



*ugr* | Universidad  
de **Granada**

## Grado en Ingeniería Informática. Cuarto.

### Práctica 2.

---

**Nombre de la asignatura:**

Visión por Computador. Jueves de 10:30 a 12:30.

**Realizado por:**

Néstor Rodríguez Vico. DNI: 75573052C.

email: [nrv23@correo.ugr.es](mailto:nrv23@correo.ugr.es)



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS  
INFORMÁTICA Y DE TELECOMUNICACIÓN.

---

Granada, 21 de noviembre de 2017.

# Índice

<b>1 Aclaración.</b>	<b>3</b>
<b>2 Ejercicio 1. Detección de puntos Harris multiescala.</b>	<b>3</b>
2.1 Apartado a. Escribir una función que extraiga la lista potencial de puntos Harris a distintas escalas de una imagen de nivel de gris. . . . .	3
2.2 Apartado b. Refinamiento de puntos Harris. . . . .	5
2.3 Apartado c. Orientación. . . . .	6
2.4 Apartado d. Descriptores. . . . .	7
2.5 Resultado. . . . .	7
<b>3 Ejercicio 2.</b>	<b>10</b>
<b>4 Ejercicios 3 y 4. Construcción de un mosaico.</b>	<b>16</b>

# 1. Aclaración.

Se han usado funciones de la práctica anterior que no se van a mostrar en esta memoria para no repetir.

## 2. Ejercicio 1. Detección de puntos Harris multiescala.

### 2.1. Apartado a. Escribir una función que extraiga la lista potencial de puntos Harris a distintas escalas de una imagen de nivel de gris.

Para realizar este apartado se han implementado varias funciones. La primera de ellas, *harris\_matrix\_binary* se encarga de calcular la matriz harris para una imagen dada. El tamaño de *block\_size* usado ha sido 7 (se nos dice que usemos un  $\sigma = 1,5$  así que:  $block\_size = 3 * \text{int}(\sigma) + 1 + 3 * \text{int}(\sigma) = 7$ ). El tamaño de *ksize* usado ha sido 7 (se nos dice que usemos un  $\sigma = 1$  así que:  $ksize = 3 * \text{int}(\sigma) + 1 + 3 * \text{int}(\sigma) = 7$ ). Esta función también devuelve una matriz binaria (initializada a 255). El criterio de selección empleado es el criterio Harris, con un valor de  $k = 0,04$ . La fórmula de dicho criterio es la siguiente:

$$f = \lambda_1\lambda_2 - k * (\lambda_1 + \lambda_2)^2$$

Donde  $\lambda_1\lambda_2$  representa el determinante de la matriz Harris y  $\lambda_1 + \lambda_2$  representa la traza. Nosotros vamos a calcular  $\lambda_1$  y  $\lambda_2$  como los autovalores de la matriz asociada a la imagen.

```
1 def harris_matrix_binary(src , block_size , ksize , k=0.04, border=0):
2     eigenvals = cv2.cornerEigenValsAndVecs(src=src , blockSize=block_size , ksize=ksize ,
3         borderType=border)
4     harris_matrix = np.zeros(shape=np.shape(src))
5
6     first = eigenvals[:, :, 0]
7     second = eigenvals[:, :, 1]
8
9     for i in range(np.shape(src)[0]):
10        for j in range(np.shape(src)[1]):
11            harris_matrix[i][j] = (first[i][j] * second[i][j]) - k * ((first[i][j] +
12                second[i][j]) ** 2)
13
# Return the harris matrix and a binary one (0 or 255)
return harris_matrix , np.ones(shape=np.shape(src)) * 255
```

El siguiente paso es la supresión de valores no máximos. Para ello se han implementado la función *suppress\_non\_max*. Esta función se apoya de la función *is\_center\_max*, la cual devuelve verdadero si el centro de una ventana es el valor máximo de la ventana y falso en caso contrario, y de la función *make\_zeros*, la cual dada una ventana y una imagen, asigna cero a los elementos correspondientes a la ventana de la matriz. La función *suppress\_non\_max* se encarga de ir recorriendo la matrix binaria y ver si cada elemento es máximo local o no. En caso de serlo, pone toda la ventana local a cero. Para hacerlo de forma más eficiente, lo que hace es sólo preguntar si es máximo a aquello valores de la matriz binaria que son distintos de cero, ya que si es cero, sabemos que

no es el máximo puesto que en una consulta anterior se ha puesto a cero por no serlo. Para no perder los valores que son máximos, escribimos en la matriz binaria un 255, indicando que ese punto ( $i, j$ ) es máximo.

```

1 def suppress_non_max(binary, harris_matrix, window_size=7):
2     def is_center_max(windows):
3         return np.max(windows) == windows[np.shape(windows)[0] // 2][np.shape(windows)
4                                         [1] // 2]
5
6     def make_zeros(img, size, i, j):
7         # The center is i, j
8         img[i - size:i + size + 1, j - size:j + size + 1] = 0
9
10    # Add the border to not have problems with the window_size
11    # window_size is the full width
12    size = window_size // 2
13    harris_matrix_augmented = cv2.copyMakeBorder(src=harris_matrix, top=size, bottom=
14                                                 size, right=size, left=size, borderType=cv2.BORDER_CONSTANT)
15
16    for i in range(np.shape(binary)[0]):
17        for j in range(np.shape(binary)[1]):
18            if binary[i][j] != 0:
19                if is_center_max(harris_matrix_augmented[i:i + size + size + 1, j:j +
20                                                 size + size + 1]):
21                    make_zeros(binary, size, i, j)
22                    binary[i, j] = 255
23                else:
24                    binary[i, j] = 0

```

Con las funciones mencionadas anteriormente, podemos construir la siguiente función:

```

1 def harris_points(src, block_size=7, ksize=7, windos_size=7, n_best=500, border=cv2.
2 BORDER_CONSTANT):
3     points = []
4     harris = []
5
6     sigmas = np.arange(0, len(src))
7     for img, scale in zip(src, sigmas):
8         set = []
9         harris_matrix, binary = harris_matrix_binary(src=img, block_size=block_size,
10                                                       ksize=ksize, border=border)
11         suppress_non_max(binary=binary, harris_matrix=harris_matrix, window_size=
12                         windos_size)
13         harris.append(harris_matrix)
14
15         index_i, index_j = np.where(binary == 255)
16         for i, j in zip(index_i, index_j):
17             p = cv2.KeyPoint(x=j, y=i, _size=block_size, _octave=scale)
18             set.append(p)
19
20         points.append(set)
21
22     return get_n_best(src=harris, n=n_best, points=points), harris

```

Esta función recibe la pirámide Gaussiana en el argumento  $src$  y aplica las funciones anteriores para cada nivel de la pirámide. Tras aplicar las funciones, podemos obtener los *KeyPoints* asociados a cada nivel (líneas 12 a 15). Para obtener los puntos sólo debemos recoger los índices de la matriz binaria correspondiente cuyos elementos son 255 (línea 12).

Debemos tener cuidado al crear los *KeyPoints*. Si nosotros determinamos que el elemento  $(i, j)$  es un punto Harris, el valor  $x$  del *Keypoint* es  $j$ , no  $i$ , y el valor de  $y$  es  $i$ ,

no  $j$  (ver línea 14). Esto debemos tenerlo en cuenta para todo el resto de la práctica y no equivocarnos al pasar un *KeyPoint* a una matriz.

También debemos tener cuidado en como asignamos el campo *octave*. Yo lo he asignado a  $0$  para la escala original, a  $1$  para el primer nivel, a  $n$  para el nivel  $n$ .

El campo *\_size* representa el diámetro. Como en nuestro caso caso no tenemos un círculo sino una ventana cuadrada, el diámetro sería el lado de nuestro cuadrado que representa la ventana, es decir, *block\_size*.

Tras este proceso tenemos, para cada nivel de la pirámide, una matriz harris, una matriz binaria y un conjunto de *KeyPoints*. Una vez lo tenemos todo, lo que debemos hacer es coger los mejores *KeyPoints* de todos los encontrados, independientemente del nivel de la pirámide en el que se encuentren. Para ello hemos desarrollado la función *get\_n\_best*, cuyo resultado es el devuelto por la función *harris\_points*:

```

1 def get_n_best(src, n, points):
2     for img, set in zip(src, points):
3         index = [(int(np.round(p.pt[0])), int(np.round(p.pt[1]))) for p in set]
4         index_j = [t[0] for t in index]
5         index_i = [t[1] for t in index]
6         values = img[index_i, index_j]
7
8         for p, v in zip(set, values):
9             p.response = v
10
11    points_flatten = [p for set in points for p in set]
12    sorted_points = sorted(points_flatten, key=lambda x: x.response, reverse=True)
13
14    if len(sorted_points) > n:
15        return sorted_points[0:n]
16    else:
17        return sorted_points

```

Lo que hace esta función es asignar a todos los *KeyPoints* su valor de la matriz Harris correspondiente (líneas 2 a 9). Una vez los tenemos, juntamos todos los puntos de los distintos niveles (línea 11). Los ordenamos por su valor *response* de menor a mayor valor (es decir, primero los mejores puntos) (línea 12) y finalmente devolvemos el número indicado, argumento  $n$ , de puntos.

## 2.2. Apartado b. Refinamiento de puntos Harris.

Una vez ya tenemos los mejores puntos Harris, vamos a refinarlos. Para ello necesitaremos usar la función de *OpenCV cornerSubPix*. La idea es refinar los puntos encontrados en cada nivel en su nivel correspondiente de la pirámide. Para ello usaremos la siguiente función:

```

1 def refine_best_points(src, points, harris_matrix):
2     sets_points = [[] for _ in range(len(src))]
3
4     octaves = np.unique(np.array([p.octave for p in points]))
5     d = {}
6     for octave, i in zip(octaves, range(len(octaves))):
7         d[octave] = i

```

```

8
9     for p in points:
10        sets_points[d[p.octave]].append(p)
11
12    for img, set, i in zip(src, sets_points, range(len(src))):
13        only_points = [(p.pt[0], p.pt[1]) for p in set]
14        float_points = np.array(only_points, dtype=np.float32)
15        cv2.cornerSubPix(image=img, corners=float_points, winSize=(5, 5), zeroZone=(-1,
16                                         -1), criteria=(cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 40,
17                                         0.01))
18
19        for p, kp in zip(float_points, set):
20            kp.pt = (p[0], p[1])
21            # kp.response = harris_matrix[i][int(np.round(p[1]))][int(np.round(p[0]))]
22
# Delete outside points
return [[p for p in set if p.pt[1] < np.shape(src[d[p.octave]])[0] and p.pt[0] < np
       .shape(src[d[p.octave]])[1]] for set in sets_points]
```

Lo primero que hacemos es separar los puntos en distintos conjuntos en función del nivel en el que se han encontrado (esto lo sabemos gracias al campo *octave* de los *KeyPoints*) (líneas 2 a 10). Una vez tenemos los distintos conjuntos, extraemos únicamente las coordenadas, x e y, de los puntos y los pasamos a la función *cornerSubPix* junto con la imagen correspondiente de la pirámide Gaussiana. Para no tener problemas, los puntos deben pasarse como una matriz de dos columnas, una representa la x y otra la y. La matriz debe ser de tipo *float32*. Una vez hemos refinado los puntos, los asignamos a sus *KeyPoints* correspondientes actualizando su valor *responde* asociado (líneas 17 a 19). Finalmente, devolvemos los puntos refinados que no se salen de la imagen (línea 22).

### 2.3. Apartado c. Orientación.

Finalmente, podemos calcular la orientación de cada punto Harris. Para ello usamos la siguiente función:

```

1 def orientation(src, points):
2     for img, set_points in zip(src, points):
3         int_points = [(p.pt[0], p.pt[1]) for p in set_points]
4         i, j = [], []
5         for p in int_points:
6             i.append(int(np.round(p[1])))
7             j.append(int(np.round(p[0])))
8
9         dx, dy = derived_convolution(img=img, sigma=5, dx=1, dy=1, size=3, border=0,
10                                     normalize=False)
11         dx_g = cv2.GaussianBlur(src=dx, ksize=(6 * 5 + 1, 6 * 5 + 1), sigmaX=4.5,
12                                 borderType=cv2.BORDER_REFLECT)
13         dy_g = cv2.GaussianBlur(src=dy, ksize=(6 * 5 + 1, 6 * 5 + 1), sigmaX=4.5,
14                                 borderType=cv2.BORDER_REFLECT)
15
16         set_angles = ((np.arctan2(dy_g[i, j], dx_g[i, j])) * 180) / np.pi
17
18         for p, a in zip(set_points, set_angles):
19             p.angle = a
```

Esta función recibe la pirámide Gaussiana y los puntos (separados ya en los niveles correspondientes). Para calcular la orientación lo que hacemos es obtener las derivadas con respecto a x y con respecto a y, usando un valor de  $\sigma = 5$ . Luego alisamos con *GaussianBlur* y un valor de  $\sigma = 4,5$  ambas derivadas. Finalmente, calculamos el ángulo como

la arcotangente de la división del valor de la derivada alisada con respecto a y entre el valor de la derivada alisada con respecto a x. Esto lo hacemos de forma vectorizada en la línea 13. Finalmente asignamos los ángulos calculados a los correspondientes *KeyPoints*.

## 2.4. Apartado d. Descriptores.

Para extraer los descriptores usaremos la siguiente función:

```

1 def extract_descriptors(img, keypoints):
2     extractor = cv2.xfeatures2d.SIFT_create()
3     kp = [kp for set in keypoints for kp in set]
4     kp_news = [cv2.KeyPoint(x=k.pt[0] * 2 ** k.octave, y=k.pt[1] * 2 ** k.octave,
5                             _octave=k.octave, _size=k.size, _angle=k.angle, _response=k.response) for k in
6                             kp]
7     return extractor.compute(image=img, keypoints=kp_news)

```

La idea es simple, usaremos el descriptor SIFT junto con la función *compute*, la cual recibe una lista de *KeyPoints* y la imagen sobre la que obtener los descriptores. Los *KeyPoints* nos los llevamos a los valores de x e y correspondientes a la escala original (línea 4) antes de llamar a la función *compute*. El resto de campos los dejamos igual.

## 2.5. Resultado.

Una vez hemos desarrollado todas las funciones, podemos escribir una función que las llame de forma correcta y nos permita obtener el resultado final. La función en cuestión es:

```

1 def harris(img, block_size=7, ksize=7, windos_size=7, n_best=500, circle_color=(0, 0,
2     0), thickness=1):
3     pyramid = gaussian_pyramid(img=img, n_images=2, border=cv2.BORDER_REFLECT, collage=
4         False)
5     # Best points, no matter the level
6     points, harris_matrix = harris_points(src=pyramid, block_size=block_size, ksize=
7         ksize, windos_size=windos_size, n_best=n_best, border=cv2.BORDER_CONSTANT)
8
9     refined_points = refine_best_points(src=pyramid, points=points, harris_matrix=
10         harris_matrix)
11     orientation(src=pyramid, points=refined_points)
12
13     return refined_points, add_circles_lines(src=img, best_points=refined_points, color=
14         circle_color, thickness=thickness)

```

Esta función crea una pirámide Gaussiana (línea 2), luego obtiene los puntos Harris para todos los niveles de la pirámide (línea 4). A continuación, refina todos los puntos (línea 6) para calcular la orientación sobre los puntos refinados (línea 7). Finalmente, la función devuelve los puntos refinados (con su orientación correspondiente asignada y una imagen obtenida mediante la función *add\_circles\_lines*. Esta función es la siguiente:

```

1 def add_circles_lines(src, best_points, color=(0, 0, 0), thickness=1):
2     processed_image = copy.deepcopy(src)
3
4     for set_points in best_points:
5         for p in set_points:
6             cv2.circle(img=processed_image, center=(int(np.round(p.pt[0] * 2 ** p.
7                 octave)), int(np.round(p.pt[1] * 2 ** p.octave))), radius=int(np.round
8                 (10 / int(p.octave + 1))), color=color, thickness=thickness)
9

```

```

8     cv2.arrowedLine(img=processed_image , pt1=(int(np.round(p.pt[0] * 2 ** p.
9         octave)), int(np.round(p.pt[1] * 2 ** p.octave))), pt2=(int(np.round(p.
10        pt[0] * 2 ** p.octave)) + math.floor(np.sin(p.angle) * np.round(10 /
    int(p.octave + 1))), int(np.round(p.pt[1] * 2 ** p.octave)) + math.
        floor(np.cos(p.angle) * np.round(10 / int(p.octave + 1)))), color=color
    , thickness=thickness)
10

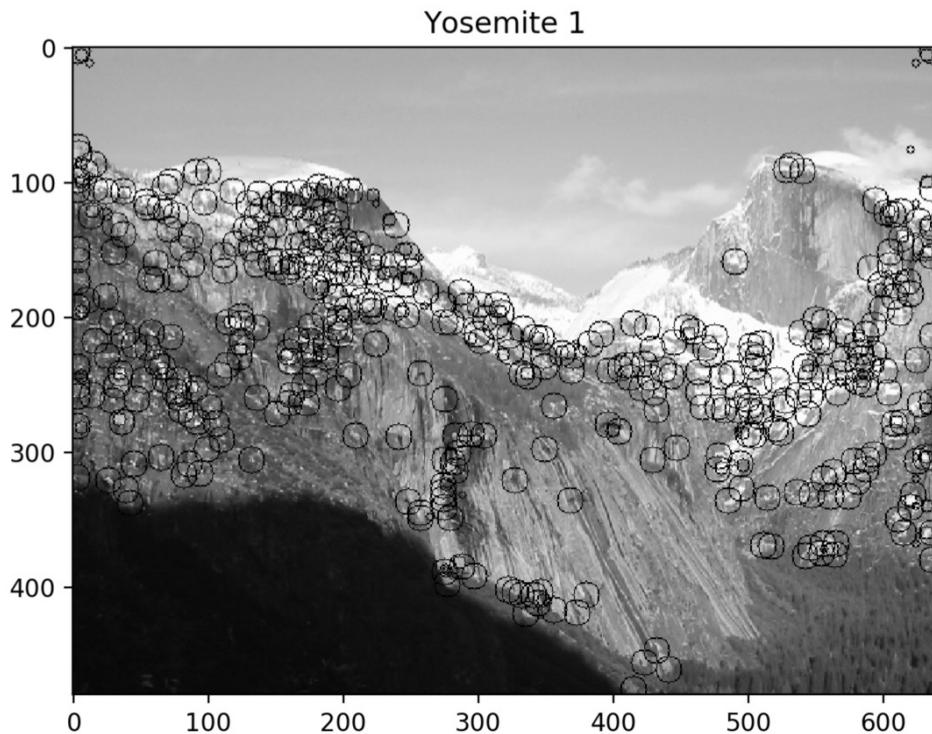
```

Esta función lo que hace es pintar los círculos mediante la función *circle*. Dichos círculos están centrados en el punto Harris en cuestión y tienen un radio proporcional al nivel de la pirámide donde han sido encontrados. A continuación pinta una línea mediante la función *arrowedLine* representando la orientación. Como origen del segmento se toma el centro del punto y como destino se toma un punto en su circunferencia. En concreto se toma el punto que venga indicado por la orientación. Es decir, aplicamos la siguiente fórmula:

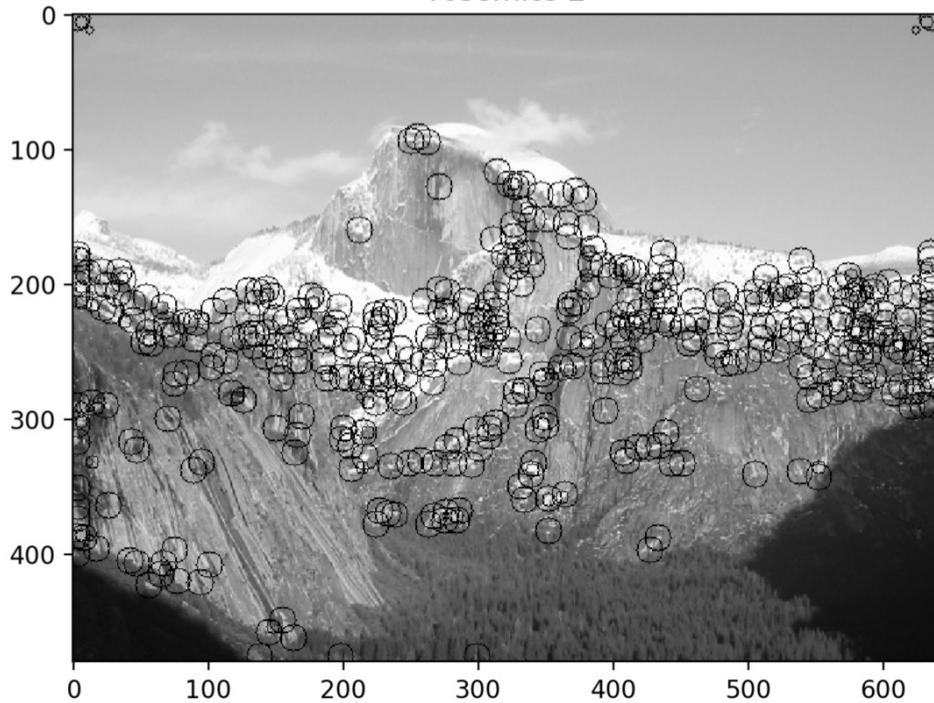
$$x_2 = x_1 + (distancia * \cos(\alpha)) \quad y_2 = y_1 + (distancia * \sin(\alpha))$$

Donde  $(x_1, y_1)$  es el punto de inicio,  $\alpha$  es el ángulo y *distancia*, en este caso, es el radio de nuestro círculo.

Primero vamos a ver el resultado del apartado a, es decir, obtener los puntos sin refinar y pintarlos sin la línea que representa el ángulo. El resultado es el siguiente:

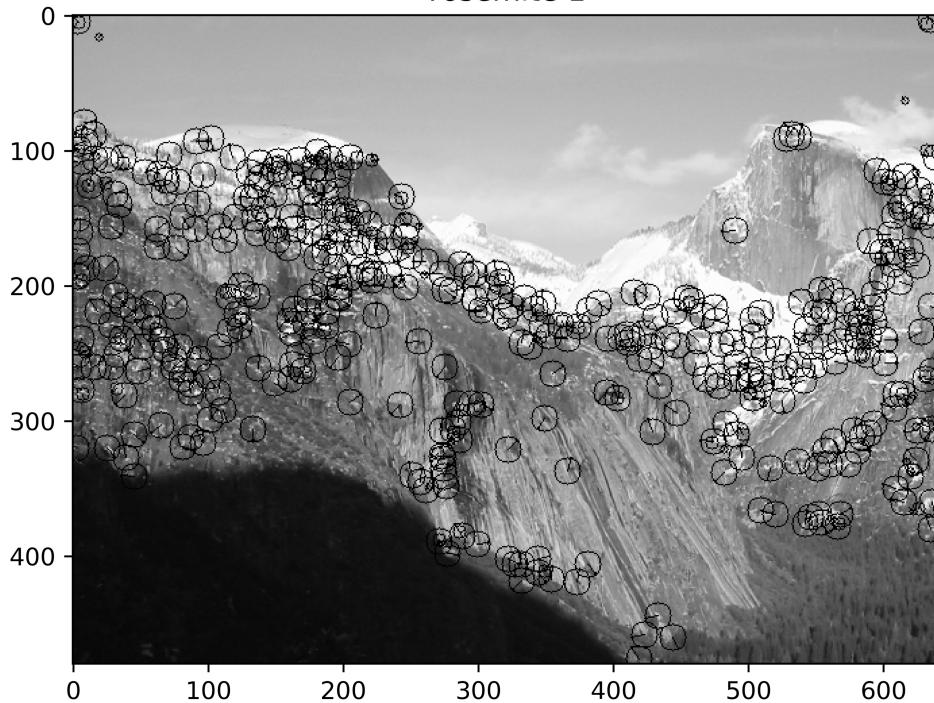


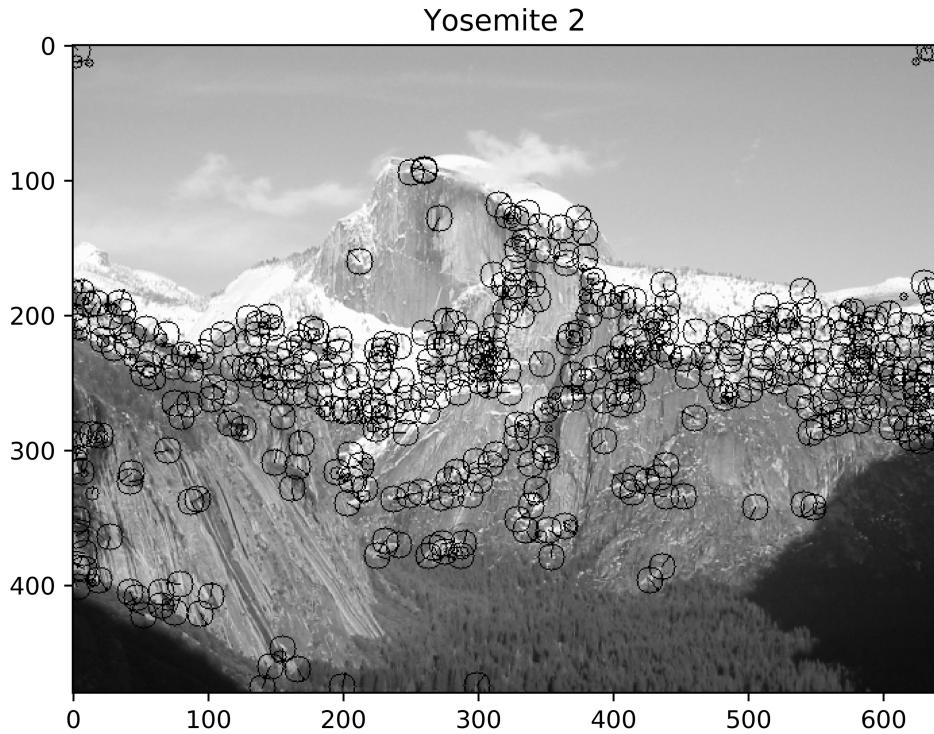
Yosemite 2



Una vez con todo listo, vamos a ver el resultado obtenido para las dos imágenes de Yosemite, ya con los puntos refinados y con los ángulos calculados y representados:

Yosemite 1





Como podemos ver, no nos aparecen puntos Harris en sitios raros, como podría ser la zona correspondiente a la mancha negra de la parte inferior de la esquina ni en el cielo. También podemos ver que aparecen en zonas interesantes, como pueden ser las siluetas de las montañas.

### 3. Ejercicio 2.

La función desarrollada es la siguiente:

```

1 def sift_descriptor(img1, img2, norm_type=cv2.NORM_L2, n_matches=50, bruteForce=True,
2     threshold=0.7, keypoints=None):
3     if keypoints is None:
4         descriptor = cv2.xfeatures2d.SIFT_create()
5         kp1, desc1 = descriptor.detectAndCompute(image=img1, mask=None)
6         kp2, desc2 = descriptor.detectAndCompute(image=img2, mask=None)
7     else:
8         kp1, desc1 = extract_descriptors(img=img1, keypoints=keypoints[0])
9         kp2, desc2 = extract_descriptors(img=img2, keypoints=keypoints[1])
10
11     if bruteForce:
12         bf_matcher = cv2.BFMatcher(normType=norm_type, crossCheck=True)
13         matches = bf_matcher.match(desc1, desc2)
14     else:
15         bf_matcher = cv2.BFMatcher(normType=norm_type, crossCheck=False)
16         all_matches = bf_matcher.knnMatch(desc1, desc2, 2)
17         matches = [m[0] for m in all_matches if m[0].distance < m[1].distance *
18             threshold]
19
20     output = np.zeros(shape=(1, 1)).astype(np.float64)
21     full = cv2.drawMatches(img1=img1, keypoints1=kp1, img2=img2, keypoints2=kp2,
22                           matches1to2=matches, outImg=output)

```

```

21     if n_matches < len(matches):
22         partial_matches = np.array(sorted(matches, key=lambda x: x.distance, reverse=
23                                         False))[0:n_matches]
23         partial = cv2.drawMatches(img1=img1, keypoints1=kp1, img2=img2, keypoints2=kp2,
24                                   matches1to2=partial_matches, outImg=output)
24         return full, partial, partial_matches
25     else:
26         return full, full, matches

```

Dicha función obtiene dos imágenes para calcular correspondencias entre ellas, el número de matches a dibujar, un flag que indica si queremos usar el vecino más cercano o usar un método de fuerza bruta y un último argumento, *keypoints*, el cual si vale *None* se calcularán los *keypoints* usando *OpenCV* (líneas 2 a 5). En caso contrario, se pasarán los *keypoints* a usar (los calculados mediante las funciones del ejercicio anterior) (líneas 6 a 8). A continuación vemos si queremos usar *BruteForce+CrossCheck* o queremos usar *Lowe-Average-2NN*. Para el primero de ellos simplemente debemos usar un objeto de la clase *BFMatcher* con el flag *crossCheck=True*. Para el segundo vamos a usar la función *knnMatch* indicando que queremos que nos devuelva 2 correspondencias para cada descriptor. La idea es comprobar la distancia entre el punto denominado *query* y los dos dados por la correspondencia asociada y ver si hay una distancia mayor que un umbral. Este umbral se ha fijado a 0.7. Este proceso es el que se lleva a cabo en la línea 16. Una vez tenemos los matches, si los puntos son los calculados por *OpenCV*, usamos la función *drawMatches* para representarlos.

Veamos los resultados obtenidos:

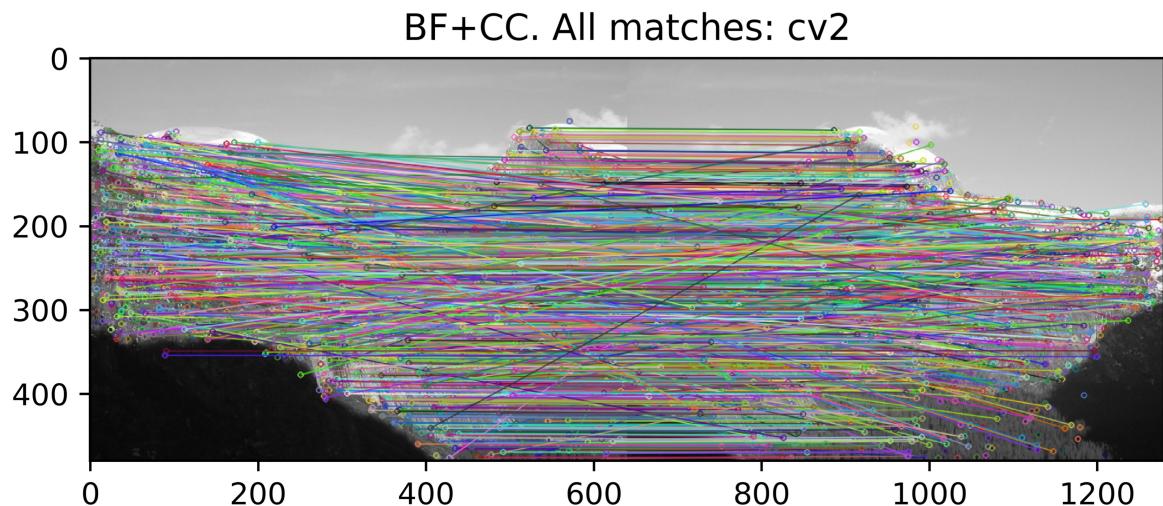


Figura 3.1: Todos los matches. BruteForce+CrossCheck. KeyPoints de OpenCV.

kNN. All matches: cv2

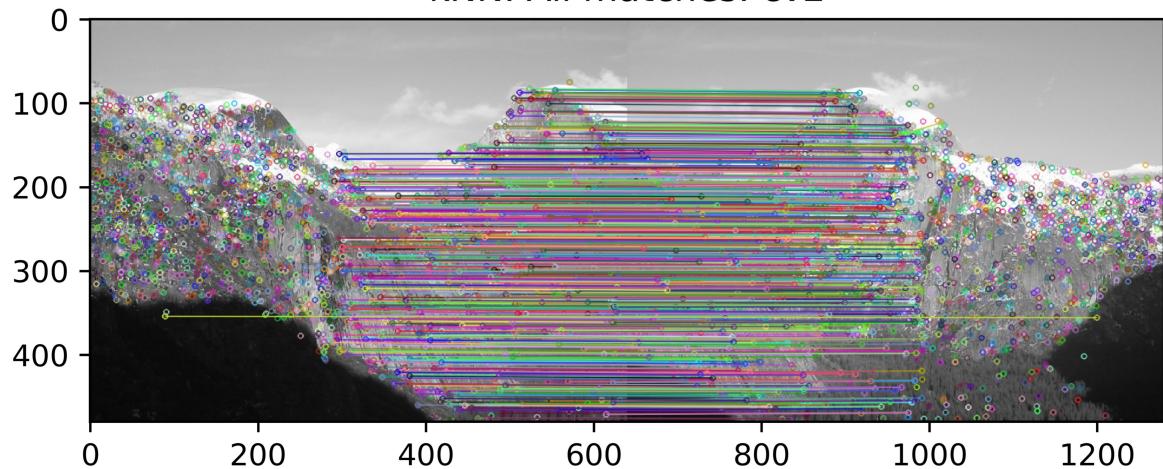


Figura 3.2: Todos los matches. Lowe-Average-2NN. KeyPoints de OpenCV.

BF+CC. All matches

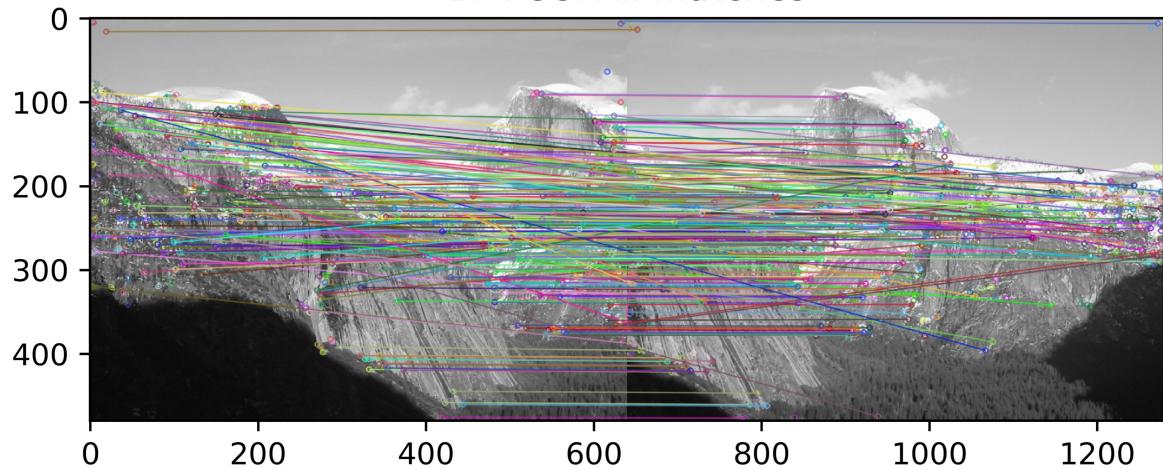


Figura 3.3: Todos los matches. BruteForce+CrossCheck. KeyPoints propios.

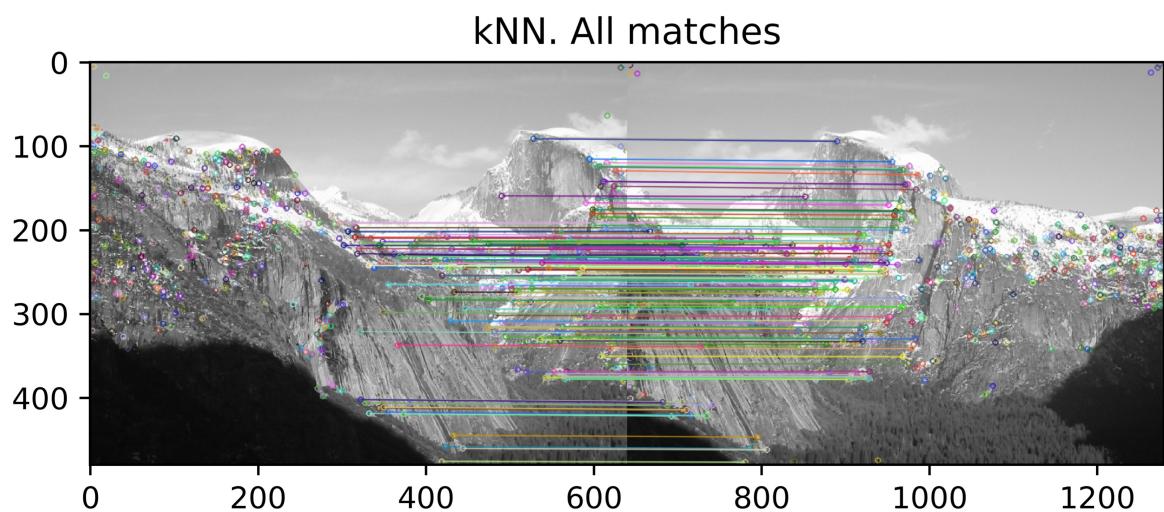


Figura 3.4: Todos los matches. Lowe-Average-2NN. KeyPoints propios.

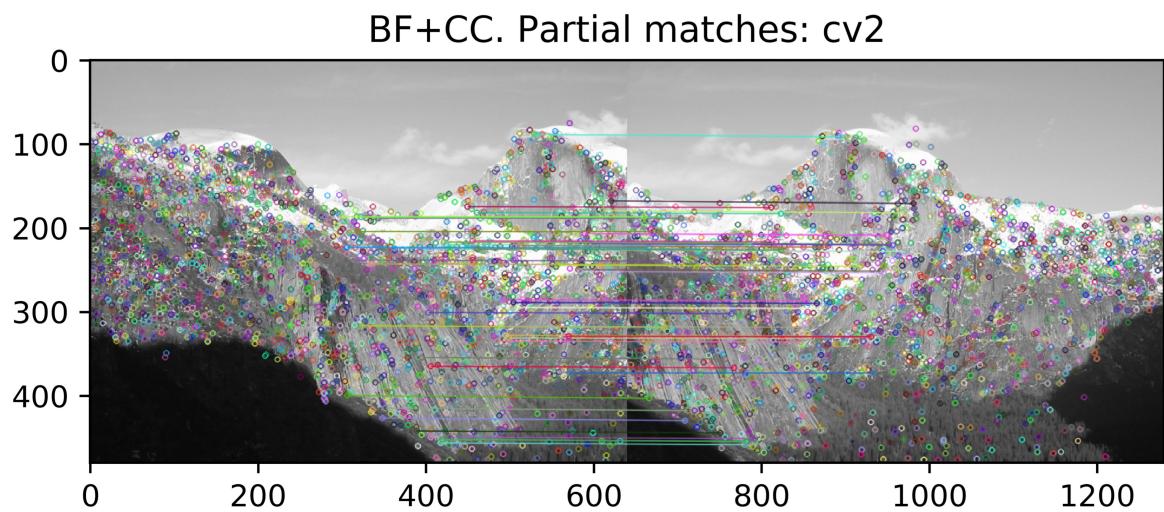


Figura 3.5: Los 50 mejores matches. BruteForce+CrossCheck. KeyPoints de OpenCV.

kNN. Partial matches: cv2

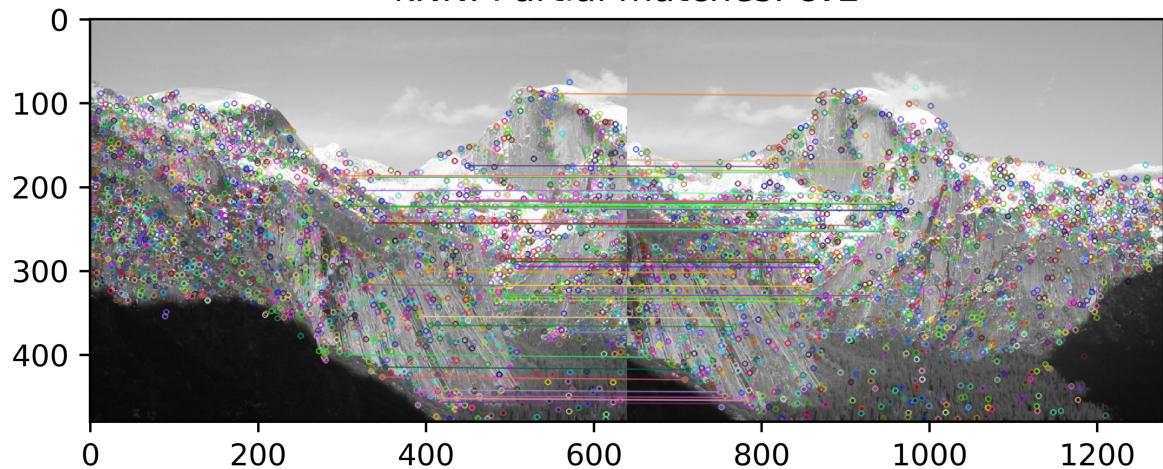


Figura 3.6: Los 50 mejores matches. Lowe-Average-2NN. KeyPoints de OpenCV.

BF+CC. Partial matches

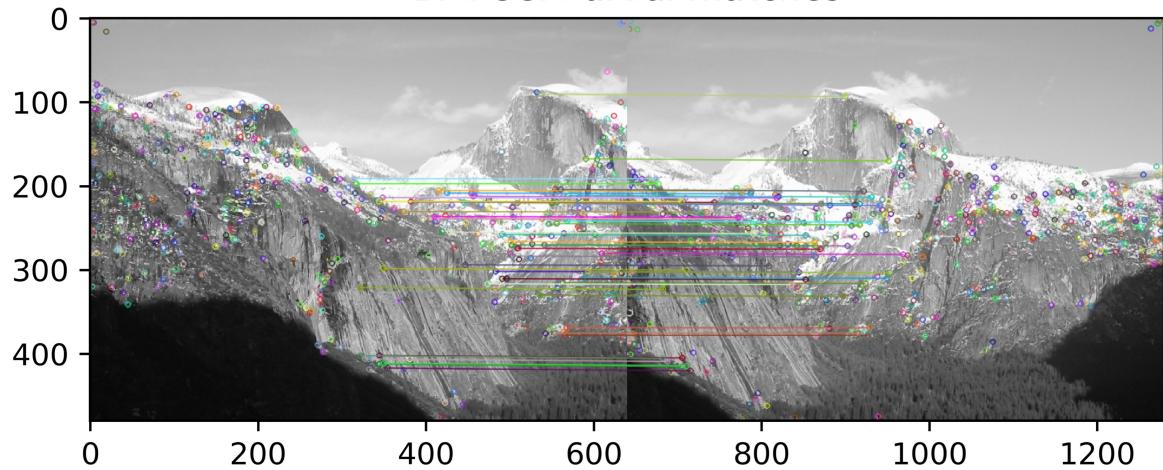


Figura 3.7: Los 50 mejores matches. BruteForce+CrossCheck. KeyPoints propios.

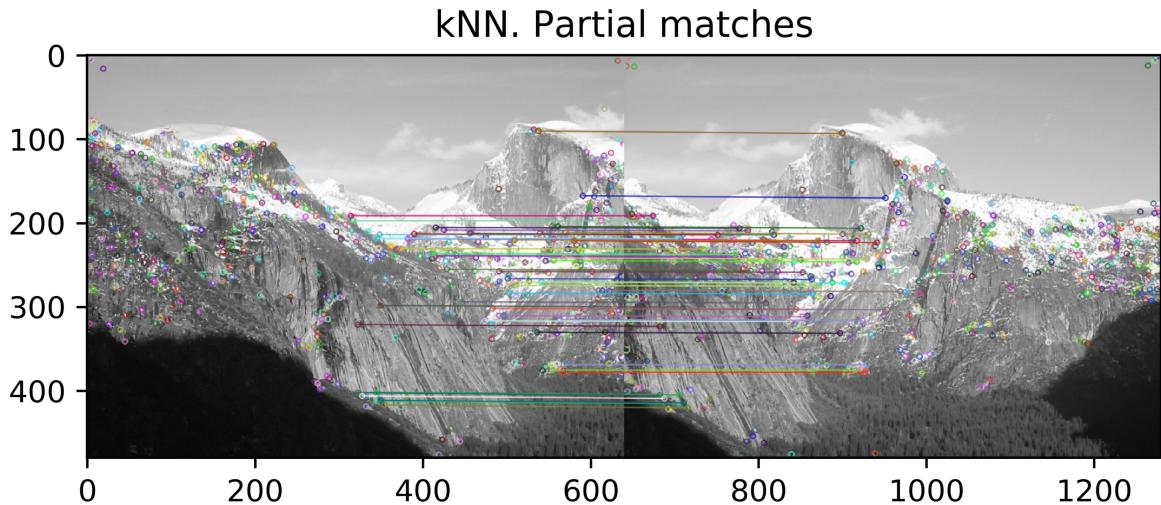


Figura 3.8: Los 50 mejores matches. Lowe-Average-2NN. KeyPoints propios.

Como podemos ver, el resultado entre usar mis KeyPoints y los de OpenCV son bastante similares. De hecho, en ambos casos obtenemos unos buenos matches. Voy a mostrar unas tablas para comparar los resultados obtenidos:

OpenCV	numero de matches	match[912].distance
BF+CC	1535	209.671173
kNN	912	242.604614

KeyPoints propios	numero de matches	match[174].distance
BF+CC	297	181.085617
kNN	174	241.921478

OpenCV	numero de matches	match[50].distance
BF+CC	50	28.460498
kNN	50	28.460498

KeyPoints propios	numero de matches	match[50].distance
BF+CC	50	32.526912
kNN	50	31.780496

La idea es comparar el atributo *distance* de los matches. Sabemos que, a menor distancia, mejor es el match encontrado. Para ser justos, vamos a comparar siempre el último match (ya que están ordenados de mejor a peor) pero, si un método encuentra más matches que otro, compararemos los matches que están en el índice de la longitud del mínimo número de matches. Es decir, en el caso de usar los *KeyPoints* de *OpenCV*,

*BF+CC* nos encuentra 1535 matches mientras que *kNN* 912. Por lo tanto compararemos la distancia del match 912 en ambos casos. Como podemos ver la distancia es menor en el caso de *BF+CC*, por lo tanto podemos decir que obtenemos un mejor resultado con el.

En el caso de usar mis propios *KeyPoints*, *BF+CC* nos encuentra 297 matches mientras que *kNN* 174. Por lo tanto compararemos la distancia del match 174 en ambos casos. Como podemos ver la distancia es menor en el caso de *BF+CC*, por lo tanto podemos decir que obtenemos un mejor resultado con el.

Si nos fijamos en los 50 mejores matches y seguimos la misma filosofía, en el caso de usar los *KeyPoints* de *OpenCV* obtenemos un resultado igual pero si usamos los míos propios, obtenemos un mejor resultado con *kNN*.

## 4. Ejercicios 3 y 4. Construcción de un mosaico.

He fusionado los dos ejercicios puesto que he hecho una función genérica que no le importa el número de imágenes.

Primero he desarrollado una función que calcula una homografía entre dos imágenes:

```

1 def get_homography(img1, img2, norm_type=cv2.NORM_L2, threshold=0.7):
2     sift = cv2.xfeatures2d.SIFT_create()
3
4     kp1, desc1 = sift.detectAndCompute(img1, None)
5     kp2, desc2 = sift.detectAndCompute(img2, None)
6
7     bf_matcher = cv2.BFMatcher(normType=norm_type, crossCheck=False)
8     all_matches = bf_matcher.knnMatch(desc1, desc2, 2)
9     matches = [m[0] for m in all_matches if m[0].distance < m[1].distance * threshold]
10
11    sorted_matches = sorted(matches, key=lambda x: x.distance, reverse=False)
12
13    src_points = np.float32([kp1[p.queryIdx].pt for p in sorted_matches])
14    dst_points = np.float32([kp2[p.trainIdx].pt for p in sorted_matches])
15
16    return cv2.findHomography(src_points, dst_points, cv2.RANSAC, 1)
```

Esta función calcula los *KeyPoints* y los descriptores como en el ejercicio anterior. Una vez tenemos los *KeyPoints*, simplemente llamamos a la función *cv2.findHomography* con la imagen 1 como fuente y la imagen dos como destino. Debemos tener en cuenta que la homografía va de un plano a otro, en este caso de *img1* a *img2*. Una vez tenemos el cálculo de la homografía, podemos desarrollar la función que monta el mosaico:

```

1 def mosaic(src):
2     if len(src) % 2 == 0:
3         src.append(src[-1])
4
5     height = int(np.sum([np.shape(img)[0] for img in src]))
6     width = int(np.sum([np.shape(img)[1] for img in src]))
7     result = np.ones(shape=(height, width)) * 255
8     center = len(src) // 2
9     shape_center = np.shape(src[center])
10    shape_mosaic = np.shape(result)
11
12    homographies = [get_homography(src[i], src[i + 1])[0] for i in range(len(src) - 1)]
```

```

13
14     for i in range(center, len(homographies)):
15         homographies[i] = np.linalg.inv(homographies[i])
16
17     h_0 = [[1, 0, int(np.round(shape_mosaic[1] / 2)) - int(np.round(shape_center[1] / 2))], [0, 1, int(np.round(shape_mosaic[0] / 2)) - int(np.round(shape_center[0] / 2))], [0, 0, 1]]
18
19     left, right = np.split(np.array(homographies), 2)
20     left = list(reversed(left))
21     left = list(itertools.chain(np.array([h_0]), left))
22     right = list(itertools.chain(np.array([h_0]), right))
23
24     for i in range(1, len(left)):
25         left[i] = np.dot(left[i - 1], left[i])
26         right[i] = np.dot(right[i - 1], right[i])
27
28     left = list(reversed(left))
29     del left[-1]
30
31     final_homographies = list(itertools.chain(left, right))
32
33     for i in range(len(src)):
34         result = cv2.warpPerspective(src=src[i], M=np.array(final_homographies[i]),
35                                       dtype=np.float32, dst=result, dsize=(np.shape(result)[1], np.shape(result)[0]),
36                                       borderMode=cv2.BORDER_TRANSPARENT)
37
38     r, g, b = cv2.split(result)
39     r = np.transpose(r)
40     g = np.transpose(g)
41     b = np.transpose(b)
42     transposed = cv2.merge([r, g, b])
43
44     rows = [i for i, row in zip(range(np.shape(result)[0]), result) if not np.all(row == [0, 0, 0])]
45     cols = [i for i, col in zip(range(np.shape(result)[1]), transposed) if not np.all(col == [0, 0, 0])]

    return result[rows][:, cols]

```

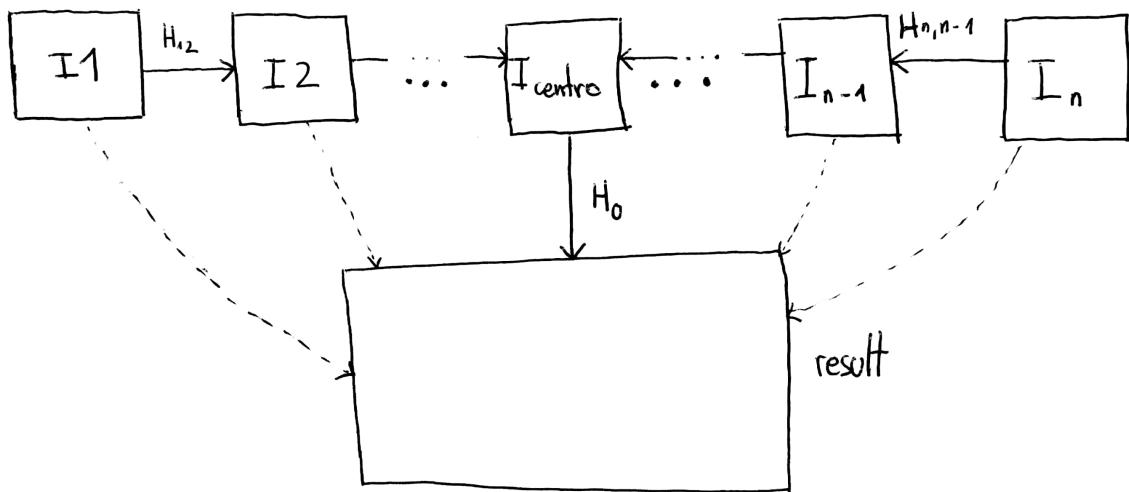
Lo primero que comprueba nuestra función es ver si el número de imágenes recibidas en *src* es impar. Si no lo es, duplica la última imagen. Esto se hace simplemente para poder trabajar bien con el concepto de imagen central. A continuación creamos nuestro fondo negro donde vamos a ir colocando el mosaico, *result*. Luego calculamos todas las homografías necesarias, es decir, la homografía que va de la imagen *i* a la imagen *i + 1*. Una vez tenemos dichas imágenes, tal y como hemos comentado antes, esas homografías van hacia un sentido. Imaginemos que vamos a colocar N imágenes en el mosaico, pues tendremos una imagen central y  $(N - 1)/2$  imágenes a cada lado. Para la parte izquierda si nos interesa la homografía que va de la imagen *i* a la imagen *i + 1* pero para la parte derecha nos interesa la homografía que va de la imagen *i+1* a la imagen *i*. Esta homografía no es más que la inversa de la que lleva la imagen *i* a la imagen *i+1*. Este proceso de inversión lo podemos ver en las líneas 14 y 15.

Una vez tenemos las homografías que nos relacionan las imágenes, sólo necesitamos  $H_0$ , es decir, la homografía central al mosaico. Al ser la imagen central, suponemos que no sufre ningún tipo de deformación y por lo tanto sólo nos interesa una homografía que haga la operación de traslación. Como bien sabemos de teoría, estas homografías tienen la siguiente forma:

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

donde  $x$  e  $y$  representan los origenes de coordenadas que se van a trasladar. En nuestro caso nos interesa que el centro de la imagen central coincida con el centro de nuestro mosaico, entonces los valores que debemos tomar de  $x$  e  $y$  son  $(n\_cols\_mosaico/2) - (n\_cols\_imgCentral/2)$  para  $x$  y  $(n\_filas\_mosaico/2) - (n\_filas\_imgCentral/2)$  para  $y$ . De esta manera conseguimos que los centros queden alineados (línea 17).

Teniendo las homografías que relacionan las imágenes y  $H_0$  no necesitamos mucho más. Lo que tenemos es lo siguiente:

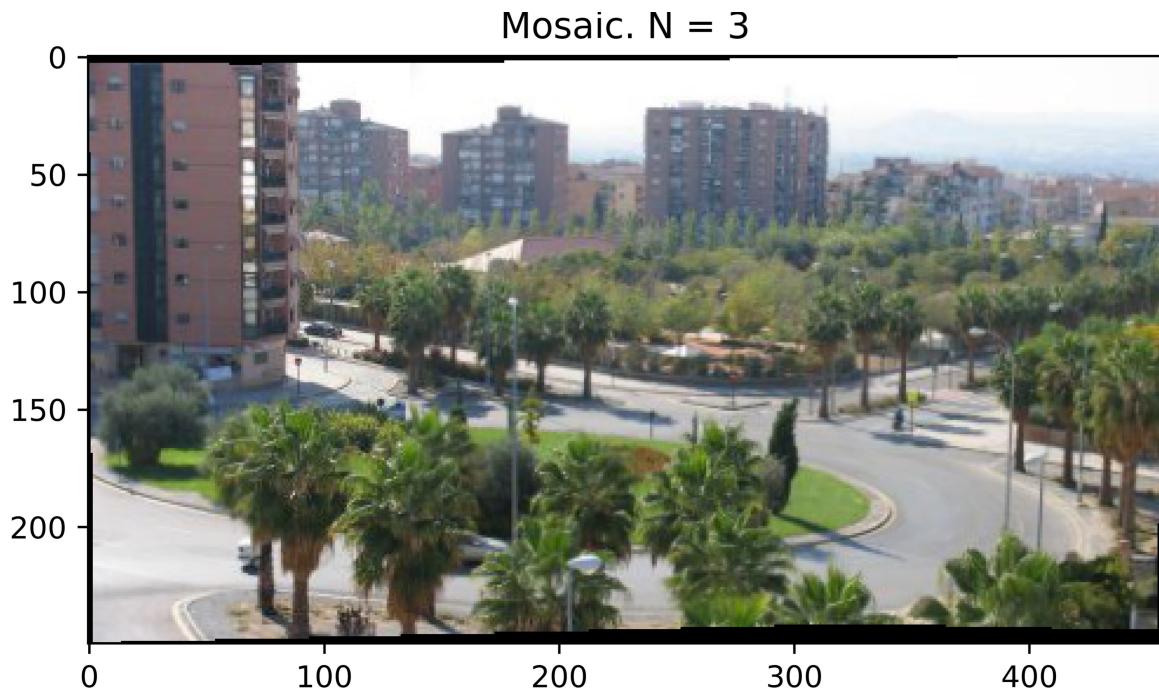


Lo único que debemos calcular son las homografías que nos llevan cada imagen al mosaico (representadas por líneas discontinuas en la imagen superior). Estas homografías lo que deben hacer es, por ejemplo, para la imagen 1, debe llevar cada punto de la imagen 1 al plano de la imagen 2, luego al plano de la imagen 3, y así sucesivamente hasta llegar al plano de la imagen del centro. Una vez estamos ahí, con  $H_0$  podemos llevar dicho punto hasta el mosaico. Es decir, debemos encontrar, para cada imagen, una homografía equivalente a aplicar varias homografías hasta llegar a la imagen central y luego aplicar  $H_0$ . Dicha homografía la podemos obtener multiplicando las homografías involucradas. Es decir, la homografía que va de la imagen  $i$  al mosaico es:  $H = (\dots((H_0 * H_1) * H_2) * \dots * H_i)$  Este proceso de construcción de las homografías se puede ver desde la línea 19 a la 31. Primero sepáramos las homografías correspondientes en dos mitades, la asociada a la parte izquierda y la asociada a la parte derecha del mosaico. Invertimos la parte izquierda (para tener las homografías en orden de como cerca están las imágenes correspondientes de la imagen central). A ambas partes le añadimos  $H_0$  para calcular las homografías en un bucle, aplicando la idea comentada anteriormente, calcular la homografía  $H_i$  en

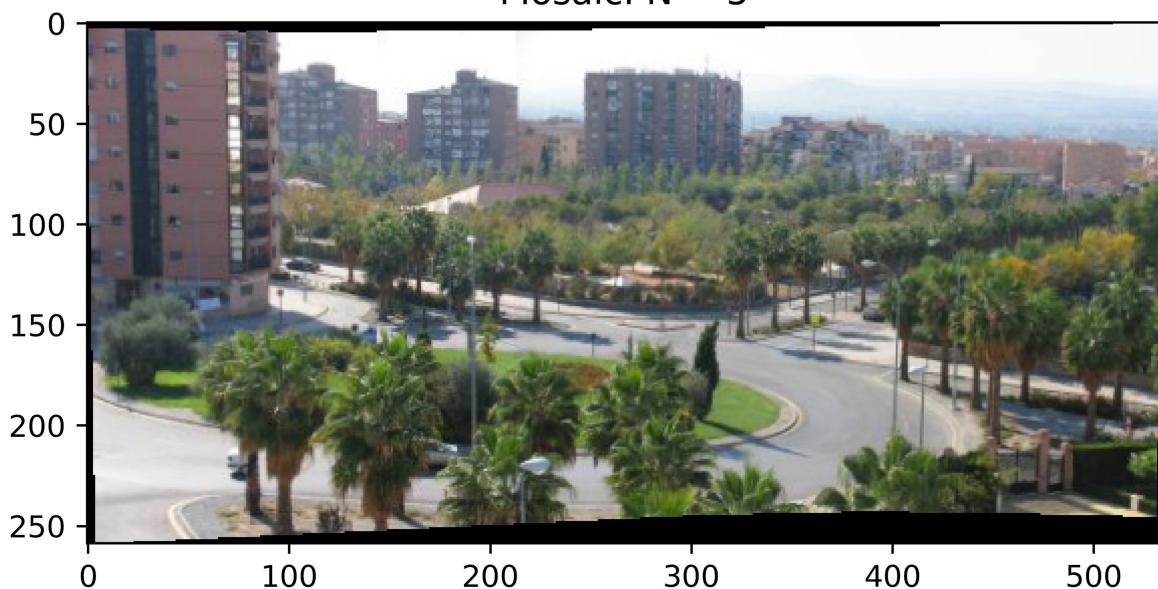
función de la homografía  $H_{i-1}$ , la cual es un cúmulo de las homografías anteriores y  $H_0$ . Finalmente, invertimos de nuevo el orden a las homografías de la parte izquierda, y quitamos  $H_0$  (para no tenerla repetida) y juntamos ambas partes (líneas 28 a 31).

Una vez tenemos las homografías, sólo debemos hacer una llamada a *warpPerspective* para cada imagen. En dicha llamada interviene la imagen que deseamos llevar al mosaico, *src[i]*, el mosaico como destino y la homografía correspondiente calculada según el proceso comentado. Para obtener unos buenos resultados, debemos usar el flag *cv2.BORDER\_TRANSPARENT* e indicar que el tamaño del resultado debe ser igual que el tamaño del mosaico donde vamos construyendo nuestro panorama. El resultado de las llamadas a esta función lo guardamos en el propio mosaico, para ir montándolo de forma incremental.

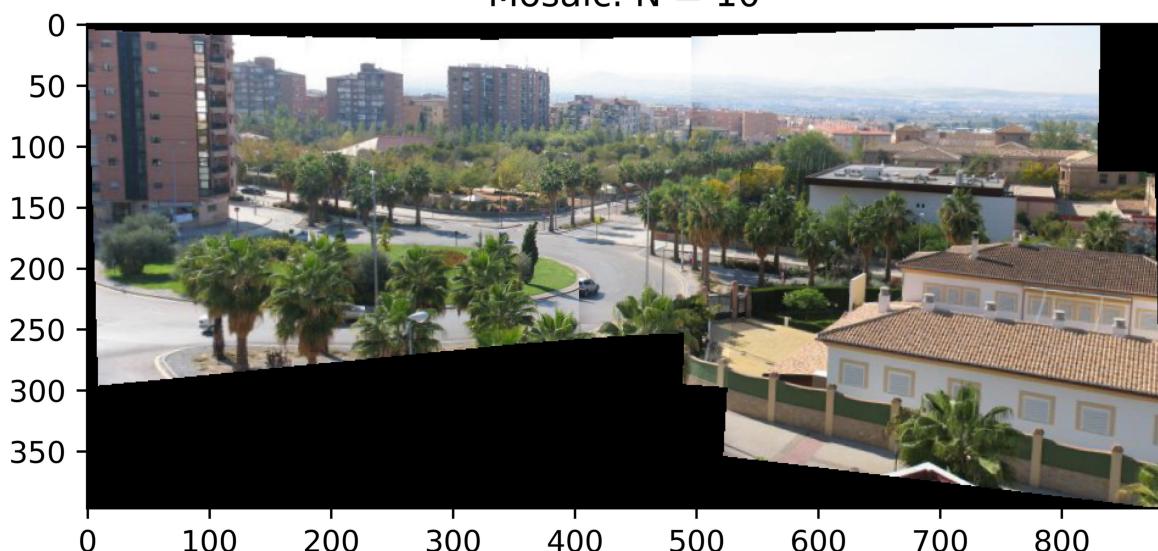
Finalmente, por estética, borramos las filas y columnas cuyo contenido son todo ceros, es decir, son en negro. Debemos tener cuidado con los tres canales a la hora de trasponer la imagen para procesar las columnas, por eso los sepáramos, los trasponemos y luego juntamos (líneas 36 a 40). Una vez sabemos que filas y columnas no debemos borrar (líneas 42 y 43) sólo debemos devolver la imagen con dichas filas y columnas (línea 44). Los resultados obtenidos son los siguientes. Primero para 3 imágenes (ejercicio 3), luego para 5 imágenes y finalmente para 10 imágenes.



Mosaic. N = 5



Mosaic. N = 10



Como podemos apreciar el resultado es bastante bonito y visual. Además, podemos ver que el proceso de borrar las filas y columnas “innecesarias” nos aporta un mejor resultado visual.

He querido probar como funciona mi función y la he probado sobre un mosaico propio.

El resultado es bastante bueno, pero por temas de luminosidad no queda del todo bien.  
Aún así, creo relevante mostrarlos aquí.<sup>1</sup>

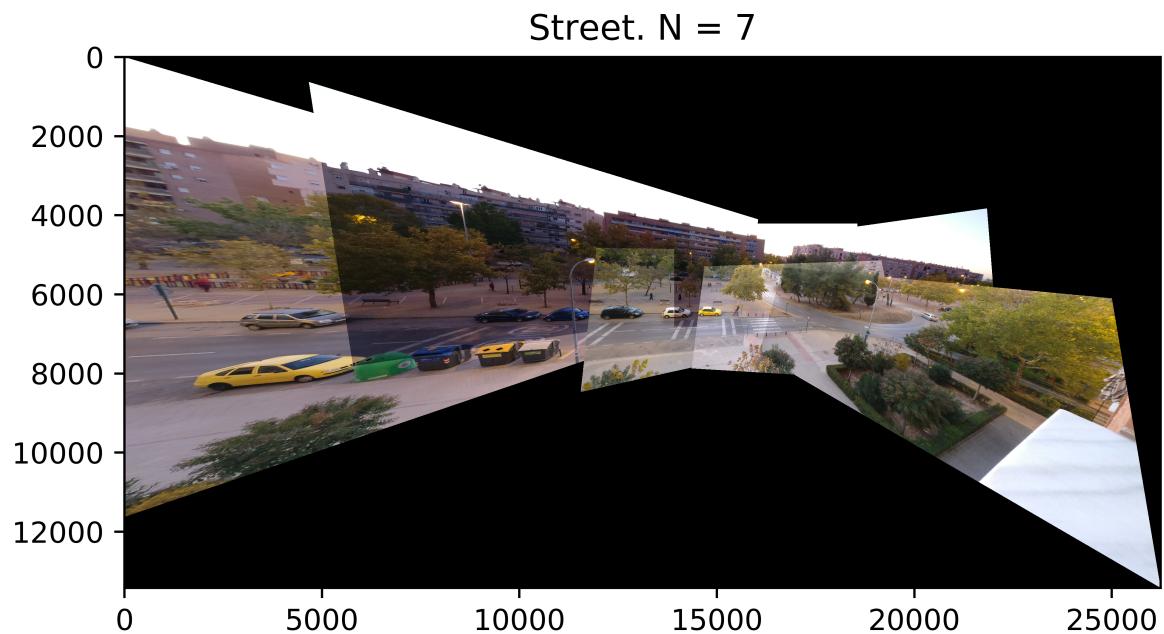


Figura 4.1: Imágenes de la carpeta */data/street*

---

<sup>1</sup>Las imágenes originales se encuentran dentro de la carpeta data que he entregado.