más cercano en la otra imagen. El parámetro `crossCheck` obliga a que esta correspondencia sea en ambos sentidos.

```

def matches_bruteforce(desc1, desc2):
    # Creamos el objeto BFMatcher con crossCheck
    bf = cv2.BFMatcher_create(normType = cv2.NORM_HAMMING, crossCheck = True)

    # Calculamos los matches entre los descriptores de las imágenes
    return bf.match(desc1, desc2)

```

El segundo criterio, Lowe-Average-2NN, considera para cada descriptor los dos más cercanos en la otra imagen, y para reducir la ambigüedad lo considera como bueno únicamente si supera el test siguiente [2]: el cociente entre la distancia más cercana y la segunda más cercana debe ser menor que un cierto umbral, que fijamos a 0.75.

```

def matches_lowe_2nn(desc1, desc2):
    # Creamos el objeto BFMatcher
    bf = cv2.BFMatcher_create(normType = cv2.NORM_HAMMING)

    # Calculamos los 2 mejores matches entre los descriptores de las imágenes
    matches = bf.knnMatch(desc1, desc2, k = 2)

    # Descartamos correspondencias ambiguas según el criterio de Lowe
    selected = []
    for m1, m2 in matches:
        if m1.distance < 0.75 * m2.distance:
            selected.append(m1)

    return selected

```

En ambos casos usamos la distancia de Hamming en el cálculo de distancias, pues en OpenCV AKAZE proporciona descriptores binarios por defecto. Para mostrar los resultados en las imágenes de Yosemite, seleccionamos aleatoriamente 100 puntos como mucho, y mostramos mediante rectas las correspondencias obtenidas en cada caso, con la función drawMatches:

```

im_matches_bf = cv2.drawMatches(im1, kp1, im2, kp2, matches, None,
                                flags = cv2.DRAW_MATCHES_FLAGS_NOT_DRAW_SINGLE_POINTS)

```

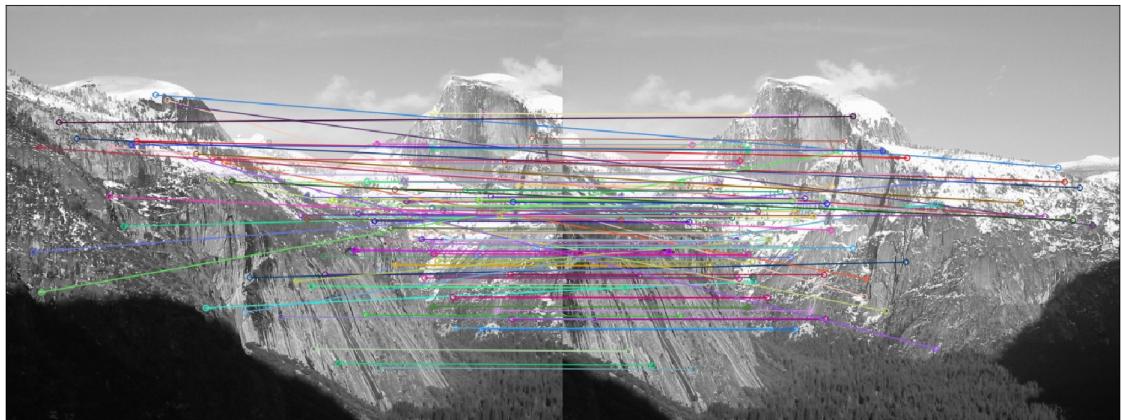


Figura 7: Correspondencias en Yosemite con BF+crossCheck

Con el primer criterio no conseguimos un resultado muy acertado, pues por la forma en la que están dispuestas las imágenes, las correspondencias deberían estar representadas por líneas (casi) horizontales. Si bien es cierto que hay parejas en correspondencia casi perfecta, otras están muy lejos de serlo, como vemos por las líneas oblicuas que aparecen.

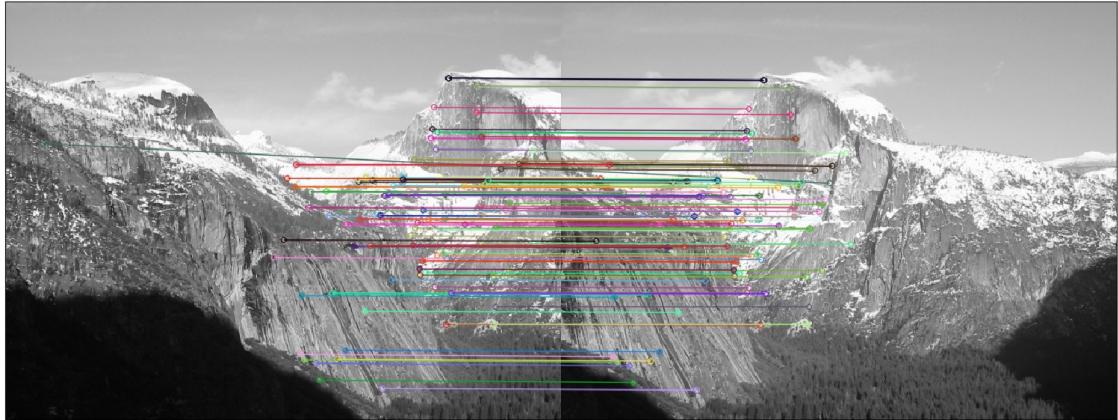


Figura 8: Correspondencias en Yosemite con Lowe-Average-2NN

Con el segundo criterio el resultado es mucho mejor: casi todas las líneas de unión son horizontales como se esperaría. Esto se debe a haber descartado los puntos ambiguos con el test del ratio, y solo quedarnos con aquellos de los que estamos razonablemente seguros que están en correspondencia. Notamos que los puntos de origen y fin se corresponden con la región de solapamiento de ambas imágenes.

Si realizamos pruebas sobre otro par de imágenes, por ejemplo `mosaico002.jpg` y `mosaico003.jpg`, obtenemos resultados similares. El criterio de Lowe vuelve a imponerse, sin necesitar un tiempo de ejecución apreciablemente mayor en este ejemplo.

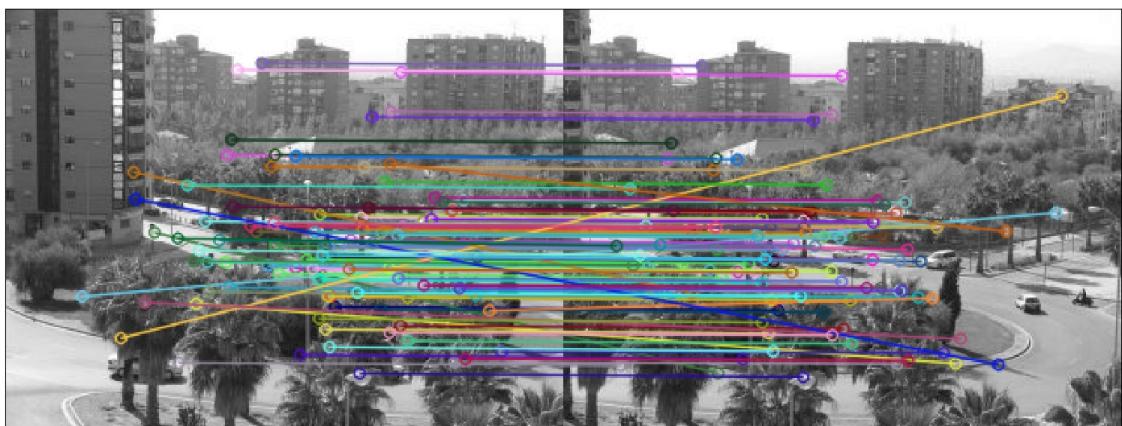


Figura 9: Correspondencias en mosaico00X con BF+crossCheck

Ejercicio 3: mosaico a partir de dos imágenes

Vamos ahora a construir un mosaico a partir de las imágenes `Yosemite1.jpg` y `Yosemite2.jpg`. En primer lugar, definimos un *canvas* donde irá la imagen resultante,



Figura 10: Correspondencias en mosaico00X con Lowe-Average-2NN

en el que encajamos directamente la primera imagen. Posteriormente, debemos estimar una homografía desde la segunda imagen a la primera, para después componer con la homografía de la primera imagen al mosaico (en este caso, la identidad) y obtener una homografía que lleve la segunda imagen en el mosaico.

```
# Definimos un canvas ajustado manualmente
h, w = im1.shape[0], 940
canvas = np.zeros((h, w, 3), dtype = np.float32)

# La homografía que lleva la primera imagen al mosaico es la identidad
canvas[:im1.shape[0], :im1.shape[1]] = im1
```

Para estimar la homografía H_{21} necesitaremos los descriptores y las correspondencias **de la segunda imagen a la primera**, calculadas de acuerdo al apartado anterior. Una vez conseguidos, la función `findHomography` nos permite estimar la homografía buscada:

```
def get_homography(im1, im2):
    kp1, desc1 = akaze_descriptor(im1)
    kp2, desc2 = akaze_descriptor(im2)

    matches = matches_lowe_2nn(desc1, desc2)

    query = np.array([kp1[match.queryIdx].pt for match in matches])
    train = np.array([kp2[match.trainIdx].pt for match in matches])

    return cv2.findHomography(query, train, cv2.RANSAC)[0]

# Estimamos homografía de la 2 a la 1
H21 = get_homography(im2, im1)
```

Esta función utiliza la relación entre los puntos dados para obtener los coeficientes de la homografía resolviendo un problema de mínimos cuadrados. Le decimos que emplee el método RANSAC para conseguir una estimación robusta. Ya solo nos queda trasladar la segunda imagen al mosaico utilizando la homografía encontrada. Para ello empleamos la

función `warpPerspective`, que aplica una homografía a una imagen guardando el resultado en un destino que se pasa como parámetro (en este caso, el *canvas*):

```
# Elegimos bordes transparentes con el método de extrapolación BORDER_TRANSPARENT
canvas = cv2.warpPerspective(im2, H21, (w, h), dst = canvas,
                             borderMode = cv2.BORDER_TRANSPARENT)
```

Es posible que al aplicar la homografía obtengamos puntos que no se encuentran en la otra imagen, por lo que se realiza extrapolación de píxeles.



Figura 11: Mosaico construido a partir de Yosemite1.jpg y Yosemite2.jpg

Apreciamos cómo se nota ligeramente el cambio de imagen en la región de solapamiento, debido también a la diferencia de color, pero en general el resultado final es bueno.

Ejercicio 4: mosaico a partir de N imágenes

Este ejercicio es similar al anterior, solo que ahora tenemos que construir un mosaico con un número arbitrario de imágenes. Suponemos que tenemos las imágenes ordenadas de la 0 a la N-1 en una lista llamada `ims`, y que existen correspondencias entre cada dos imágenes (la 0 con la 1, la 2 con la 3, etc.).

La estrategia que seguiremos ahora es ajustar manualmente un canvas en el que quepan todas las imágenes, y trasladar a él inicialmente la imagen central. Esta vez, la homografía H_0 que se encarga de ello no es la identidad, sino que es una translación no trivial:

```
k = len(ims) // 2

# Las dimensiones del canvas son (h, w)
tx = (w - ims[k].shape[1]) / 2
ty = (h - ims[k].shape[0]) / 2
H0 = np.array([[1, 0, tx], [0, 1, ty], [0, 0, 1]])
```

```
# Trasladamos la imagen central al mosaico
canvas = cv2.warpPerspective(ims[k], H0, (w, h), dst = canvas,
                             borderMode = cv2.BORDER_TRANSPARENT)
```

Vamos ahora a estimar las homografías entre cada dos imágenes. Suponemos que la imagen central es la k -ésima, y que la homografía $H_{i,j}$ lleva la imagen i -ésima, I_i , en la j -ésima, I_j . Queremos obtener una cadena como esta:

$$I_0 \xrightarrow{H_{0,1}} I_1 \longrightarrow \cdots \longrightarrow I_{k-1} \xrightarrow{H_{k-1,k}} I_k \xleftarrow{H_{k+1,k}} I_{k+1} \longleftarrow \cdots \longleftarrow I_{N-2} \xleftarrow{H_{N-1,N-2}} I_{N-1}$$

Es decir, para las imágenes a la izquierda de la central queremos estimar la homografía entre cada imagen y la siguiente, mientras que para las que están a la derecha de la central queremos estimar homografías entre una imagen y la anterior. Esto lo conseguimos mediante el siguiente bucle:

```
# Calculamos las homografías entre cada dos imágenes
homographies = []
for i in range(len(ims)):
    if i != k:
        j = i + 1 if i < k else i - 1
        homographies.append(get_homography(ims[i], ims[j]))

    else: # No se usa la posición central
        homographies.append(np.array([]))
```

A la hora de trasladar las imágenes al mosaico, tomaremos la central como referencia. La estrategia para trasladar la imagen I_i será llevarla mediante una composición de homografías convenientes a la imagen I_k , para posteriormente componer con la traslación de esta al mosaico, H_0 . De nuevo diferenciamos dos casos:

- Si $i < k$, realizamos la cadena de homografías $H_0 \circ H_{k-1,k} \circ \cdots \circ H_{i,i+1}$, que resulta en el producto matricial

$$H_0 H_{k-1,k} \cdots H_{i,i+1}$$

- Si $i > k$, realizamos la cadena de homografías $H_0 \circ H_{k+1,k} \circ \cdots \circ H_{i,i-1}$, que resulta en el producto matricial

$$H_0 H_{k+1,k} \cdots H_{i,i-1}$$

Aprovechamos que podemos reutilizar las transformaciones calculadas. Vamos recorriendo las imágenes por parejas, comenzando por las que rodean a la imagen central, y avanzando hacia los extremos. En cada paso componemos con la homografía correspondiente y proyectamos las imágenes en el *canvas* mediante la homografía resultante. Recordamos que las homografías tienen distinto significado a un lado y a otro de la imagen central.

```
# Trasladamos el resto de imágenes al mosaico
H = H0
G = H0
for i in range(k)[::-1]:
    H = H @ homographies[i]
    canvas = cv2.warpPerspective(ims[i], H, (w, h), dst = canvas,
```

```

borderMode = cv2.BORDER_TRANSPARENT)

j = 2 * k - i
if j < len(ims):
    G = G @ homographies[j]
    canvas = cv2.warpPerspective(ims[j], G, (w, h), dst = canvas,
                                borderMode = cv2.BORDER_TRANSPARENT)

```

Realizamos ahora el mosaico para las imágenes de Yosemite. Como no hay correspondencias entre las imágenes `Yosemite4.jpg` y `Yosemite5.jpg`, realizamos dos mosaicos: uno de la 1 a la 4 y otro de la 5 a la 7.

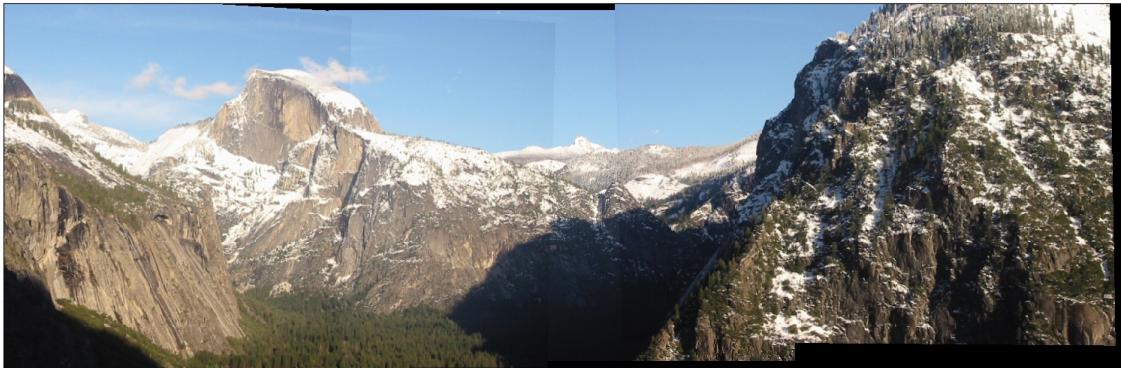


Figura 12: Mosaico a partir de `Yosemite{1-4}.jpg`

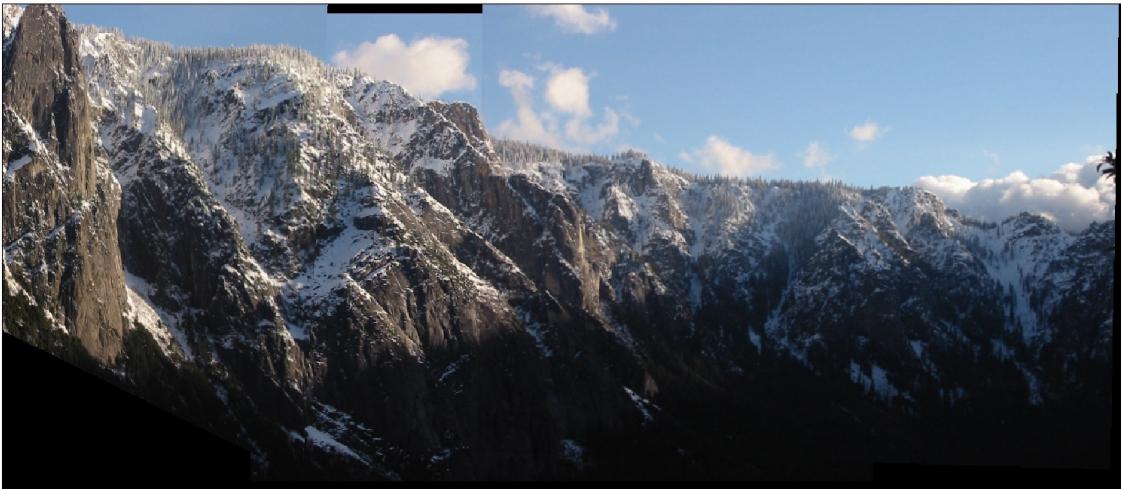


Figura 13: Mosaico a partir de `Yosemite{5-7}.jpg`

Podemos ver cómo las imágenes no encajan perfectamente. A la hora de estimar las matrices de homografía y de aplicarlas a una imagen se introducen defectos visuales, debidos también a errores en la detección de los descriptores y el cálculo de correspondencias. También se produce el efecto de propagación de errores, haciendo que el resultado final no sea perfecto. Sin embargo, es lo suficientemente agradable estéticamente.

Por último, construimos un mosaico con 10 fotos tomadas cerca de la ETSIIT. En este caso el mosaico parece encajar mejor, y aunque no podemos evitar el efecto visual en el

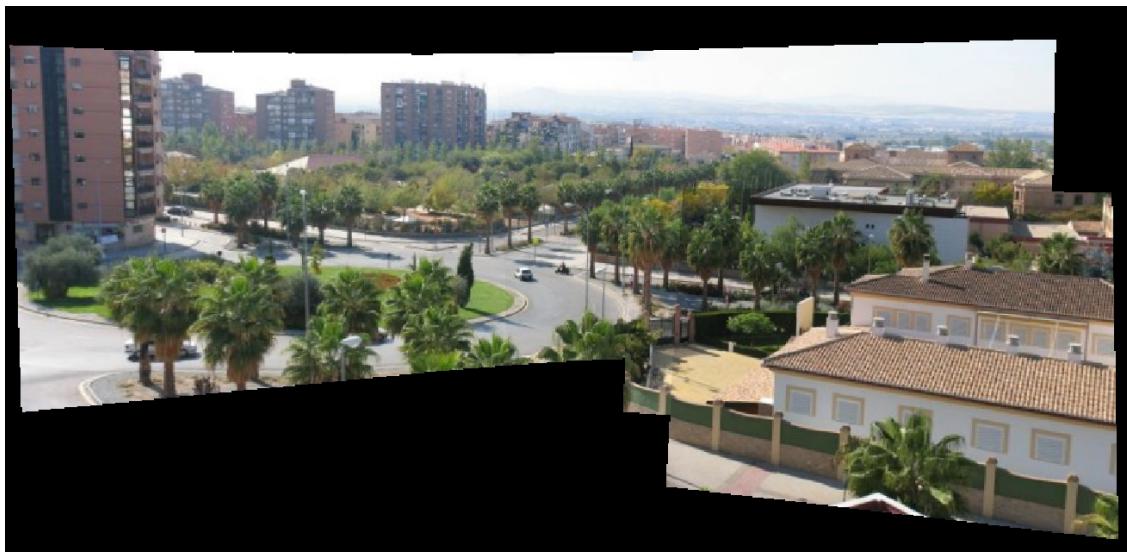


Figura 14: Mosaico a partir de mosaico{002-011}.jpg

que la imagen se curva, no se notan tanto los cambios entre una imagen y otra, y tampoco hay mucho solapamiento.

Bibliografía

- [1] Multi-Image Matching using Multi-Scale Oriented Patches
- [2] Distinctive Image Features from Scale-Invariant Keypoints