



UNIVERSIDAD DE GRANADA

VISIÓN POR COMPUTADOR
GRADO EN INGENIERÍA INFORMÁTICA

PRÁCTICA 3

DETECCIÓN DE PUNTOS RELEVANTES Y CONSTRUCCIÓN DE PANORAMAS

Autor

Vladislav Nikolov Vasilev

Rama

Computación y Sistemas Inteligentes



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE
TELECOMUNICACIÓN

CURSO 2019-2020

Índice

1. DETECCIÓN DE PUNTOS HARRIS	2
1.1. Estimación de los <i>keypoints</i> utilizando Harris	2
1.2. Experimentación con los parámetros	14
1.3. Comparación de la estimación con los puntos refinados	24
2. EXTRACCIÓN DE DESCRIPTORES AKAZE	29
2.1. Obtención de los descriptores AKAZE	29
2.2. <i>Matching</i> mediante BruteForce y Cross Check	30
2.3. <i>Matching</i> mediante Lowe-Average-2NN	31
2.4. Comparativa	33
3. MOSAICO DE DOS IMÁGENES	38
4. MOSAICO DE N IMÁGENES	38
Referencias	39

1. DETECCIÓN DE PUNTOS HARRIS

En esta primera sección vamos a estudiar el comportamiento del detector de puntos Harris, y cómo podemos utilizarlo para extraer *keypoints* de la imagen. Para ello, vamos a implementar algunas funciones que simulen el comportamiento de las de OpenCV. Además, vamos a experimentar con los parámetros que reciben dichas funciones para ver cómo se ve afectada la salida. Finalmente, vamos a refinar un poco los puntos que hemos obtenido, para ver cómo de buenas son las estimaciones que hemos hecho.

1.1. Estimación de los *keypoints* utilizando Harris

Nuestro principal objetivo es, dada una imagen en escala de grises, obtener los *keypoints*, los cuales pueden ser utilizados para otras muchas cosas, como por ejemplo para el *matching* entre imágenes. Existen muchos operadores que podemos utilizar para extraer los *keypoints*, pero nosotros aquí vamos a utilizar el operador de Harris, y lo combinaremos con algunas técnicas más. Es importante recalcar la parte de **imagen en escala de grises**, ya que Harris solo funciona con imágenes de este tipo.

Harris se utiliza principalmente para extraer las esquinas de una imagen. Aprovechando dicha información, podemos sacar información relevante que sea representativa de la imagen que tenemos a diversas escalas, de forma que en un principio nos fijamos en qué se encuentra en las escalas más bajas, y por tanto en detalles más destacables a simple vista, y luego vamos subiendo, hasta ver detalles algo más ocultos o pequeños.

Para extraer información interesante utilizando Harris, podemos seguir el siguiente esquema:

1. Extraer puntos de Harris de la imagen.
2. Aplicar un umbral a los puntos obtenidos anteriormente, eliminando aquellos que no lo superen.
3. Aplicar supresión de no máximos, quedándonos solo con el máximo local.
4. Utilizar los puntos restantes como *keypoints*, obteniendo información sobre sus posiciones, la escala y su orientación.

Este proceso se ha extraído del paper de Matthew Brown [1], y si se quiere tener información más en detalle, se recomienda consultarla. Nosotros iremos mucho más al grano, y explicaremos lo esencial.

Lo primero que necesitamos es tener la pirámide Gaussiana de la imagen, ya que así tenemos distintas escalas de la imagen. También necesitamos las pirámides de las derivadas, ya que las utilizaremos para calcular las orientaciones de los píxeles en cada una de las escalas. De esta forma, tenemos todas las escalas de la imagen y de las derivadas asociadas a dichas escalas calculadas desde un principio, con lo cuál no necesitamos ningún cálculo adicional en el proceso.

Para obtener la pirámide la imagen original, nos hemos ayudado de la siguiente función:

```

1 def compute_gaussian_pyramid(img, n_octaves):
2     """
3         Funcion que permite calcular una piramide Gaussiana de n_octaves
4             escalas
5
6     Args:
7         img: Imagen de la que extraer la piramide
8         n_octaves: Numero de octavas que tiene que tener la piramide
9
10    Return:
11        Devuleve la piramide Gaussiana de la imagen de entrada
12    """
13
14    # Crear lista que contendra la piramide Gaussiana
15    # Inicialmente contiene la imagen de entrada (el nivel mas bajo)
16    gauss_pyr = [img]
17
18    # Obtener piramide
19    for i in range(1, n_octaves):
20        gauss_pyr.append(cv2.pyrDown(gauss_pyr[i-1]))
21
22    return gauss_pyr

```

Para obtener la pirámide, nos hemos ayudado de la función `pyrDown` de OpenCV. Esta función lo que hace es devolver el siguiente nivel de la pirámide, encargándose de aplicar el alisamiento Gaussiano correspondiente y de reducir el tamaño de la imagen en el proceso a un cuarto de la original (se reduce la mitad en el eje de las X y la mitad en el eje de las Y).

Para extraer las pirámides de las derivadas, hemos utilizado la siguiente función:

```

1 def compute_derivative_pyramids(img, ksize_der, n_octaves, sigma
2     =4.5):
3     """
4         Funcion que calcula las piramides Gaussianas de las derivadas en
5             los
6             ejes X e Y dada una imagen de entrada. La imagen de entrada ese
7             alisada
8             inicialmente con un filtro Gaussiano de sigma 4.5
9
10    Args:
11        img: Imagen de la que extraer las piramides de las derivadas
12        ksize_der: Tamaño del kernel de la derivada

```

```

10     n_octaves: Numero de imagenes que compondran las piramides
11     sigma: Sigma del alisamiento Gaussiano (default: 4.5)
12     Return:
13         Devuelve dos listas, una para la piramide de las derivadas
14         en el eje
15             X y otra para la piramide de las derivadas en el eje Y
16             """
17     # Aplicar alisamiento Gaussiano
18     smooth = gaussian_kernel(img, int(3*sigma) * 2 + 1, sigma)
19
20     # Calcular derivadas
21     dx = derivative_kernel(smooth, ksize_der, 1, 0)
22     dy = derivative_kernel(smooth, ksize_der, 0, 1)
23
24     # Añadir derivadas a las correspondientes listas
25     dx_pyr = [dx]
26     dy_pyr = [dy]
27
28     # Crear piramide
29     for i in range(1, n_octaves):
30         dx_pyr.append(cv2.pyrDown(dx_pyr[i-1]))
31         dy_pyr.append(cv2.pyrDown(dy_pyr[i-1]))
32
33     return dx_pyr, dy_pyr

```

Se puede ver que en general no hay mucho misterio en el proceso, ya que es bastante parecido a lo que se pudo ver en la función anterior. Al principio se hace un alisamiento Gaussiano con $\sigma = 4.5$, tal y como se indica en el *paper*. El tamaño del *kernel* es proporcional a σ , de forma que se está discretizando en el rango $[-3\sigma, 3\sigma]$. Se puede ver como dicho tamaño se incrementa en uno en la función, con el objetivo de que sea impar. Al aplicar este alisamiento lo que conseguimos es eliminar las frecuencias altas de la imagen, eliminando ruido. Después calculamos las derivadas con las funciones de la práctica anterior, aplicando por debajo el operador de *Sobel* con una apertura de *ksize* (un *kernel* de dicho tamaño, dicho de otra forma). Finalmente, sacamos las pirámides utilizando de nuevo *pyrDown*.

Vamos a ver ahora cómo se calculan ahora los puntos de Harris. Recordemos que los puntos se calculan para cada píxel de la imagen original, valiéndose para ello de la descomposición en valores singulares de una matriz determinada. Esta descomposición nos permite obtener dos valores singulares: λ_1 y λ_2 . Lo importante de aquí es saber que estos valores ofrecen información sobre cómo de rápido cambian los valores de la intensidad de la imagen en una ventana de un tamaño determinado en los ejes *X* e *Y* respectivamente. Además, nos permiten deducir una serie de cosas:

- Si λ_1 y λ_2 son pequeños, entonces no se produce un cambio de intensidad en la región. Por tanto, se podría ignorar dicha información, ya que no aporta información relevante.

- Si uno es más grande que el otro, significa que hay un borde en el eje del mayor valor singular, ya que la variación de intensidad es mayor en un eje que en el otro.
- Si los dos son grandes, significa que hay una variación de la intensidad grande en ambos ejes, y por tanto, que estamos en una esquina.

Lo dicho anteriormente se puede ver también en la siguiente figura, donde además se muestra qué tipo de elipse formarían los valores singulares (ya que también se puede hacer una interpretación geométrica de lo dicho anteriormente):

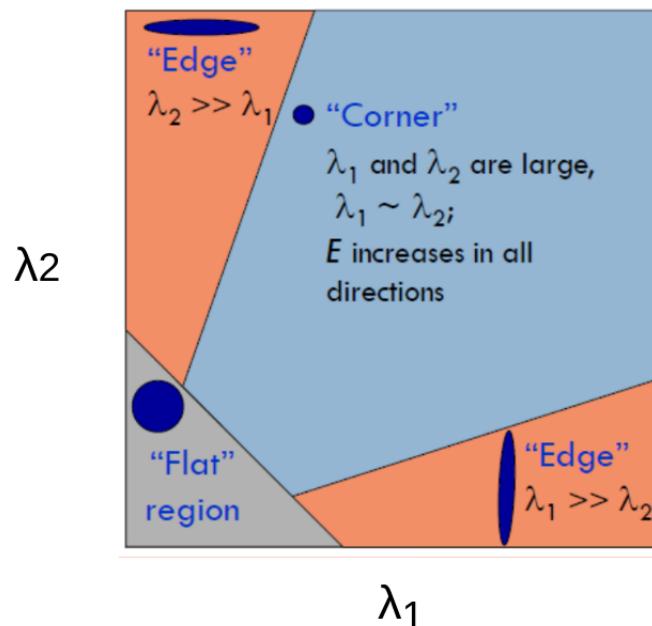


Figura 1: Ilustración de cómo interpretar los valores de λ_1 y λ_2 .

Teniendo los valores anteriores para cada píxel, podemos calcular la media armónica de un píxel, la cuál llamaremos p , de la siguiente forma:

$$p = \frac{\lambda_1 \lambda_2}{\lambda_1 + \lambda_2} \quad (1)$$

Una vez dicho todo esto, vamos a ver cómo se podría implementar:

```

1 def compute_points_of_interest(img, block_size, ksize):
2     """
3         Funcion que calcula los puntos de interes dada una imagen de
4         entrada.

```

```

4     Dichos puntos de interes son calculados mediante el operador de
5     Harris.
6
7     Args:
8         img: Imagen de la que sacar los puntos de interes
9         block_size: Tamaño del bloque que se va a tener en cuenta a
10        la hora de
11           calcular los valores singulares.
12         ksize: Tamaño del operador de Sobel
13
14     Return:
15         Devuelve una imagen del mismo tamaño que la entrada que
16        contiene los
17         puntos de interes calculados con el operador de Harris
18         """
19         # Obtener valores singulares y vectores asociados
20         sv_vectors = cv2.cornerEigenValsAndVecs(img, block_size, ksize)
21
22         # Quedarse solo con los valores singulares
23         # Los valores singulares son los dos primeros valores de la
24         matriz
25         sv = sv_vectors[:, :, :2]
26
27         # Calcular valor de cada pixel como  $\frac{\lambda_1 * \lambda_2}{\lambda_1 + \lambda_2}$ 
28         # Ahi donde el denominador sea 0, se pone un 0, para evitar que
29         # se calcule
30         # un infinito
31         prod_vals = np.prod(sv, axis=2)
32         sum_vals = np.sum(sv, axis=2)
33         points_interest = np.divide(prod_vals,
34             sum_vals,
35             out=np.zeros_like(img),
36             where=sum_vals!=0.0
37         )
38
39     return points_interest

```

Lo primero que hacemos es utilizar la función `cornerEigenValsAndVecs()` de OpenCV para obtener los valores λ_1 , λ_2 y los vectores propios asociados a cada uno de los valores singulares de cada píxel de la imagen. Esto nos dará de salida una matriz de las mismas dimensiones que la de entrada, pero cada posición contendrá los 6 valores anteriormente dichos. Los parámetros que se le pasan son `img`, que es la imagen de donde extraer la información, `block_size`, que indica cuál es el tamaño de la región alrededor del píxel que se debe consultar para obtener los valores singulares, y `ksize`, que indica el tamaño del *kernel* de Sobel que va a utilizar la función.

A continuación nos quedamos solo con los valores singulares, los cuáles están situados en las dos primeras posiciones. Calculamos la suma y el producto para cada par de λ_1 y λ_2 , y después calculamos la media armónica. Para evitar problemas

donde por ejemplo el denominador es 0, la operación solo se realiza para aquellos valores en los que el denominador sea distinto de 0. De esta forma, en las posiciones en las que no se dé, se pondrá un 0, ya que se considerará que no ofrecen información relevante. Dicha operación se puede ver en el código anterior en las líneas 27-31, donde en la línea 29 se declara que la salida será una matriz inicialmente con ceros y en la 30 se especifica que la división solo se haga en aquellas posiciones donde el denominador sea distinto de 0.

Una vez que hemos obtenido los puntos de Hrris, el siguiente paso es aplicar un umbral, de forma que los píxeles de la imagen resultante que están por debajo del valor umbral serán eliminados, poniéndolos a 0. De esta forma, podemos eliminar aquellos puntos con valores bajos, ya que la mayoría de ellos estarán asociados a regiones planas, es decir, regiones donde la intensidad varíe muy poco, y por tanto, donde los valores λ_1 y λ_2 sean bajos. También es posible que en el proceso se elimine algún punto asociado a un borde que no sea muy importante, aunque eso depende bastante del valor umbral que se utilice. En el *paper* utilizaron un valor de 10, aunque nosotros probaremos luego con otros valores.

Para aplicar el umbral a la imagen, hemos hecho una función, la cuál se muestra a continuación:

```

1 def threshold_points_of_interest(points, threshold):
2     """
3         Funcion que aplica un umbral sobre una imagen, poniendo los
4         pixels por
5         debajo del umbral a 0
6
6     Args:
7         points: Puntos/Imagen sobre la que aplicar la umbralizacion
8         threshold: Valor umbral
9
10    Return:
11        Devuelve una imagen en la que los valores por debajo del
12        umbral han
13        sido puestos a 0
14    """
15    points[points < threshold] = 0.0

```

Lo único que se hace es encontrar las posiciones en las que el píxel tenga un valor inferior al umbral y se pone dicho píxel a 0.

Posteriormente, tenemos que aplicar la supresión de no máximos, de forma que eliminamos los valores que no sean máximos locales. De esta forma, eliminamos valores que puedan estar asociados a ruido. El código es casi el mismo que el utilizado en la práctica 1. La única diferencia es que ahora el tamaño de la ventana está parametrizado, pero el funcionamiento sigue siendo el mismo que teníamos anteriormente.

Una vez que hemos aplicado los pasos anteriores, los píxeles que queden “vivos” en la imagen son los que ofrecen información relevante sobre esta, ya que son aquellos que podríamos decir que, en general, ofrecen información sobre las esquinas que se puedan encontrar en la imagen a una escala determinada. Por tanto, podríamos tratar dichos puntos como descriptores o *keypoints*, aunque nos falta algo más de información. Tenemos que conocer, aparte de la posición del punto, la escala en la que se ha detectado y su orientación.

Calcular la escala es algo trivial. Siguiendo las indicaciones proporcionadas, podemos calcular dicho valor como $blockSize \times nivel_piramide$, donde *blockSize* es el tamaño del bloque que se ha utilizado para calcular los valores de λ_1 y λ_2 para cada píxel, y *nivel_piramide* es, como su propio nombre indica, el nivel actual de la pirámide.

No obstante, el cálculo de la orientación no es tan directo como en el caso anterior, ya que necesitamos información sobre los gradientes de un punto determinado en una escala concreta. Afortunadamente, aquí es donde entran en juego las pirámides Gaussianas que calculamos al principio para las derivadas de la imagen, las cuales nos facilitan mucho la vida. Vamos a ver primero la implementación y luego comentaremos lo que se hace:

```

1 def compute_orientation(dx_grad, dy_grad):
2     """
3         Funcion que calcula la orientacion del gradiente de una serie de
4         puntos
5
6     Args:
7         dx_grad: Derivadas en el eje X
8         dy_grad: Derivadas en el eje Y
9
10    Return:
11        Devuelve un array en el que estan las orientaciones de todos
12        los
13        pares de gradientes de dx_grad y dy_grad. Las orientaciones
14        estan
15        en grados, y se encuentran en el rango [0, 360)
16
17    """
18    # Obtener vectores u y sus normas
19    u = np.concatenate([dx_grad.reshape(-1,1), dy_grad.reshape(-1,1)],
20                      axis=1)
21    u_norm = np.linalg.norm(u, axis=1)
22
23    # Calcular vectores [cos \theta, sen \theta]
24    vec_cos_sen = u / u_norm.reshape(-1, 1)
25    cos_vals = vec_cos_sen[:, 0]
    sen_vals = vec_cos_sen[:, 1]

26    # Calcular sen/cos arreglando posibles errores como 0/0 y x/0
27    # Se arreglan los errores poniendolos a 0.0
28    orientations = np.divide(sen_vals,
29                            cos_vals,
```

```

26         out=np.zeros_like(sen_vals),
27         where=cos_vals!=0.0
28     )
29
30     # Obtener \theta usando arcotangente (resultado en radianes
31     # entre [-pi/2, pi/2])
32     orientations_rad = np.arctan(orientations)
33
34     # Obtener angulos y arreglarlos (sumar 180 grados en caso de que
35     # cos < 0
36     # y pasarlo al rango [0, 360], eliminando negativos)
37     orientations_degrees = np.degrees(orientations_rad)
38     orientations_degrees[cos_vals < 0.0] += 180.0
39     orientations_degrees[orientations_degrees < 0.0] += 360.0
40
41     return orientations_degrees

```

La función recibe como parámetro las derivadas en el eje X y en el eje Y de los puntos “vivos” de una escala determinada. Lo primero que hace es calcular el vector de gradientes \mathbf{u} asociado a cada par de derivadas, el cuál viene dado por $\mathbf{u} = [\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}]$. Se calcula dicho vector para todas las parejas a la vez, ya que solo consiste en juntar las derivadas, poniéndolas como vectores columna. También se calcula $|\mathbf{u}|$, que es la norma euclídea de \mathbf{u} .

Después, se divide cada vector \mathbf{u}_i entre su norma $|\mathbf{u}|_i$, resultando en un vector donde tenemos que los valores son $[\cos(\theta), \sin(\theta)]$. Estos valores son el coseno y el seno del ángulo θ que queremos calcular. De aquí sacamos los valores de forma separada, para poder acceder a ellos de forma más sencilla. Los cosenos están en la primera columna, y los senos en la segunda.

Ahora, para obtener la orientación, lo primero que tenemos que hacer es dividir el seno entre el coseno para cada vector. De esta forma obtenemos $\tan(\theta)$. Esta operación puede verse en las líneas 24-28. De nuevo, tal y como pasaba en el caso de los puntos de Harris, es posible que alguno de los senos sea 0. Para evitar que el resultado no sea válido, se aplica la corrección vista anteriormente, donde solo se realiza la operación allí donde el seno sea distinto a 0. En caso contrario, la salida generada es 0.

A partir del resultado anterior ya podemos sacar el ángulo. Como el resultado anterior es la tangente de θ , podemos aplicar la función arco tangente, la cuál es la inversa, para sacar el ángulo. El principal problema es que el valor devuelto está en radianes, y está en el rango $[-\frac{\pi}{2}, \frac{\pi}{2}]$. Por tanto, tenemos que pasar los ángulos de radianes a grados. Dicha transformación se realiza con una función de numpy, la cuál es `degrees()`. Esta función pasará los valores a grados, aunque estarán en el rango $[-90, 90]$. Por tanto, hay una serie de operaciones extra que tenemos que hacer:

1. Tenemos que identificar los puntos cuyo coseno sea negativo, ya que los valores obtenidos anteriormente se sitúan en el primer y en el cuarto cuadrante. Esto viene a raíz de que la tangente es positiva tanto en el primer como en el tercer cuadrante, y es negativa tanto en el segundo como en el cuarto cuadrante. Por tanto, tenemos que ajustar los ángulos obtenidos. Para hacer esto, podemos guiarnos por el coseno, ya que este es positivo en el primer y cuarto cuadrante y negativo en los otros dos. Por tanto, simplemente tenemos que buscar los índices de los cosenos que sean negativos, y sumar a los ángulos en las mismas posiciones 180, de forma que se ajusten al ángulo en el cuadrante que les corresponda.
2. Puede que aun queden valores negativos porque están en el cuarto cuadrante. En un principio, esto no debería ser un problema, pero OpenCV exige que los grados estén en el rango [0, 360). Por tanto, a todos los ángulos menores que 0, hay que sumarles 360.

Después de todo este proceso, ya tenemos las orientaciones calculadas, y podríamos proceder a la creación de *keypoints* con la información que tenemos.

Para simplificar todo el proceso, hemos creado una única función que se encarga de todo el proceso anterior. A continuación se puede ver el código de dicha función:

```

1 def harris_corner_detection(img, block_size, window_size, ksize_der,
2                             n_octaves, threshold=10.0):
3     """
4         Funcion que detecta los puntos de Harris de una imagen a
5             distintas
6             escaslas.
7
8     Args:
9         img: Imagen de la que se quieren extraer los puntos de
10            Harris
11            block_size: Tamaño del bloque que se va a tener en cuenta a
12            la hora de
13                calcular los valores singulares.
14            window_size: Tamaño de la ventana al realizar la supresión
15            de no
16                máximos
17            ksize_der: Tamaño del operador de Sobel (utilizado en el
18            cálculo
19                de los valores singulares)
20            n_octaves: Número de octavas/escalas de la imagen de la que
21            sacar
22                puntos
23            threshold: Umbral utilizado para eliminar todos los valores
24            inferiores
25                a este.
26
27    Return:
28        Devuelve dos listas: una que contiene los keypoints
29        extraídos y otra

```

```

21     que contiene los keypoints corregidos
22 """
23 # Obtener piramide gaussiana de la imagen
24 img_pyr = compute_gaussian_pyramid(img, n_octaves)
25
26 # Obtener piramides de las derivadas
27 dx_pyr, dy_pyr = compute_derivative_pyramids(img, ksize_der,
28 n_octaves)
29
30 # Lista de keypoints y keypoints corregidos
31 keypoints = []
32 corrected_keypoints = []
33
34 for i in range(n_octaves):
35     # Obtener puntos de interes de la escala
36     points_interest = compute_points_of_interest(img_pyr[i],
37         block_size,
38         ksize_der
39     )
40
41     # Aplicar umbralizacion
42     threshold_points_of_interest(points_interest, threshold)
43
44     # Aplicar supresion de no maximos
45     points_interest = non_max_suppression(points_interest,
46     window_size)
47
48     # Obtener valores mayores que 0.0 (aquejlos que no han sido
49     # eliminados)
50     points_idx = np.where(points_interest > 0.0)
51
52     # Calcular escala del KeyPoint
53     # Hace falta incrementar el valor de i en 1 porque se
54     # empieza en 0
55     scale = (i+1) * block_size
56
57     # Obtener las derivadas correspondientes a los puntos no
58     # eliminados
59     dx_grad = dx_pyr[i][points_idx]
60     dy_grad = dy_pyr[i][points_idx]
61
62     # Calcular orientaciones de los puntos no eliminados
63     orientations = compute_orientation(dx_grad, dy_grad)
64
65     # Lista que contiene los keypoints de la octava/escala
66     # Se corrigen las coordenadas segun la escala
67     keypoints_octave = [cv2.KeyPoint(x*2**i, y*2**i, scale, o)
68                         for y, x, o in zip(*points_idx,
69                         orientations)]
70
71     # Unir las coordenadas de forma que sean n vectores [x,y]
72     # formando una
73     # matriz

```

```

67     points_x = points_idx[0].reshape(-1,1)
68     points_y = points_idx[1].reshape(-1,1)
69     points = np.concatenate([points_x, points_y], axis=1)
70
71     # Establecer criterio de parada
72     # Se parara o bien a las 15 iteraciones o cuando epsilon sea
73     # menor a 0.01
74     criteria = (cv2.TERM_CRITERIA_EPS + cv2.
75                 TERM_CRITERIA_MAX_ITER, 15, 0.01)
76
77     # Corregir keypoints
78     points = cv2.cornerSubPix(img_pyr[i],
79                               points.astype(np.float32),
80                               (3,3),
81                               (-1,-1),
82                               criteria
83                           )
84
85     # Redondear, cambiar x por y y viceversa (OpenCV carga las
86     # imágenes
87     # invirtiendo los ejes) y transformar coordenada a la de la
88     # imagen original
89     points = np.round(points)
90     points = np.flip(points, axis=1)
91     points *= 2**i
92
93     # Guardar keypoints y keypoints corregidos
94     keypoints.append(keypoints_octave)
95     corrected_keypoints.append(points)
96
97 return keypoints, corrected_keypoints

```

La función recibe como parámetros la imagen de la que sacar los puntos de Harris, el tamaño de bloque utilizado en el cómputo de los valores singulares (tal y como se ha explicado anteriormente), el tamaño de la ventana que se va a utilizar en la supresión de no máximos el tamaño del *kernel* de la derivada, el número de octavas o escalas que se quieren sacar de la imagen (lo cuál determina el tamaño de las pirámides) y el umbral que se tiene que aplicar para reducir el número de puntos. Por defecto, el umbral está puesto a 10, que es el valor que se probó en el *paper*.

Esta función, aparte de hacer todo lo especificado anteriormente, también se encarga de corregir las posiciones de los *keypoints* que se han obtenido, utilizando para ello la función de OpenCV `cornerSubPix(...)`. De esta forma, realizamos todos los cálculos a la vez, y no necesitamos repetir el proceso. Estos puntos serán utilizados más adelante para comparar qué tal han salido los puntos estimados. De momento, vamos a explicar lo que hace la función, y cuando aparezca la llamada a esta función nos pararemos a explicarla más detenidamente.

Lo primero que hace la función es obtener las pirámides de la imagen y de las derivadas, además de crear las listas que contendrán la información de salida.

Una vez hecho esto, repite todos los pasos descritos anteriormente tantas veces como octavas/escalas se le hayan especificado. Primero extrae los puntos de Harris, después elimina los que están por debajo del umbral y realiza la supresión de no máximos. Una vez hecho esto, determina las posiciones en las que los puntos tienen un valor superior a 0 y las guardan. Después, se encarga de calcular la escala y las orientaciones de la forma descrita anteriormente. Con la información que ha extraído, crea los *keypoints* de la escala, lo cuál se puede ver en las líneas 62-63. Es importante destacar que, a la hora de crear los *keypoints*, se cambian las x por las y y viceversa. Esto se hace así porque si no, a la hora de pintar los *keypoints* posteriormente, estos saldrán girados, ya que OpenCV carga las imágenes invirtiendo los ejes. Por tanto, necesitamos invertir las posiciones manualmente para que posteriormente se puedan ver bien. También es importante destacar que las coordenadas se multiplican por 2^i , donde i es la octava/escala actual. Esto se debe a que, por ejemplo, al pasar de la base de la pirámide al siguiente nivel, lo que originalmente estaba en la posición (x, y) ahora estará en la $(\frac{x}{2}, \frac{y}{2})$. En el siguiente nivel estará en la posición $(\frac{x}{4}, \frac{y}{4})$, y así sucesivamente. Esto se debe a que la imagen se va reduciendo la mitad en cada eje a medida que se va subiendo de nivel en la pirámide Gaussiana. Por tanto, como las coordenadas obtenidas son relativas a las de la imagen de una escala determinada, tenemos que adaptarlas a las de la imagen original multiplicando por este factor.

Una vez que se han extraído los *keypoints*, comienza el proceso en el que se refinan. Para ello, lo primero que se hace es juntar las coordenadas obtenidas en una única matriz, la cuál va a tener dos columnas: una para el eje X y una para el Y . Después de eso, establecemos el criterio de parada del método de refinamiento, el cuál es una tupla que contiene los *flags* y los valores. Con los *flags* se indica que se parará o bien cuando se pase un número de épocas o cuando el movimiento a la hora de refinar en alguna época sea menor a un ϵ dado. En este caso, se especifica que como máximo se hagan 15 iteraciones o el refinamiento se mueva menos de $\epsilon = 0.01$. De esta forma, no hacemos muchas iteraciones, y el valor de ϵ se ha considerado como el adecuado, ya que si se mueve menos que eso, significa que hemos encontrado una posición muy próxima al mejor lugar, y se ha considerado suficiente el poco margen de error que deja. Además, al tener tan pocas iteraciones límite, es difícil ajustar mejor.

Con el criterio ya definido llamamos a la función `cornerSubPix`, la cuál recibe la imagen, las posiciones iniciales de los puntos que tiene que refinar (los cuales hemos obtenido antes), el tamaño de la ventana (el valor que recibe es la **mitad** del tamaño total de la ventana), la región alrededor del píxel en la que no buscar (la mitad de esta, igual que en el caso anterior) y el criterio de parada. En este caso, para el tamaño de la ventana hemos especificado que se use $(3, 3)$, de forma que se busque

en una región 7×7 alrededor del centro, ya que se ha estimado suficiente teniendo en cuenta los tamaños de bloque que se usarán más adelante, los cuáles en ningún caso serán mayores que 5 debido a que el tamaño de la imagen se reduce demasiado rápido a medida que vamos subiendo niveles en la pirámide Gaussiana. En cuanto a la región alrededor de píxel en la que no buscar, he especificado $(-1, -1)$, de forma que se busque en toda la región, ya que se ha considerado que no es buena idea limitar el poco espacio de búsqueda que tenemos, ya que de por sí es pequeño.

Una vez que se obtienen los puntos refinados, se redondean los valores al entero más próximo, se cambian las x por las y y viceversa (línea 86) y se escalan los valores de las coordenadas a los de la imagen original.

Finalmente, se guardan tanto los *keypoints* obtenidos como las posiciones refinadas en las listas correspondientes, y se repite todo el proceso para el número de octavas/escalas que se haya especificado.

1.2. Experimentación con los parámetros

Una vez que hemos descrito cómo se ha llevado a cabo todo el proceso, vamos a experimentar un poco con los parámetros para ver qué *keypoints* se obtienen.

Para hacer las pruebas, vamos a utilizar la imagen *Yosemite1.jpg*. Vamos a extraer los *keypoints* en la imagen en blanco y negro y los pintaremos sobre la imagen en color. Vamos a probar a variar el tamaño del bloque, el tamaño del *kernel* de las derivadas, el tamaño de la ventana y el umbral. En todos los casos utilizaremos 5 escalas, ya que se ha considerado un número adecuado. En caso de haber escogido más, la imagen se hubiese quedado muy pequeña en los últimos niveles, y con menos, a lo mejor se perdía detalle. Con la mejor combinación de parámetros se mostrará un desglose de los *keypoints* por escalas.

El caso base será uno en el que se use un tamaño de bloque de 5, un tamaño de ventana y de *kernel* de derivada 3, 5 escalas y un umbral de 10. El tamaño del bloque se ha escogido de 5 ya que la imagen decrece muy rápido de tamaño, con lo cuál coger uno más grande no tiene mucho sentido. El tamaño de la ventana se ha cogido de 3 porque se cree que es suficiente, ya que busca en una región más o menos pequeña cuál, de forma que no es influido por píxels que están más lejos del actual. Se ha estimado que un tamaño de *kernel* para las derivadas de 3 es suficiente, ya que suele ofrecer buenos resultados. En cuanto al umbral, se ha dejado el umbral por defecto, ya que es el que venía en el *paper*.

En cada caso, se indicará cuáles son los parámetros modificados y cuántos puntos se han detectado a lo largo de todas las escalas. A continuación se pueden ver los resultados:

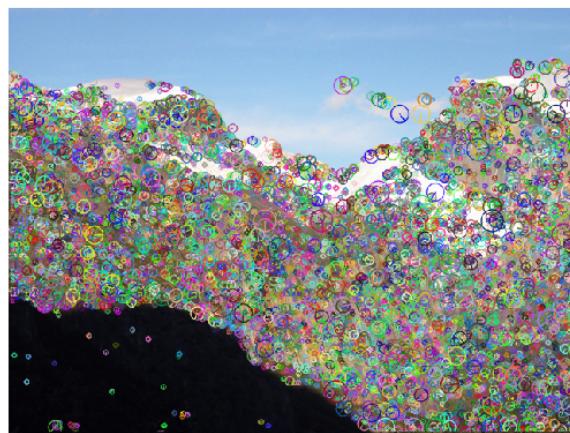


Figura 2: *Keypoints* obtenidos para el caso base. 6731 puntos detectados.

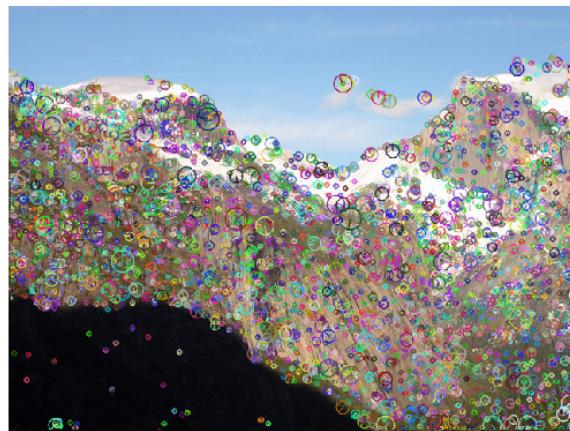


Figura 3: *Keypoints* obtenidos con tamaño de ventana de 5. 2982 puntos detectados.

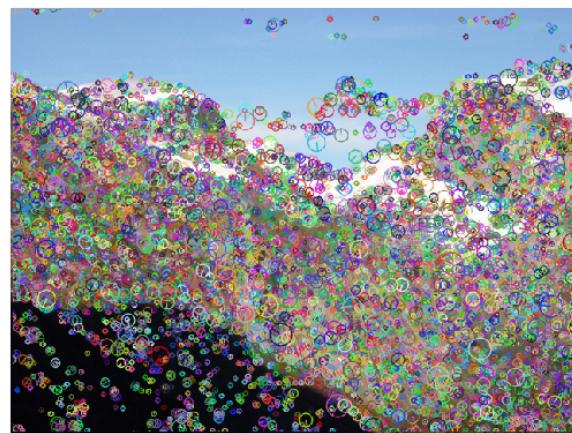


Figura 4: *Keypoints* obtenidos con tamaño del *kernel* de derivadas de 5. 6911 punto detectados.



Figura 5: *Keypoints* obtenidos con tamaño de bloque de 3. 11013 puntos detectados.



Figura 6: *Keypoints* obtenidos con umbral de 60. 5610 puntos detectados.



Figura 7: *Keypoints* obtenidos con umbral de 90. 5113 puntos detectados.

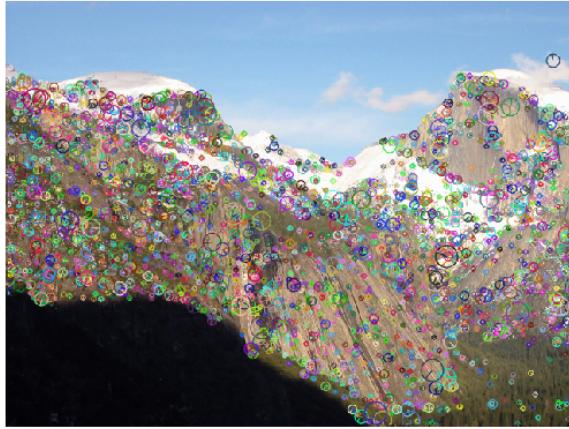


Figura 8: *Keypoints* obtenidos con ventana de tamaño 5 y umbral de 90. 2460 puntos detectados.

Observando los resultados de forma general, podemos ver que en todos los casos se obtienen más de 2000 *keypoints*. Parece que, en general, disminuir el tamaño del bloque y aumentar el tamaño del *kernel* de la derivada hacen que se detecten más puntos que en el caso base. En el primer caso, esto parece lógico, ya que se tienen en cuenta menos píxeles para determinar la variación de la intensidad, y por tanto, las estimaciones se quedan bastante más cortas, ya que posiblemente si se utilizase un entorno mayor, los valores singulares hubiesen sido más pequeños. En el segundo caso, puede deberse a que el operador haga que las variaciones de intensidad sean más notables, ya que ponderaría más a los píxeles cercanos al centro. Por tanto, estaría de alguna forma sobreestimando de alguna forma las variaciones, aunque tampoco estamos muy seguros de que sea esto lo que suceda.

Si en cambio aumentamos el tamaño de la ventana o el del umbral, como es lógico, el número de *keypoints* se ve disminuido. Este descenso es mucho más drástico cuando se aumenta el tamaño de la ventana, ya que se eliminan más puntos debido a que se considera un entorno mucho mayor a la hora de buscar el máximo de la región. Por tanto, encontraremos valores que antes no habíamos considerado dentro de la región, y que por tanto, harán que lo que antes era un máximo local deje de serlo. Esto sin embargo presenta un pequeño problema, y es que no debemos abusar del tamaño de la ventana: un tamaño demasiado grande nos puede hacer perder demasiados puntos. Por tanto, lo suyo sería intentar coger, como mucho, una ventana con las mismas dimensiones que el tamaño del bloque utilizado, de forma que se busque el máximo en el entorno de donde se han extraído los valores singulares. Si en vez de aumentar el tamaño de la ventana aumentamos el umbral,

tal y como dijimos antes, también se reducen el número de puntos encontrados. Esto se debe a que se exige que los valores singulares sean más grandes, y que por tanto, que las variaciones en intensidad sean más pronunciadas. Por tanto, estamos haciendo que la detección sea algo más robusta, ya que buscamos las esquinas más destacables. Aunque, igual que en el caso anterior, no debemos abusar del umbral, ya que uno muy grande nos puede hacer empezar a perder puntos que pudiesen ser clave.

Vemos que si combinamos tanto el incremento del umbral como el aumento del tamaño de la ventana, como se ha hecho en la figura 8, obtenemos un conjunto de *keypoints* que parece ser razonable, ya que son más de 2000, pero no es un número excesivamente grande. Además de eso, parece que se dejan de detectar algunos puntos que no tenían mucho sentido. Por ejemplo, no se detectan los puntos de la zona oscura, los cuáles sí que se detectaban cuando solo se aumentaba el tamaño de la ventana, y no se detectaban al aumentar umbral. Estos puntos no tenían mucho sentido, ya que realmente no hay ninguna esquina ahí y no parece que pueda ofrecer mucha información. Además, a simple vista, aunque haya cierta variación de la intensidad, esta no es muy significativa.

Para dibujar los *keypoints* anteriores se ha utilizado la siguiente función:

```

1 def draw_all_keypoints(img, keypoint_list):
2     """
3         Funcion que dibuja todos los keypoints detectados
4
5     Args:
6         img: Imagen sobre la que pintar los keypoints
7         keypoint_list: Lista con los keypoints
8     """
9
10    # Juntar los keypoints de todas las escalas
11    keypoints= [k for keypoints_octave in keypoint_list for k in
12    keypoints_octave]
13
14    # Transformar imagen a uint8 y RGB
15    vis = transform_img_uint8_RGB(img)
16
17    # Crear imagen de salida vacia del mismo tamaño que la original
18    out = np.empty_like(vis)
19
20    # Dibujar keypoints
21    out = cv2.drawKeypoints(vis,
22                           keypoints,
23                           out,
24                           flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS
25                           )
26
27    # Visualizar la imagen
28    visualize_image(out)

```

Esta función recibe la imagen sobre la que se quieren pintar los *keypoints* y una lista con todos los puntos extraídos. Lo primero que hace es juntar todos los *keypoints* extraídos a distintas escalas en una única lista. Después transforma la imagen a entero sin signo y al formato RGB, ya que se usa `matplotlib`. Es necesario que la imagen esté en entero sin signo, ya que la función `drawKeypoints` obliga a que sea de este tipo para poder pintar los *keypoints*. Después de esto, se crea una imagen de salida, la cuál inicialmente está vacía, y que es donde escribirá el resultado. Finalmente, se pinta cada *keypoint* sobre la imagen de salida utilizando la función `drawKeypoints`, a la cuál se le pasa la imagen de entrada, los puntos, la imagen de salida, y un *flag*, que en este caso indica que se pinten los *keypoints* utilizando un formato rico en el que se muestran círculos de distintos colores, escalas y orientaciones.

Una vez que hemos comentado esto, vamos a mostrar ahora los *keypoints* que se han detectado en cada escala para alguno de los casos anteriores. En este caso, vamos a utilizar los puntos que se pueden ver en la figura 8, ya que se considera que es el mejor conjunto de *keypoints* que se han extraído, tanto porque no son demasiados como porque parecen ser relevantes, sin incluir puntos que puedan aportar poca o nula información.

Para dibujar dichos puntos nos hemos ayudado de la siguiente función:

```

1 def draw_keypoints_octave(img, keypoint_list):
2     """
3         Funcion que dibuja los keypoints por cada escala
4
5     Args:
6         img: Imagen sobre la que pintar los keypoints
7         keypoint_list: Lista con los keypoints
8     """
9
10    # Transformar imagen a uint8 y RGB
11    vis = transform_img_uint8_RGB(img)
12
13    for keypoints in keypoint_list:
14        # Crear imagen de salida vacia del mismo tamaño que la
15        # original
16        out = np.empty_like(vis)
17
18        # Dibujar keypoints de la octava
19        out = cv2.drawKeypoints(vis,
20                               keypoints,
21                               out,
22                               flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS
23                               )
24
25        # Visualizar la imagen
26        visualize_image(out)

```

La función es muy parecida a la que teníamos antes. Lo único que cambia es

que no juntamos *keypoints* en una única lista y que pintamos de forma individual los *keypoints* de cada escala.

Una vez dicho esto, vamos a ver qué puntos hemos detectado en cada escala:



Figura 9: *Keypoints* detectados en el primer nivel.



Figura 10: *Keypoints* detectados en el segundo nivel.

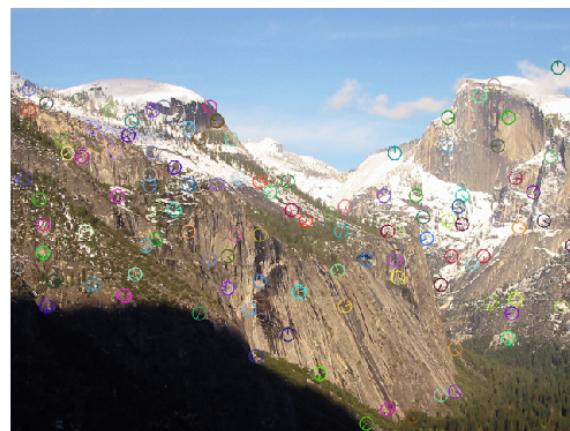


Figura 11: *Keypoints* detectados en el tercer nivel.

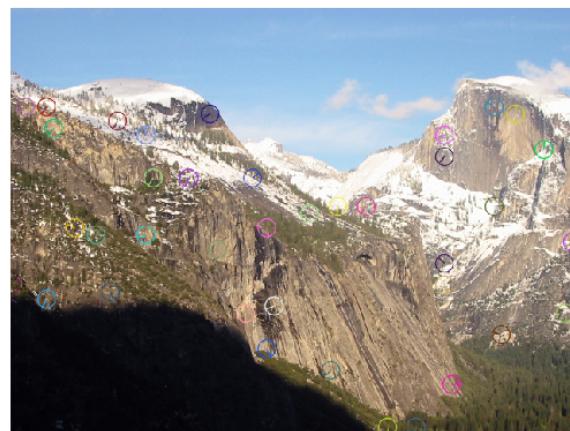


Figura 12: *Keypoints* detectados en el cuarto nivel.

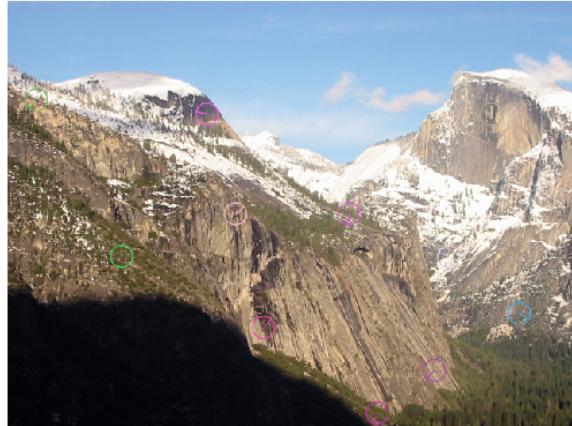


Figura 13: *Keypoints* detectados en el quinto nivel.

Vemos que a medida que extraemos *keypoints* de niveles más elevados de la pirámide, el número que extraemos es cada vez menor, ya que al hacerse la imagen cada vez más pequeña, el número de píxeles que pueden ser un *keypoint* son cada vez menos. Además, esta organización en escalas provoca también que la información no se extraiga de golpe, sino que se haga poco a poco a medida que se va aumentando la escala. La información que se extraiga en las escalas más bajas no va a ser obviamente la misma que la que se extrae en los más altos, como es lógico, aunque en algunos casos, los *keypoints* que se extraen en escalas consecutivas pueden estar relativamente cerca, como por ejemplo se puede ver en las imágenes 9 y 10.

En general, parece que los *keypoints* están localizados dentro de la región iluminada de la montaña, y en muy pocas ocasiones se saca un punto que no esté dentro de esta región. Un ejemplo de esto se puede ver en la figura 11, donde se extrae un punto que se localiza en una de las nubes. Dicha nube se encuentra como tal fuera de la región donde se han extraído todos los puntos anteriormente, y por tanto, es el único punto que se ha extraído en todas las escalas que no se encuentra localizado en la región previamente mencionada.

Por tanto, como pequeña conclusión, podemos decir que es importante que al realizar la extracción de descriptores, este proceso se haga en múltiples escalas, ya que permite extraer distinta información de distintas partes de la imagen de forma efectiva. Hacerlo con una única escala no nos permitiría obtener toda la información directamente, ya que en general hay mucha información que no se puede obtener observando solo la imagen original. Además, otra cosa muy importante es que nos permite obtener información invariantes a las escalas, ya que los descriptores

no dependen de que se hayan encontrado de una única forma, si no que pueden ser detectados de distintas formas en distintas escalas, haciéndolos por tanto más flexibles para usos futuros.

1.3. Comparación de la estimación con los puntos refinados

Por último, para concluir este apartado, vamos a estudiar cómo de buenas son las estimaciones de los puntos que hemos realizado anteriormente. Para ello, utilizaremos la información que hemos extraído con la función `cornerSubPix`. Vamos a escoger, de entre aquellos *keypoints* que no coincidan con los puntos refinados, tres pares de puntos. Pintaremos primero unos círculos pequeños alrededor de los puntos estimados y de los corregidos y, en una imagen diferente, unos círculos más grandes, para que podamos observar dónde se encuentran las diferencias en la imagen original. Los círculos serán de color **verde** para los puntos estimados y de color **rojo** para los puntos corregidos. Finalmente, cogeremos una región 11×11 alrededor del *keypoint* original en la imagen que hemos dibujado el círculo pequeño, y mostraremos dicha región aumentando 5 veces la imagen, es decir, aplicándole un zoom $\times 5$. De esta forma, podremos ver de más cerca las diferencias entre los puntos estimados y los corregidos. El motivo de coger una región de ese tamaño es más que nada para que se pueda visualizar mejor, ya que el *keypoint* original quedará en el centro, y el otro estará en alguna posición cercana a la suya.

Para realizar esta parte hemos utilizado la imagen *Yosemite2.jpg*, de la cuál hemos extraído los correspondientes *keypoints*, los cuales se pueden ver en la figura 14, y las coordenadas refinadas de los puntos. Hemos decidido utilizar esta imagen para variar un poco, y para ver cómo funciona la estimación en una imagen diferente. Para extraer los *keypoints* nos hemos guiado un poco por los parámetros que se utilizaron para la imagen 8. Por tanto, hemos utilizado un tamaño de bloque y de ventana de 5, un tamaño de *kernel* de derivada de 3 y un umbral de 80, rebajándolo un poco del umbral de 90 que utilizamos anteriormente. Con estos parámetros podemos decir, antes de ver los resultados, que se van a extraer una cantidad suficiente de puntos, y que estos van a ser bastante buenos en general (recordemos los resultados del caso anterior), ya que el tamaño del bloque y del *kernel* de la derivada son los adecuados, además de que con el umbral especificado vamos a buscar puntos con valores singulares fuertes (es decir, donde haya una diferencia notable de intensidad), y con el tamaño de ventana haremos una supresión de no máximos efectiva, ya que buscará el máximo con una ventana del mismo tamaño que el tamaño de bloque utilizado para extraer los puntos Harris.

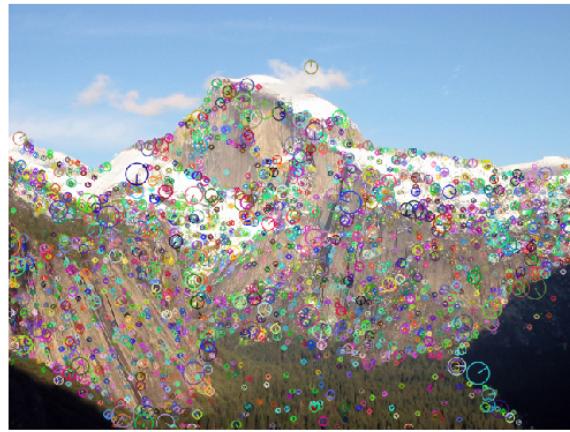


Figura 14: *Keypoints* estimados para la imagen *Yosemite2.jpg*. 2380 puntos estimados.

Una vez que hemos visto los *keypoints* extraídos, vamos a ver las comparaciones. Para mostrarlas, hemos creado la siguiente función:

```

1 def compare_keypoints_orig_corrected(img, keypoints, corrected):
2     """
3         Funcion que compara los keypoints estimados con los corregidos.
4         Escoge tres pares de puntos y dibuja un circulo verde alrededor
5         del keypoint
6         original y uno rojo alrededor el keypoint corregido. Se muestra
7         el resultado
8         en una imagen 11x11 a la que se la hecho un zoom x5.
9
10    Args:
11        img: Imagen sobre la que pintar los circulos
12        keypoints: Lista con los keypoints originales
13        corrected: Lista con las coordenadas de los keypoints
14        corregidos
15        """
16
17        # Transformar datos de entrada a matrices de numpy
18        keypoints_coord = np.array([list(k.pt) for keys in keypoints for
19                                     k in keys], dtype=np.int)
20        corrected_coord = np.array([c for correct_octave in corrected
21                                    for c in correct_octave], dtype=np.int)
22
23        # Encontrar los indices de los keypoints que difieren
24        same_x = keypoints_coord[:, 0] == corrected_coord[:, 0]
25        same_y = keypoints_coord[:, 1] == corrected_coord[:, 1]
26        same_values = same_x * same_y
27        idx_different_vals = np.where(same_values == False)
```

```

22
23     # Escoger keypoints que mostrar (escoger sin repeticion)
24     random_keypoints = np.random.choice(idx_different_vals[0], 3,
25                                         replace=False)
26
27     # Normalizar imagen y pasarla a RGB
28     vis = transform_img_uint8_RGB(img)
29
30     # Copiar la imagen para pintar los circulos en grande
31     vis_big = np.copy(vis)
32
33     # Dibujar keypoints originales y corregidos en pequeño primero y
34     # en grande luego
35     for k, c in zip(keypoints_coord[random_keypoints],
36                      corrected_coord[random_keypoints]):
37         vis = cv2.circle(vis, tuple(k), 2, (0, 255, 0))
38         vis = cv2.circle(vis, tuple(c), 2, (255, 0, 0))
39
40     vis_big = cv2.circle(vis_big, tuple(k), 7, (0, 255, 0))
41     vis_big = cv2.circle(vis_big, tuple(c), 7, (255, 0, 0))
42
43     # Visualizar circulos grandes
44     visualize_image(vis_big)
45
46     for idx in random_keypoints:
47         # Obtener pixels donde el keypoint original y el corregido
48         # no coinciden
49         x_kp, y_kp = np.flip(keypoints_coord[idx])
50
51         # Obtener la region 11 \times 11
52         region = np.copy(vis[x_kp-5:x_kp+6, y_kp-5:y_kp+6])
53
54         # Hacer un resize de la region (un zoom)
55         region = cv2.resize(region, None, fx=5, fy=5)
56
57         # Mostrar imagen
58         visualize_image(region)

```

La función no es muy difícil de entender, pero vamos a comentarla brevemente. Lo primero que hacemos es crear dos listas que contienen las coordenadas de los *keypoints* originales y la de los corregidos. En el primer caso, se obtienen a partir del atributo `pt` de la clase `KeyPoint`. En el segundo caso, solo se tienen que juntar los puntos obtenidos para cada octava en una única lista.

A continuación, buscamos las posiciones en las que los puntos originales y los corregidos no coinciden. Para ello, primero comparamos la primera coordenada de los puntos originales con la primera de los corregidos, y obtenemos las posiciones de las listas donde coinciden. Hacemos lo mismo para la segunda coordenada y obtendremos las posiciones donde coinciden. Para tener las posiciones donde las dos coordenadas coinciden, simplemente tenemos que hacer el producto de los re-

sultados anteriores. Al ser valores booleanos, el producto solo va a ser verdadero en las posiciones de las listas donde las coordenadas de los puntos originales y los corregidos coincidan. Por tanto, una vez que hemos obtenido el producto, solo tenemos que obtener las posiciones donde este producto sea falso, es decir, donde no coincidan. Todo este proceso se puede ver en las líneas 18-21.

Después, utilizando las posiciones obtenidas anteriormente, escogemos tres de ellas de forma aleatoria sin repeticiones. Esto puede verse en la línea 24.

A continuación, transformamos la imagen de salida a entero sin signo de 8 bits y hacemos una copia de esta última para pintar los círculos en grande. Para los índices que hemos obtenido anteriormente, accedemos a los coordenadas correspondientes en cada lista y pintamos un círculo pequeño y uno grande según lo especificado anteriormente.

Una vez que los hemos dibujado, mostramos primero los círculos grandes. Para los círculos pequeños, accedemos a cada una de las posiciones obtenidas aleatoriamente y obtenemos las coordenadas x e y asociadas al punto estimado. Como en los *keypoints* hemos guardado las coordenadas correctas para nosotros y no para OpenCV, tenemos que deshacer la inversión hecha anteriormente. Por tanto, tenemos que volver a invertir las coordenadas obtenidas anteriormente. Todo esto puede verse en la línea 45.

Ahora, tal y como se ve en la línea 48, obtenemos la región 11×11 y le hacemos una copia, por si al hacer zoom se modifica la imagen original. Para hacer el zoom, utilizamos la función `resize`, la cuál recibe como parámetro la imagen sobre la que aplicar el zoom, el tamaño de la imagen de salida (con `None` especificamos que el tamaño sea el producto de los dos siguientes parámetros por las dimensiones correspondientes) y el factor de zoom que aplicar en cada eje. En este caso, tenemos que hacer un zoom de 5 en cada eje, por tanto se lo indicamos de la forma antes vista. Finalmente, dibujamos la imagen de la región.

Una vez dicho esto, vamos a ver qué resultados se han obtenido. A continuación se puede ver la imagen con los círculos en grande:

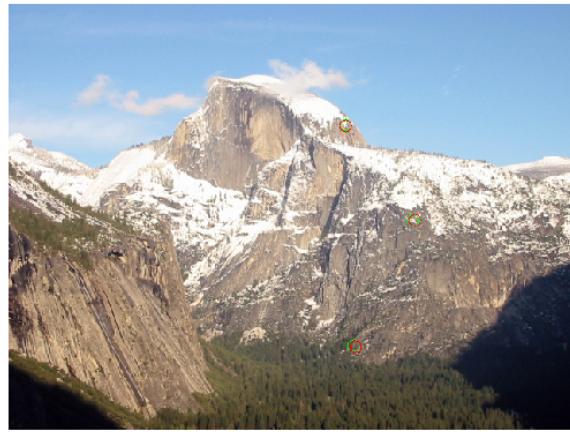
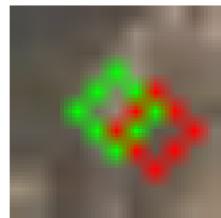
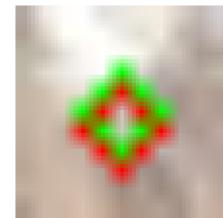


Figura 15: Localizaciones en la imagen original de los *keypoints* estimados (verde) y los corregidos (rojo).

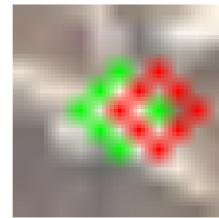
Vemos que, a primera vista, existen ciertas diferencias, aunque sean pequeñas. Si observamos los resultados más de cerca, podemos ver lo siguiente:



(a) Corrección parte baja de la montaña.



(b) Corrección cima de la montaña.



(c) Corrección parte media de la montaña.

Figura 16: Correcciones de cada *keypoint* obtenido.

Vemos que, efectivamente, las correcciones desplazan solo un poco a los puntos estimados. Si observásemos el valor de la intensidad del punto corregido (el centro del círculo rojo) veríamos que, efectivamente, dicho punto tienen una mayor intensidad. También tenemos que tener en cuenta otros factores que pueden hacer que los valores corregidos estén tan próximos al estimado, como por ejemplo el tamaño de la región a buscar, la zona a ignorar y los criterios de parada (recordemos que estamos utilizando los mismos valores de antes).

De aquí podemos sacar algunas conclusiones. La primera es que las estimaciones realizadas son bastante buenas, ya se han equivocado solo por unos pocos píxeles. En segundo lugar, es normal que se den este tipo de errores, ya que recordemos que estamos utilizando operaciones con números en coma flotante, y por tanto, se podrían obtener errores de precisión. Además, todas las técnicas que extraen descriptores de una imagen cometan cierto error, debido por ejemplo a la presencia de ruido en la imagen o a errores en los cálculos. Aparte de todo esto, al corregir los *keypoints*, también redondeamos los valores obtenidos para que las posiciones sean valores enteros, y por tanto, al discretizar, estamos perdiendo algo más de precisión.

Por tanto, como resumen general, podemos ver que la aproximación que hemos tomado nos ha permitido obtener unos resultados bastante buenos y que tienen un margen de error relativamente pequeño, ya que el punto de Harris real estará bastante cerca del estimado.

2. EXTRACCIÓN DE DESCRIPTORES AKAZE

En esta segunda sección vamos a utilizar el descriptor AKAZE para extraer información de la imagen, con el objetivo de hacer *matches* entre dos imágenes, para ver si tienen puntos en común. Vamos a comparar también dos criterios para realizar el *match*, y veremos cuál de ellos es mejor.

2.1. Obtención de los descriptores AKAZE

El descriptor AKAZE es algo más sofisticado que lo que hemos visto hasta ahora, ya que obtiene información a partir de un espacio de escalas no lineal, aunque no vamos a entrar en mucho detalle en como funciona.

OpenCV permite realizar el cálculo de este descriptor de forma relativamente sencilla. Para ello, disponemos de la clase `AKAZE`, la cual permite calcular tanto los *keypoints* como los descriptores con el método `detectAndCompute`. Vamos a ver cómo se haría esto:

```

1 def compute_akaze_keypoints_descriptors(img, threshold=0.1):
2     """
3         Funcion que calcula los keypoints y los descriptores de una
4             imagen
5                 utilizando AKAZE
6
7     Args:
8         img: Imagen de la que extraer la informacion
9             threshold: Umbral de AKAZE (no se aceptan puntos por debajo
10                del umbral)
11
12     Return:
13         Devuelve los keypoints y los descriptores extraidos
14
15     # Crear objeto AKAZE para extraer las caracteristicas
16     akaze = cv2.AKAZE_create(threshold=threshold)
17
18     # Extraer keypoints y descriptores
19     keypoints, descriptors = akaze.detectAndCompute(img, None)
20
21     return keypoints, descriptors

```

Esta función permite obtener tanto los *keypoints* como los descriptores de una imagen de entrada utilizando AKAZE. Lo primero que se hace es crear un objeto de la clase AKAZE con la función `AKAZE_create`, ya que el constructor de AKAZE está deprecado. A esta función le podemos especificar algunos parámetros, pero uno de los más interesantes es el umbral. Este valor permite descartar aquellos puntos que estén por debajo de él. Por tanto, a mayor umbral, menos puntos se generarán. Por defecto, se ha especificado que el umbral sea 0.1, ya que se considera un valor razonable para eliminar aquella información que no sea muy relevante y dejar aquella que nos pueda ser útil.

Una vez hecho esto, simplemente tenemos que obtener la información utilizando el método `detectAndCompute`, el cuál recibe como parámetros una máscara que utilizar en los cálculos (la cuál no utilizaremos, y por tanto ponemos a `None`) y la imagen de la que extraer la información. Finalmente, devuelve los valores calculados, los cuales serán utilizados más adelante en las siguientes partes de esta sección.

2.2. Matching mediante BruteForce y Cross Check

El primer método de *matching* que veremos es el de Brute Force con Cross Check. Este es el método más sencillo posible, ya que consiste en relacionar todos los puntos de una imagen con todos los de la otra (la parte de fuerza bruta). De esta forma, se tendrían la distancia de cada punto p_{1i} extraído de la primera imagen con cada punto p_{2j} extraído de la segunda. Para establecer el *match*, solo bastaría con mirar si el punto más cercano a p_{1i} es el p_{2j} y si el más cercano al p_{2j} es el p_{1i}

(la parte de Cross Check). En caso de que la condición anterior se vea satisfecha, tendremos un *match*. En caso contrario, no habrá *match*. Por tanto, es posible que haya puntos de las dos imágenes que no queden emparejados.

Vamos a ver cómo podríamos calcular esto utilizando las funciones de OpenCV:

```

1 def brute_force_crosscheck_matcher(desc1, desc2):
2     """
3         Funcion que realiza el match entre dos conjuntos de descriptores
4             dados
5             utilizando Brute Force y Cross Checking
6
7     Args:
8         desc1: Descriptor de la primera imagen
9         desc2: Descriptor de la segunda imagen
10    Return:
11        Devuelve los matches entre las dos descripciones
12    """
13    # Crear matcher (va a utilizar el metodo Brute Force + Cross
14    # Check)
15    bf = cv2.BFMatcher_create(crossCheck=True)
16
17    # Obtener los matches
18    matches = bf.match(desc1, desc2)
19
20    return matches

```

La función anterior recibe los descriptores de dos imágenes. Lo primero que hace es crear una instancia de la clase `BFMatcher` utilizando la función `BFMatcher_create` (el constructor está deprecado). Esta función tiene un parámetro que es `crossCheck`, el cuál está puesto por defecto a falso. Cuando se pone a verdadero, se indica que se quiere realizar el Cross Check, el cuál ha sido descrito anteriormente. Finalmente, para obtener los *matches* simplemente basta con llamar al método `match`, el cuál recibe como parámetros los descriptores. El orden en el que se pasan los descriptores no influye en el resultado. Lo único que pasa es que los descriptores que hagan *match* del primer parámetro se guardarán en el atributo `queryIdx`, mientras que los del segundo se guardarán en el atributo `trainIdx`.

2.3. *Matching* mediante Lowe-Average-2NN

El segundo método de *matching* que vamos a ver es el que propone David Lowe en su trabajo [2]. Según la propuesta, para cada punto extraído de la primera imagen, al que llamaremos p_{1i} , hay que encontrar los dos puntos más cercanos a este en la segunda imagen, a los cuales llamaremos $p_{2,mejor1}$ y $p_{2,mejor2}$. Para

establecer si hay *match* o no se tiene que cumplir que:

$$d(p_{1i}, p_{2,mejor1}) < 0.8d(p_{1i}, p_{2,mejor2}) \quad (2)$$

donde d es la distancia. Se ha comprobado que el factor 0.8 que multiplica la segunda distancia permite obtener unos buenos resultados en general, ya que es capaz de descartar el 90 % de los falsos *matches* y de descartar por error menos del 5 % de los *matches* verdaderos. Por tanto, utilizaremos dicho valor también en nuestro trabajo.

Obtener los dos puntos más cercanos a uno puede parecer difícil al principio, pero afortunadamente OpenCV ofrece herramientas para llevarlo a cabo. La siguiente función es un ejemplo de esto:

```

1 def nn2_matcher(desc1, desc2):
2     """
3         Función que realiza el match entre dos conjuntos de descriptores
4             dados
5             utilizando 2NN
6
7     Args:
8         desc1: Descriptor de la primera imagen
9         desc2: Descriptor de la segunda imagen
10    Return:
11        Devuelve los matches entre las dos descripciones
12    """
13    # Crear matcher
14    knn = cv2.BFMatcher_create()
15
16    # Obtener matches
17    matches = knn.knnMatch(desc1, desc2, k=2)
18
19    return matches

```

La función de nuevo recibe dos conjuntos de descriptores de dos imágenes diferentes. Lo primero que hacemos es crear una instancia de **BFMatcher**, solo que esta vez no vamos a utilizar el Cross Check, debido a que luego aplicaremos el criterio de Lowe para eliminar aquellos *matches* que no sean buenos.

Para hacer el *match* con los dos puntos más cercanos, podemos utilizar el método **knnMatch**, el cuál recibe las dos descripciones y k , el número de puntos más cercanos que queramos sacar. En este caso, como queremos los dos puntos más cercanos, tenemos que establecer $k=2$. La función devuelve finalmente una lista con los *matches*, donde cada elemento de la lista es una tupla donde el primer elemento es *match* más cercano, y el segundo elemento el segundo *match* más cercano.

Una vez vista la forma de extraer los puntos, vamos a ver cómo se podrían

determinar los mejores *matches*.

```

1 def lowe_average_2nn_matcher(desc1, desc2):
2     """
3         Funcion que realiza el match entre dos descriptores dados
4         utilizando 2NN con Lowe
5
6     Args:
7         desc1: Descriptor de la primera imagen
8         desc2: Descriptor de la segunda imagen
9
10    Return:
11        Devuelve los matches entre las dos descripciones
12    """
13
14    # Obtener matches
15    matches = nn2_matcher(desc1, desc2)
16
17    # Quedarse el mejor match segun el criterio de Lowe
18    # El mejor match es m1 sii m1 < 0.8 * m2
19    lowe_matches = [m1 for m1, m2 in matches if m1.distance < 0.8*m2.distance]
20
21    return lowe_matches

```

Esta función parte de los descriptores de dos imágenes que se le proporcionan. Lo primero que hace es obtener los *matches* llamando a la que buscaba los 2 puntos más cercanos al actual. Ahora ya solo nos queda procesar los *matches* y quedarnos con aquellos que cumplan el criterio. Para ello, recorremos cada pareja de la lista y obtenemos los dos *matches*, m_1 y m_2 , y miramos si se cumple que $d(m_1) < 0.8d(m_2)$. En caso de que se cumpla, guardamos el *match* m_1 . En caso contrario, no guardamos nada.

2.4. Comparativa

Una vez que hemos visto cómo se implementarían, vamos a ver dos ejemplos comparativos para ver cómo se comportan. Vamos a mirar un ejemplo más sencillo, que es el de las imágenes de Yosemite, y un segundo ejemplo más complicado con las imágenes de un tablero. El objetivo es ver cómo se podrían relacionar cada pareja de imágenes, y ver quién lo hace mejor. Para hacer un análisis menos sesgado hacia uno u otro, vamos a escoger en cada caso 100 *matches* de todos los obtenidos.

Para hacer esto, vamos a sacar primero los *keypoints* y los descriptores con AKAZE, y después haremos el *matching* de cada tipo. Para evitar problemas, usaremos en esta parte las imágenes en escalas de grises, y pintaremos los resultados con las imágenes a color, para que quede algo más vistoso, tal y como hacíamos antes. Para pintar los *matches* nos vamos a ayudar de la siguiente función:

Para poder dibujar los *matches*, vamos a ayudarnos de la siguiente función:

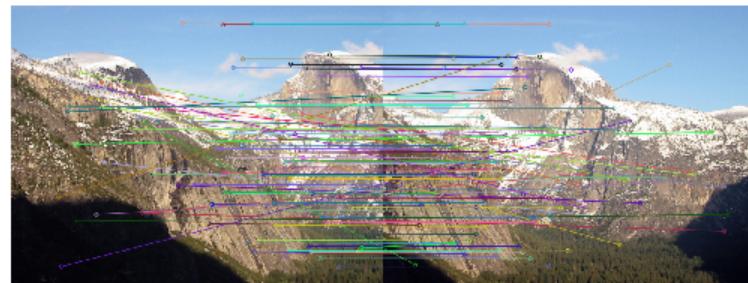
```

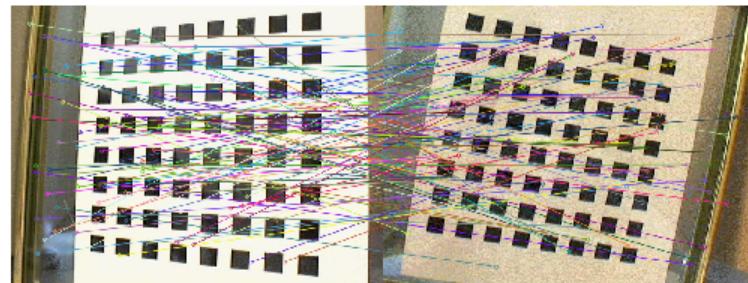
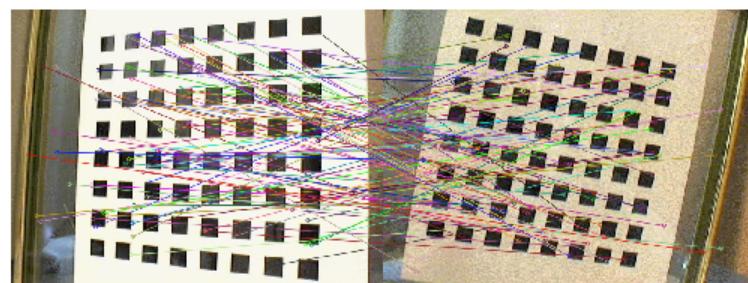
1 def draw_matches(img1, img2, keypoints1, keypoints2, matches):
2     """
3         Funcion para dibujar 100 matches aleatorios obtenidos entre dos
4         imagenes,
5         utilizando para ello los keypoints obtenidos
6
7     Args:
8         img1: Primera imagen
9         img2: Segunda imagen
10        keypoints_1: Keypoints de img1
11        keypoints_2: Keypoints de img2
12        matches: Matches obtenidos entre img1 e img2
13
14    # Transformar imagenes a uint8 y RGB y concatenarlas
15    vis1 = transform_img_uint8_RGB(img1)
16    vis2 = transform_img_uint8_RGB(img2)
17    out_img = np.concatenate([vis1, vis2], axis=1)
18
19    # Obtener 100 matches aleatorios
20    rand_matches = np.random.choice(matches, 100, replace=False)
21
22    # Dibujar matches
23    out_img = cv2.drawMatches(vis1,
24        keypoints1,
25        vis2,
26        keypoints2,
27        rand_matches,
28        out_img,
29        flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS
30    )
31
32    # Mostrar imagen
33    visualize_image(out_img)

```

Esta función recibe las imágenes que dibujar, junto con los *keypoints* extraídos para cada una de ellas y los *matches*. Lo primero que hace es pasar las imágenes a entero sin signo y juntarlas en una única imagen, utilizando para ello la función `concatenate` de `numpy`. A continuación, coge 100 de los *matches* obtenidos de forma aleatoria. Para dibujar los *matches* se utiliza la función `drawMatches`, la cuál recibe como parámetro la primera imagen y los *keypoints* extraídos para ella, la segunda imagen y sus *keypoints*, los *matches*, la imagen donde se escribirá la salida y una serie de *flags*. En este caso, con el *flag* que se ha utilizado se indica que se van a pintar solo aquellos *keypoints* de los que se pinten los *matches*, ya que en caso contrario se pintarían todos, incluso aquellos que no tienen *matches* o aquellos que sí que tienen pero que no se pintan. Finalmente se visualiza la imagen.

Una vez que hemos explicado esto, vamos a hacer la comparativa entre las dos técnicas:

(a) *Matching* utilizando BruteForce y Cross Check.(b) *Matching* utilizando Lowe-Average-2NN.Figura 17: *Matching* para el conjunto de fotografías Yosemite.

(a) *Matching* utilizando BruteForce y Cross Check.(b) *Matching* utilizando Lowe-Average-2NN.Figura 18: *Matching* para el conjunto de fotografías Yosemite.

Para las imágenes de Yosemite, tal y como se puede ver en la figura 17, se ve que el *matching* con BruteForce y Cross Check es algo peor que el otro ya que se cruzan bastante más líneas. En el caso de Lowe, vemos que solo hay una línea que se cruza. Por tanto, parece que en este caso Lowe tiene más de un 95 % de aciertos en cuanto a los *mathces*, ya que el resto de líneas sí que son paralelas entre sí y parecen corresponderse a los puntos. Este valor es aproximado, ya que con una resolución tan baja y tantos puntos es difícil de determinar si todos los puntos están unidos correctamente o no.

A parte de las líneas, vemos que los puntos tienen más sentido en el caso de Lowe que en el otro, ya que aquí no hay ningún punto en la parte oscura de la imagen que se enlace con la parte iluminada. Parece que la mayoría de puntos que saca Lowe se parecen a los que sacaría una persona, ya que la mayoría de ellos están en la zona común de ambas imágenes. Muy pocos puntos están fuera de esta zona. En el caso de la fuerza bruta, hay muchos más puntos que están fuera de la zona común, con lo cuál no es tan bueno como Lowe.

En el caso de las figuras de los tableros, el cuál se puede ver en la figura 18, vemos que ambas técnicas tienen problemas, ya que es un problema demasiado difícil de resolver incluso para un humano. Incluso a nosotros mismos nos cuesta saber cuáles son las correspondencias entre las dos imágenes, ya que al ser los tableros objetos simétricos, es fácil confundir una parte del tablero con otra. Parece que al pasar de una imagen a la otra se produce una rotación y un cambio de intensidad, con lo cuál va a ser algo más difícil determinar las correspondencias a diferencia del caso anterior, donde las imágenes sí que eran más parecidas y podían ser unidas fácilmente, ya que se correspondían mejor.

Si observamos los resultados, vemos que en ambas técnicas casi todas las líneas se cruzan. Esto se debe a que no existe un patrón tan claro para obtener los *matchings*, ya que hay bastantes posibles transformaciones que se podrían hacer para adaptar una imagen a la otra. También existen diferencias respecto a los puntos que se sacan, ya que Lowe saca más puntos del tablero que del marco donde se encuentra, a diferencia de BruteForce con Cross Check. Por tanto, si consideramos todos estos factores de forma conjunta, en este caso no podemos afirmar que una técnica sea mejor que la otra, ya que ambas han ofrecido unos resultados muy pobres.

Por tanto, de aquí podemos concluir que, en general, Lowe ofrece unos mejores resultados que BruteForce con Cross Check, ya que en el ejemplo del Yosemite, Lowe ha acertado casi todas las correspondencias. Además de eso, Lowe tiene un mejor criterio para decidir los *matches*, ya que tiene que comprobar que el punto más cercano a uno concreto no se encuentra relativamente cerca del segundo punto más cercano. Por tanto, de esta forma se asegura de que no se escoja un punto sobre otro en caso de que exista duda de cuál es el mejor, ya que ambos son candidatos

a serlo.

No obstante, ambas técnicas tienen problemas, ya que les cuesta encontrar correspondencias buenas en casos donde existe mucha simetría, debido a que es muy fácil confundir distintas partes de las imágenes.

3. MOSAICO DE DOS IMÁGENES

4. MOSAICO DE N IMÁGENES

Referencias

- [1] M. Brown, R. Szeliski, and S. Winder. Multi-image matching using multi-scale oriented patches. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 1, pages 510–517 vol. 1, June 2005.
- [2] David Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60:91–, 11 2004.