Detección de puntos relevantes y construcción de panoramas

Luis Antonio Ortega Andrés

22 de diciembre de 2019

Ejercicio 1

En este ejercicio se nos pide realizar la detección de puntos Harris sobre una pirámide Gaussiana. Dividiremos los cálculos entre dos funciones distintas, `harris_detector` se encargará de trabajar sobre la pirámide Gaussiana y calcular los puntos Harris, mientras que `corners` se encargará de calcular las coordenadas subpixel.

Comencemos por el detector Harris, los pasos seguidos para calcular los «KeyPoints» son los siguientes. Trataremos todo el rato con la escala de grises de la imagen a tratar.

- Añadimos un borde negro a la imagen hasta que tenga un tamaño potencia de 2 en ambos ejes, esto nos asegurará que las transformaciones de coordenadas que hagamos más tarde sean exactas.
- Calculamos la pirámide Gaussiana, incluyendo la imagen original.

  ```
  piramide = [gray] + piramide_gaussiana(gray, 1, scales)
  ```

- Utilizamos un filtro de Sobel sobre un alisado son $\sigma = 4.5$ para calcular el gradiente de la imagen. Esto nos da dos imágenes, donde cada una de ellas nos da la intensidad del gradiente en un eje.

  ```
  blured_img = cv.GaussianBlur(gray, ksize = (0,0), sigmaX = 4.5)

  g1 = cv.Sobel(blured_img, -1, 1, 0)
  g2 = cv.Sobel(blured_img, -1, 0, 1)
  ```

- Construimos la piramide Gaussiana de ambas imágenes, incluyéndolas también. Esto nos permitirá conocer el gradiente en cada una de las escalas de la imagen original.
- Recorremos cada una de las escalas de la pirámide de nuestra imagen original y utilizamos la función `cornerEigenValsAndVecs` para calcular los valores propios de la matriz de covarianzas de las derivadas en un entorno de cada punto. Necesitaremos pasarle dos valores, el tamaño del vecindario `block_size` y un tamaño para la máscara de Sobel que implementa, `ksize`. Ambos valores serán 3.

  ```
  data = cv.cornerEigenValsAndVecs(piramide[scale], block_size, ksize)

  e1 = data[:,:,0]
  e2 = data[:,:,1]
  ```

  La función nos devuelve una matriz con 6 valores en cada posición. Sólo nos interesan los dos primeros que corresponden a los valores propios.

- Calculamos la matriz formada por las medias armónicas de los valores propios en cada punto. En aquellas posiciones donde los valores propios son opuestos, insertamos un 0.

```
h_means = harmonic_mean(e1, e2)
h_means[np.isnan(h_means)] = 0
```

- Calculamos los máximos locales de dicha matriz. Para ello utilizamos la función `peak_local_max` presente en la librería `skimage` (source). Esta función nos permite seleccionar la distancia mínima entre máximos, cuantos máximos buscar y añadir un umbral para supresión de no máximos.

  Para la distancia utilizaremos `block_size//2` que correspondera a buscar en una ventana de tamaño `block_size`, buscaremos un máximo de 1000 puntos utilizando un umbral de 15. Estos valores se han escogido para más tarde cumplir el requisito de disponer de más de 2000 puntos entre todas las escalas.

```
peaks = peak_local_max(h_means, min_distance = block_size,
                       num_peaks = n_keypoints, threshold_abs = threshold)
```

- Para cada uno de los puntos obtenidos, calculamos el ángulo de la dirección del gradiente utilizando el valor de la escala del mismo en dicho punto. Para hacerlo, sean `dx` y `dy` las correspondientes escalas del gradiente, calculamos la norma del vector.

```
norm = np.sqrt(dx[x][y] * dx[x][y] + dy[x][y] * dy[x][y])
```

  Calculamos el seno y el coseno y lo utilizamos para calcular el valor del ángulo (le sumamos 180º para que esté entre 0 y 360).

```
sin = dy[x][y] / norm if norm > 0 else 0
cos = dx[x][y] / norm if norm > 0 else 0
angle = np.degrees(np.arctan2(sin, cos)) + 180
```

  Respecto al tamaño del `KeyPoint`, utilizamos `block_size*(escalas-escala_actual)` de forma que sean visibles en las escalas más pequeñas.

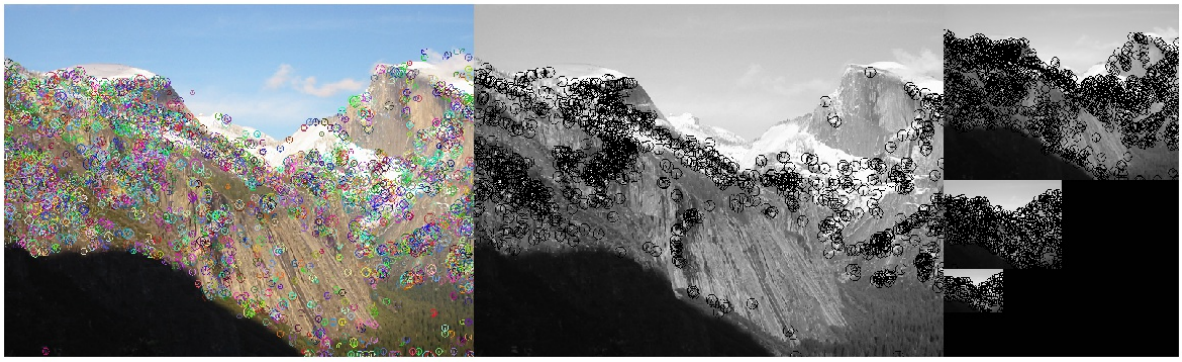
- Utilizamos la función `drawKeyPoints` para añadir dichos puntos a su escala correspondiente y a la imagen original (teniendo en cuenta la transformación sobre las coordenadas del punto). Utilizamos el flag `5` correspondiente a que se utilice el tamaño los puntos al pintarlos.

La función nos devuelve tanto las imágenes ya tratadas como el vector de `KeyPoints`.

Veamos los resultados de aplicarla a la imagen `Yosemite1`.
Obtenemos los siguientes máximos en cada escala:

```
Máximos en la escala  0 :  1000
Máximos en la escala  1 :  1000
Máximos en la escala  2 :  521
Máximos en la escala  3 :  154
```

La imagen de la izquierda es la imagen original con todos los keypoints de todas las escalas.



Podemos observar que los puntos que hemos detectado son representativos de cada escala, ya que marcan las esquinas existentes en toda la imagen y no solo en una parte. Podemos ver que se obtienen un número considerable de puntos.

Utilizamos ahora el vector de `KeyPoints` para obtener las coordenadas subpixel de los mismos. Para ello, en la función `corners` tomamos dicho vector y la imagen original.

Para utilizar la función `cornerSubPix` necesitamos definir un tamaño de ventana (escogeremos 3×3) y un criterio de finalización (utilizaremos tanto un número máximo de iteraciones, 100, como un valor mínimo de corrección por iteración, 0.1). El parámetro `zero_zone` lo anulamos poniendolo a $(-1, -1)$.

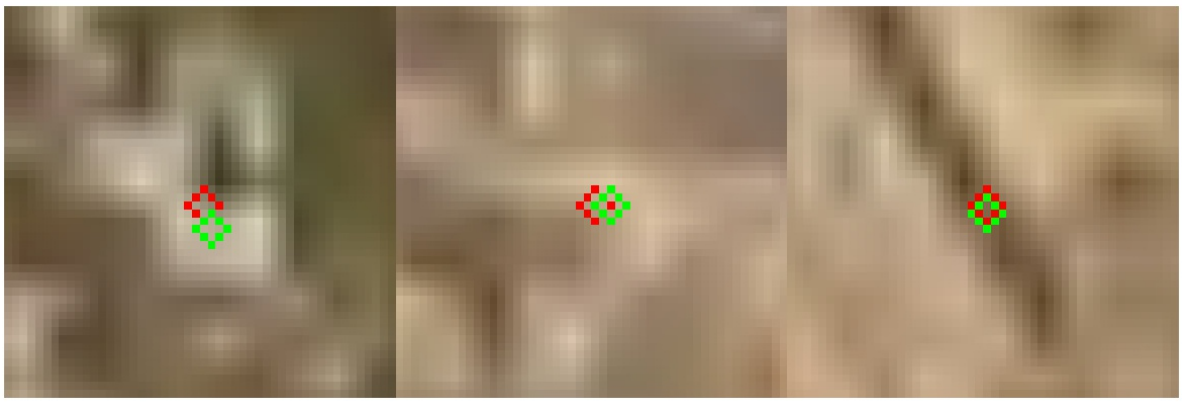
```
win_size = (3,3)
zero_zone = (-1, -1)
criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 100, 0.1)
```

Llamamos a dicha función sobre la imagen en escala de grises y tomamos una muestra aleatoria de 3 puntos corregidos y sus correspondientes sin corregir.

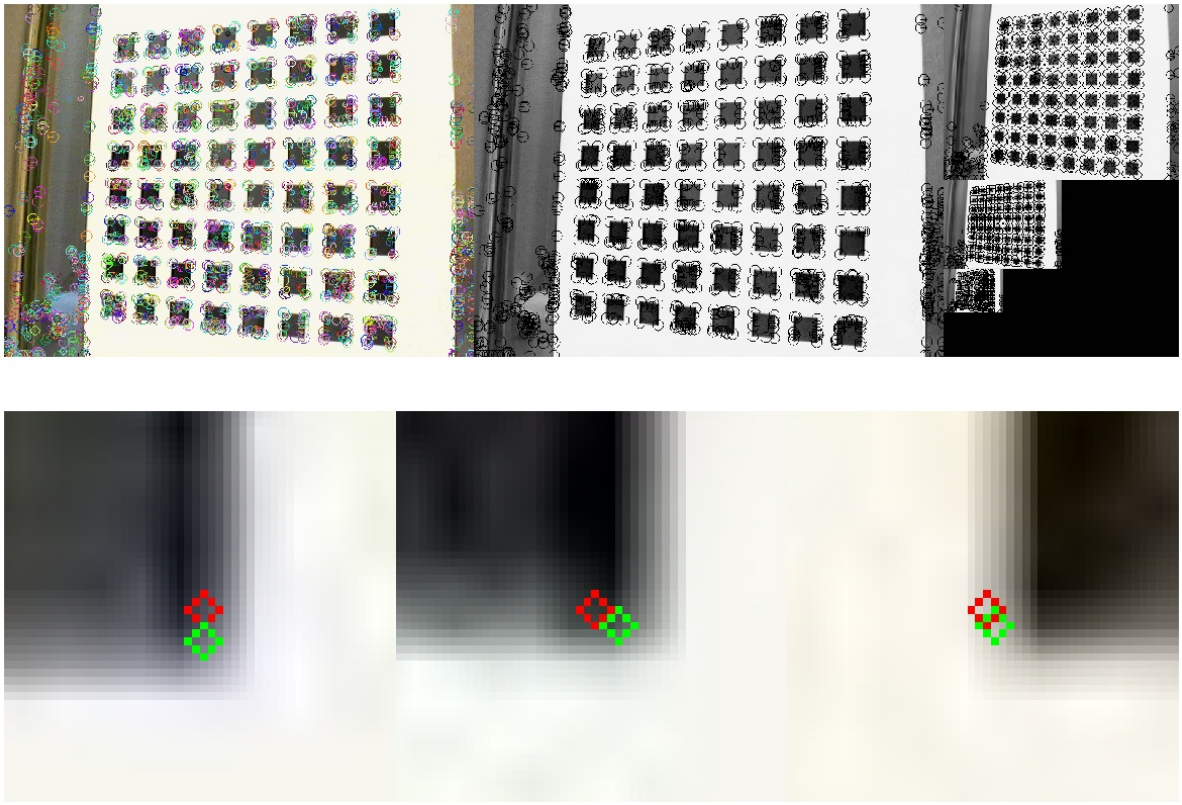
```
cv.cornerSubPix(gray, corners, win_size, zero_zone, criteria)
indexes = random.sample(range(0, len(points) - 1), 3)
```

Para cada uno de ellos aplicamos un zoom ×5 y tomamos una ventana de 10×10 a su alrededor. En ella pintamos ambos valores (el rojo sin corregir y el verde corregido).

Veamos los resultados.



Para apreciar mejor los resultados, vamos a mostrarlos también en la imagen del tablero donde es más sencillo ver las esquinas.



Podemos ver como la corrección se encuentra más próxima a la esquina real.

Ejercicio 2

En este ejercicio se busca extraer los descriptores AKAZE usando la función `detectAndCompute` para luego utilizar el objeto `BFMatcher` para establecer correspondencias entre dos imágenes.

Para ello declaramos 4 funciones que modularizarán la tarea. En la función `akaze_descriptor` obtenemos los puntos y descriptores de una imagen dada.

```
def akaze_descriptor(img):
    return cv.AKAZE_create().detectAndCompute(img, None)
```

Definimos también las funciones encargadas de construir las correspondencias entre dos imágenes. Para ello `match_bf` toma los descriptores de ambas y las construye haciendo uso de `BFMatcher` de OpenCV. Para utilizar `crossCheck` solo tenemos que utilizar el flag con el mismo nombre en el constructor del matcher. Para utilizar fuerza bruta solo tenemos que llamar al método `match` que implementa.

```
def match_bf(desc1, desc2):

    matcher = cv.BFMatcher_create(crossCheck=True)
    matches = matcher.match(desc1,desc2)

    return matches
```

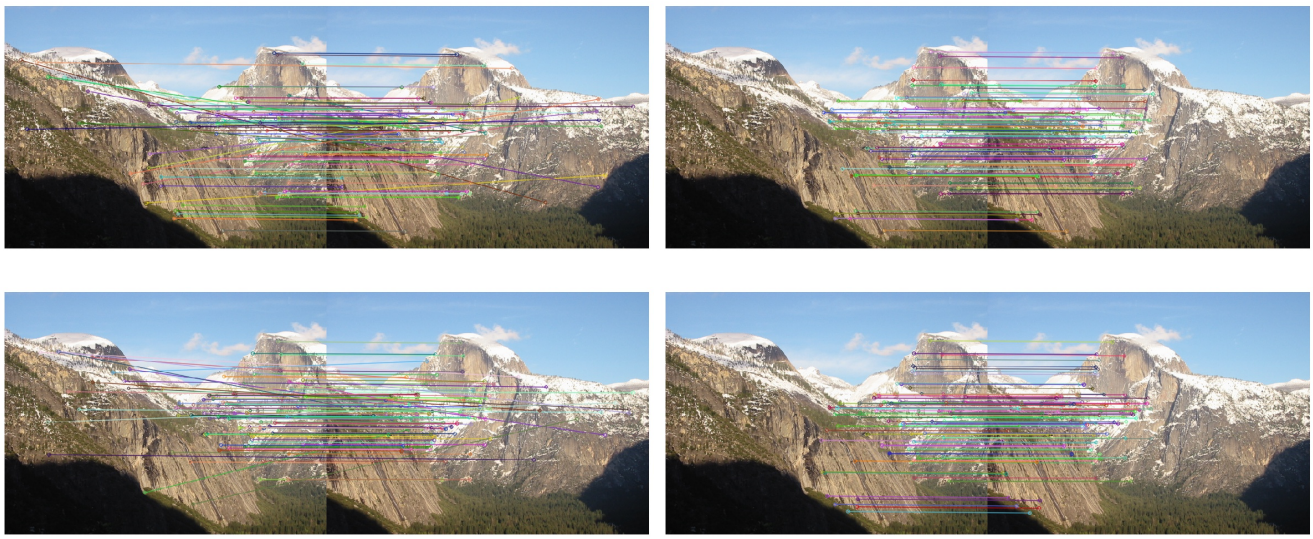
Para hacer «Lowe-Average-2NN» utilizamos el método `knnMatch` en lugar de `match`. Con el resultado del mismo, comparamos la distancia entre el más cercano y el segundo más cercano, si son lo suficientemente parecidos, descartamos el punto. Para determinar esto utilizamos un factor de proporcionalidad de 0.75.

```
def match_2nn(desc1, desc2):

    matcher = cv.BFMatcher_create()
    matches = matcher.knnMatch(desc1, desc2, k = 2)

    ret = []
    for m, n in matches:
        if m.distance < 0.75 * n.distance:
            ret.append(m)

    return ret
```

La función `get_matches` llamará a las anteriores para calcular las correspondencias entre dos imágenes dadas. Luego llamará a la función de OpenCV `drawMatches` para construir la imagen final y las devuelve.

```
bf_matches = random.sample(match_bf(desc1, desc2), 100)
bf_img = cv.drawMatches(img1, kpts1, img2, kpts2, bf_matches, None,
                        flags = cv.DRAW_MATCHES_FLAGS_NOT_DRAW_SINGLE_POINTS)


nn_matches = random.sample(match_2nn(desc1, desc2), 100)
nn_img = cv.drawMatches(img1, kpts1, img2, kpts2, nn_matches, None,
                        flags = cv.DRAW_MATCHES_FLAGS_NOT_DRAW_SINGLE_POINTS)
```

Veamos los resultados en un par de ejemplos aleatorios con una muestra de 100 correspondencias. A la izquierda los resultados utilizando fuerza bruta y a la derecha `Lowe_average_2nn`.



En estos podemos observar como el método por fuerza bruta comete una mayor cantidad de errores. Vemos que debido a la disposición de las imágenes, las correspondencias deben ser todas lineas (casi) horizontales como ocurre al utilizar `Lowe_average_2NN`.

Ejercicio 3

En este ejercicio debemos construir un mosaico a partir de 2 imágenes relacionadas por homografías. Para ello construimos un «canvas» donde mostraremos ambas imágenes, por lo que este debe tener el tamaño adecuado. La homografía que lleva la primera imagen al canvas es la identidad. Por ello, insertamos la primera imagen al principio del canvas.

```
canvas[:img1.shape[0], :img1.shape[1]] = img1
```

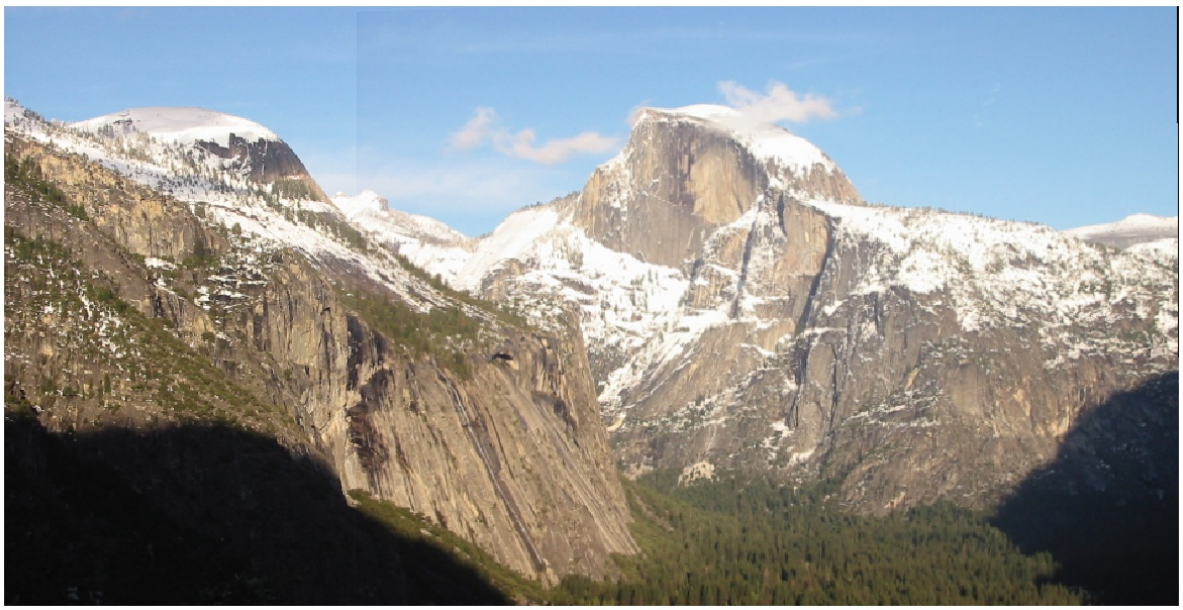
Ahora calculamos la homografía de la segunda imagen en la primera, para ello utilizamos el descriptor AKAZE y buscamos las correspondencias utilizando `Lowe-Average-2NN` como en el apartado anterior.
Utilizamos dichas correspondencias para construir los arrays que contienen los puntos de ambas imágenes que vamos a relacionar.

```
kpts1, desc1 = akaze_descriptor(img1)
kpts2, desc2 = akaze_descriptor(img2)

matches = match_2nn(desc2, desc1)

q = np.array([kpts2[match.queryIdx].pt for match in matches])
t = np.array([kpts1[match.trainIdx].pt for match in matches])
```

Usamos ahora dichos arrays para construir la homografía utilizando la función `findHomography` de OpenCV. Esto junto con `warpPerspective` nos construye el canvas final. Utilizamos un borde transparente para no pisar la primera imagen.

```
canvas = cv.warpPerspective(img2, H_21, (w, h),
                            dst = canvas, borderMode = cv.BORDER_TRANSPARENT)
```

Veamos el resultado con las dos primeras imágenes de Yosemite.



Ejercicio 4

En este ejercicio queremos generalizar el ejercicio anterior a un número arbitrario de imágenes. Para ello vamos a utilizar la imagen central como referencia para añadir el resto de imágenes. La buscamos y creamos un canvas con el suficiente tamaño.

```
index_img_center = len(imgs)//2
img_center =  imgs[index_img_center]

w = sum([img.shape[1] for img in imgs])
h = imgs[0].shape[0]*2

canvas = np.zeros( (h, w, 3), dtype = np.float32)
```

En primer lugar construimos la homografía que incluye esta imagen en el canvas. Es sencillo pues al ser la primera solo tenemos que trasladarla al centro.

```
H_0 = np.array([
    [1, 0, (w - img_center.shape[1])/2],
    [0, 1, (h - img_center.shape[0])/2],
    [0, 0, 1]])

canvas = cv.warpPerspective(imgs[index_img_center], H_0, (w, h),
                                dst = canvas, borderMode = cv.BORDER_TRANSPARENT)
```

Llamemos H_{ij} a la homografía que nos lleva la imagen `i`-ésima en la `j`-ésima, y H_0 la que hemos definido antes. Supongamos ahora que la imagen central se encuentra en el índice k.
Para construir el resto de homografías debemos diferenciar a que lado se encuentra la imagen.

- Para imágenes a la izquierda de la central $i < k$. Construiremos la siguiente homografía para incluir la imagen
$$H_0 \cdot H_{k-1,k} \cdots H_{i,i+1}$$
- Para imágenes a la derecha de la central $i > k$. Construiremos la siguiente homografía para incluir la imagen
$$H_0 \cdot H_{k+1,k} \cdots H_{i,i-1}$$

Calculamos entonces las homografías consecutivas.

```
Hom = []
for i in range(len(imgs)):
    if i != index_img_center:
        j = i + 1 if i < index_img_center else i - 1
        Hom.append(get_homography(imgs[i], imgs[j]))
    else:
        Hom.append(np.array([]))
```

Donde la función auxiliar `get_homography` devuelve la homografía calculándola de la misma forma que en el ejercicio anterior.

Añadimos las imágenes al canvas en un solo bucle apoyandonos de la simetría de los índices a un lado y al otro del centro.

```
H = H_0
G = H_0
for i in range(index_img_center)[::-1]:
    H = H @ Hom[i]
    canvas = cv.warpPerspective(imgs[i], H, (w, h),
                                dst = canvas, borderMode = cv.BORDER_TRANSPARENT)

    j = 2 * index_img_center - i
    if j < len(imgs):
        G = G @ Hom[j]
        canvas = cv.warpPerspective(imgs[j], G, (w, h),
                                    dst = canvas,
                                    borderMode = cv.BORDER_TRANSPARENT)
```

Veamos los resultados con las imágenes que tenemos.

Figura 1: Yosemite 1-4

Como se puede observar existen errores en el mosaico, probablemente provocados por que las homografías no son perfectas, sino que son estimaciones utilizando los Keypoints.

Figura 2: Yosemite 5-7