Visión por Computador

Práctica 3:
Detección de puntos relevantes y Construcción de panoramas

Curso 2019-2020
Cuarto Curso del Grado en Ingeniería Informática

Laura Rabadán Ortega 79088745W
laurarabadan@correo.ugr.es

CONTENIDO

1. DETECCIÓN DE PUNTOS HARRIS

El primer ejercicio de está práctica consiste en detectar puntos estratégicos en una imagen dada por medio del algoritmo de Harris. El algoritmo no se implementa, sino que se utiliza una función de OpenCV.

Para este apartado se han utilizado las imágenes *Yosemite1.jpg* y *Yosemite2.jpg*. Por tanto, en primer lugar, se leen las imágenes en blanco y negro y se llama, con la primera imagen, a una función propia: **`Ejercicio1(path, umbral, sigma, incremento, niveles, radio)`**, dónde:

- **`Path`**: es el camino para llegar a la imagen que se quiere estudiar.
- **`Umbral`**: es un array de valores mínimos permitidos para los puntos máximos de cada escala.
- **`Sigma`**: sigma inicial de la pirámide Gaussiana.
- **`Incremento`**: incremento de la sigma de la pirámide Gaussiana.
- **`Niveles`**: es el número de niveles que se quiere que tenga la pirámide Gaussiana.
- **`Radio`**: es el número de vecinos que va a tener de distancia mínima un máximo.

Para las imágenes que se van a presentar, se ha creado 4 niveles de la pirámide Gaussiana, número de niveles suficientemente representativos para poder estudiar las diferencias entre ellos, utilizando en el primero un radio de 20 y en el resto de 10 píxeles, y umbrales de 0.001 (nivel inicial), 0.00045 (segundo novel) y 0.0001 (últimos dos niveles).

En esta función, se crea la pirámide Gaussiana y, para cada nivel conseguido, se calculan los puntos Harris y se pinta, con **`cv2.drawKeyPoints`** (importante incluir el *flag* **`cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS`** para que pinte los puntos a escala con la imagen y las direcciones de los gradientes) los puntos conseguidos.

Para calcular los puntos de cada nivel se implementa una función, **`ConseguirKeyPoints(imagen, umbral, nivel, radio)`** (los parámetros significan lo mismo, a excepción de **`imagen`** que en este caso es la matriz de píxeles, no el nombre). Los pasos que se realizan qui son:

1. Calcular los puntos Harris y, a partir de los valores conseguidos, $f = \frac{\lambda_1 \lambda_2}{\lambda_1 + \lambda_2}$.

```
Harris=cv2.cornerEigenValsAndVecs(img, blockSize=5, ksize=3)
# Se eligen esos valores para conseguir bastantes puntos, ya que al hacer la supresión de no máximos vamos a eliminar muchísimos.
Lamdas = Harris[:,:,:1] # De todos los valores que devuelve, nos quedamos con las λ.
```

2. Calcular las direcciones de los puntos. Para esto, se le pasa un filtro Gaussiano a la imagen con valor de sigma = 4.5. Una vez se tiene la convolución, se calcula la dirección de cada punto como $dirección = \frac{dy}{dx}$. Esos valores se pasan a grados para que la función **`cv2.drawKeyPoints`** los acepte.

3. Guardar el radio utilizado, la escala.

4. Hacer la supresión de no máximos. Para esto, se utiliza una ventana auxiliar que irá indicando si el siguiente punto podrá utilizarse o no. Para cada ventana, se elegirá, si se cumplen las condiciones, un punto como máximo.

```
alto, ancho = f.shape[:2]
puntos = np.empty((1,2), dtype=np.int)     # Puntos seleccionados

# Ventana que indica si hay que estudiar o no un punto como posible
máximo
verificador = np.full((alto, ancho), True, dtype=np.bool)

for i in range (0, alto - radio + 1):
   for j in range (0, ancho - radio + 1):

      # Si ese punto está permitido, se calcula la ventana que tiene
        ese punto en la esquina superior izquierda
      if verificador[i][j]:
         maximo = -1
         ind = 0
         encontrado = False

         # Se recorre la ventana
         for k in range (0, radio):
            for l in range (0, radio):

               # Si es un valor permitido y es el mayor de la ventana
                 y superior al umbral, se guarda
               if verificador[i+k][j+l] and (f[i+k][j+l] > maximo) and
                 (f[i+k][j+l] > umbral):

                  maximo = f[i+k][j+l]
                  ind = [i+k, j+l]
                  encontrado = True

                  # Todos los valores de la ventana se anulan, ya que
                    no podrán ser máximos
                  verificador[i+k][j+l] = False

         # En caso de que se encuentre algún punto válido, se guarda
           en la lista de puntos y se vuelve a activar el punto máximo
         if encontrado:
            verificador[ind[0]][ind[1]] = True
            puntos = np.vstack((puntos,np.array(ind)))
            num_keypoints += 1
```

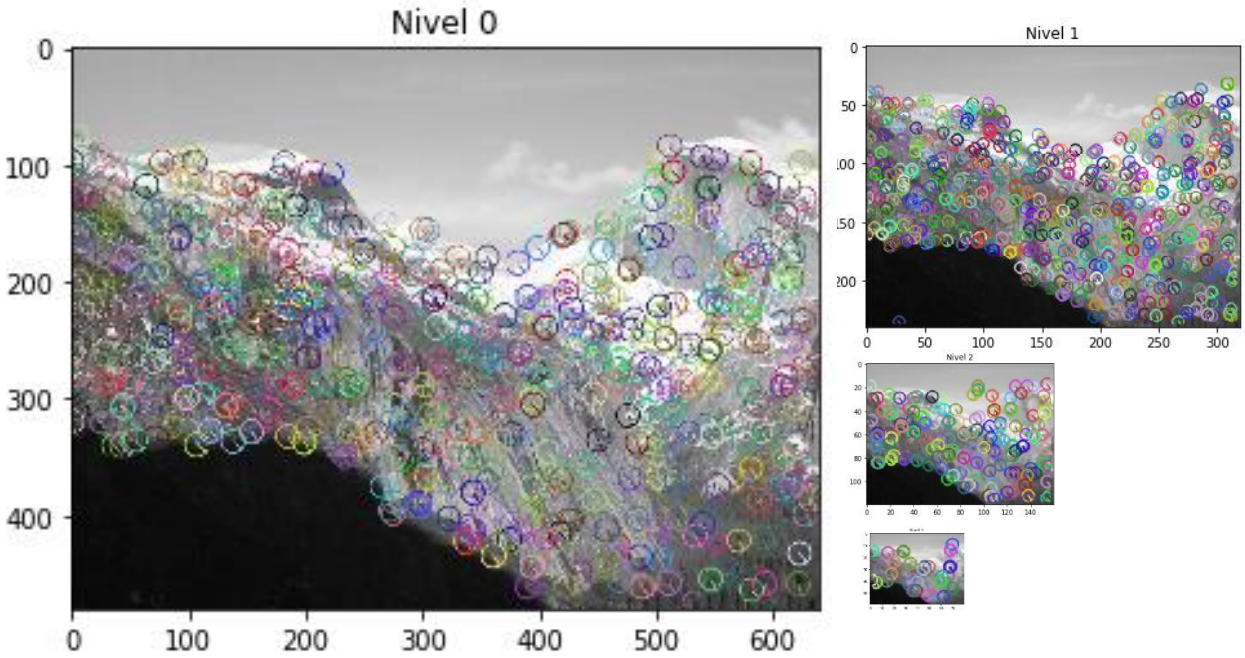
Con esto, ya tendríamos los puntos Harris detectados en nuestras imágenes. Los resultados conseguidos son los siguientes:

- Fichero *Yosemite1.jpg*:

```
Calculando puntos del nivel 1 ...
-> Puntos conseguidos:  920

Calculando puntos del nivel 2 ...
-> Puntos conseguidos:  885

Calculando puntos del nivel 3 ...
-> Puntos conseguidos:  235

Calculando puntos del nivel 4 ...
-> Puntos conseguidos:  52

Umbrales: [0.001, 0.00045, 0.0001, 0.001]

Puntos:  2092
```

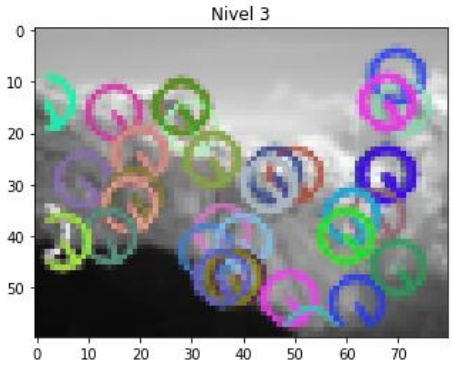
Lo primero que se observa, antes siquiera de ver las imágenes, es que cuanto más subamos en la pirámide, menos puntos se van a conseguir. Esto es coherente, ya que cuantos más niveles de la pirámide subamos, menos píxeles tenemos.

Esos valores de umbrales se obtuvieron en base a ejecuciones de prueba y error. Se fue pintando una a una las imágenes para conseguir un equilibrio entre la ventana y el umbral. Tras todas esas pruebas, se estimó que el mejor resultado, consiguiendo una gran cantidad de puntos, es la siguiente:



La más pequeña, nivel 4 de la pirámide, es la que tenemos debajo (página siguiente) con más zoom, para poder distinguir mejor los puntos conseguidos. La representación, efectivamente, muestra la cantidad de puntos conseguidos en los pasos anteriores. Al ver la representación podemos hacernos mejor una idea del por qué conseguimos más puntos en las primeras que en las últimas. En la imagen original, se tiene muchísimo detalle, por lo que conseguiremos muchos puntos, relacionados con todos esos detalles.

A medida que vamos subiendo en la pirámide, este detalle se va a ir reduciendo y, a su vez, el tamaño de la imagen. Por esta razón, el número de puntos será menor cuanto más alto sea el nivel de la pirámide.



A diferencia que en la primera imagen, en la que conseguimos 920 puntos, en la del último nivel de pirámide se consiguen 52. En esta es más fácil ver la representación de la dirección del gradiente.

Además del número de datos, podemos ver que en todas las imágenes los puntos se localizan en la misma región: la montaña blanca. Esto nos indica que esta zona será clave para futuras operaciones.

En el caso de los primeros niveles, los puntos están claramente dibujando el área de la montaña. En las otras dos tampoco nos quedamos atrás. Teniendo en cuenta cuanto se ha reducido la imagen, los puntos siguen prácticamente la misma tendencia que en las dos primeras.

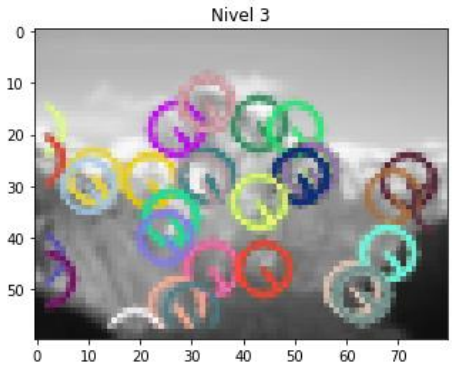
Por lo tanto, los puntos en todos los niveles van a ser, *grosso modo*, los mismos. No tienen que ser literalmente los mismos, las mismas coordenadas, pero si la misma zona local de la imagen.

- Fichero *Yosemite2.jpg*:

```
Calculando puntos del nivel 1 ...
-> Puntos conseguidos:  902

Calculando puntos del nivel 2 ...
-> Puntos conseguidos:  883

Calculando puntos del nivel 3 ...
-> Puntos conseguidos:  257

Calculando puntos del nivel 4 ...
-> Puntos conseguidos:  47

Umbrales: [0.001, 0.00045, 0.0001, 0.001]

Puntos:  2089
```

Con esta imagen comprobamos que los resultados anteriores son coherentes: cuanto más subimos en la pirámide, menos puntos tenemos.

Para este cálculo se han utilizado los mismos parámetros que en el caso anterior.

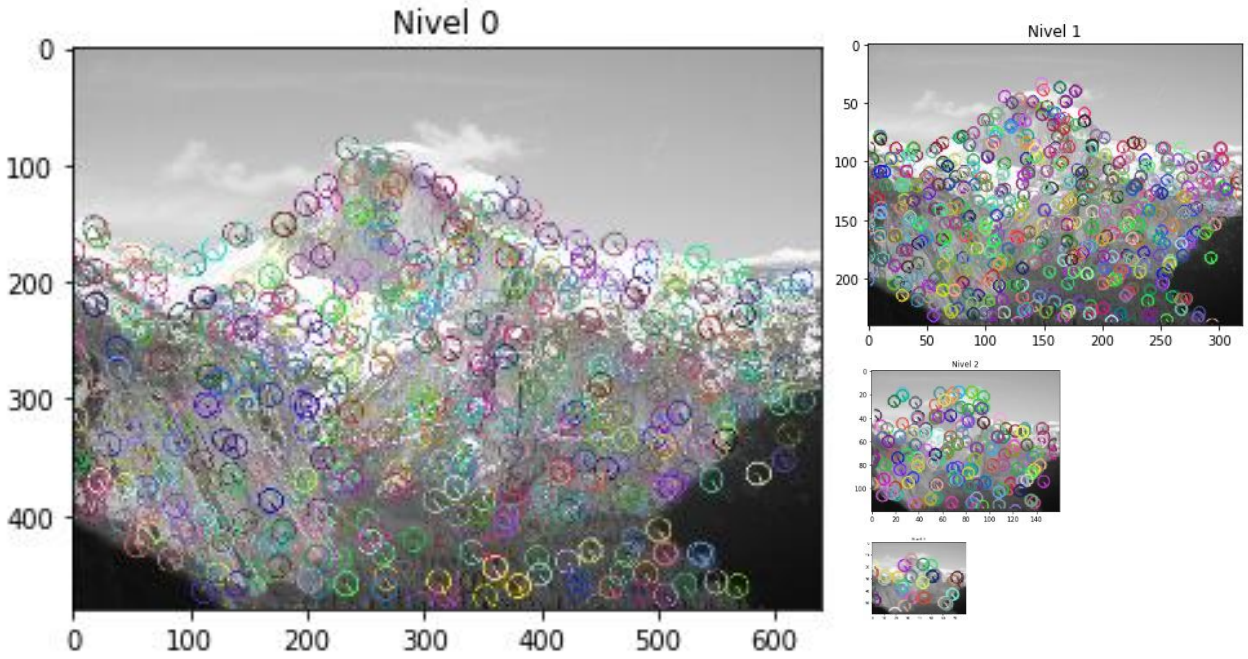


La representación gráfica que conseguimos es similar a la anterior. Todos los puntos se concentran en todos los niveles en la misma zona de la imagen: la montaña blanca. Además, como añadido a lo comentado con la imagen anterior, podemos ver que la orientación del gradiente es hacia la parte más oscura de la imagen. Es decir, la cumbre la montaña tiene mucha luz y la falda más sombra, la dirección del gradiente va de zonas con más luz a zonas con menos luz, comportamiento correcto.

La representación de todos los niveles de la pirámide está en la página siguiente.

Por tanto, una vez tenemos nuestros puntos y nuestra representación, vamos a calcular las coordenadas subpíxel, con ayuda de la función de OpenCV **`cv2.cornerSubPix`**, a la que se le pasa la imagen, las coordenadas de los puntos de la imagen que se tienen, el tamaño de ventana

que se va a utilizar (2, 2) y el criterio de parada (en nuestro caso, tras 100 iteraciones o al conseguir una distancia de 0.03).



Para sacar las coordenadas subpíxeles, se implementa una nueva función, **`CornerSubPixel`** que recibe la imagen y los puntos que se tienen. Esta función hace:

1. Consigue los nuevos puntos, adaptándolos primero al estilo que requiere la función.

```
criteria = (cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT, 100, 0.03)
coordenadas = np.empty((len(puntos),1,2), dtype=np.float32)

for i in range (0, len(puntos)):
   coordenadas[i][0][0] = puntos[i].pt[0]
   coordenadas[i][0][1] = puntos[i].pt[1]

cv2.cornerSubPix(img, coordenadas, winSize=(2,2), zeroZone=(-1,-1),
criteria=criteria)
```

2. Se seleccionan 3 puntos aleatorios, que son los que se van a representar.

```
ind = [random.randint(0, len(puntos)),random.randint(0, len(puntos)),
random.randint(0, len(puntos))]

nuevos_puntos = [puntos[ind[0]], puntos[ind[1]], puntos[ind[2]]]

# Este proceso se repite para los 3 puntos
keypoint = cv2.KeyPoint(coordenadas[ind[0]][0][0],
coordenadas[ind[0]][0][1], _size=puntos[ind[0]].size,
_angle=puntos[ind[0]].angle)
nuevas_coordenadas = [keypoint]
```

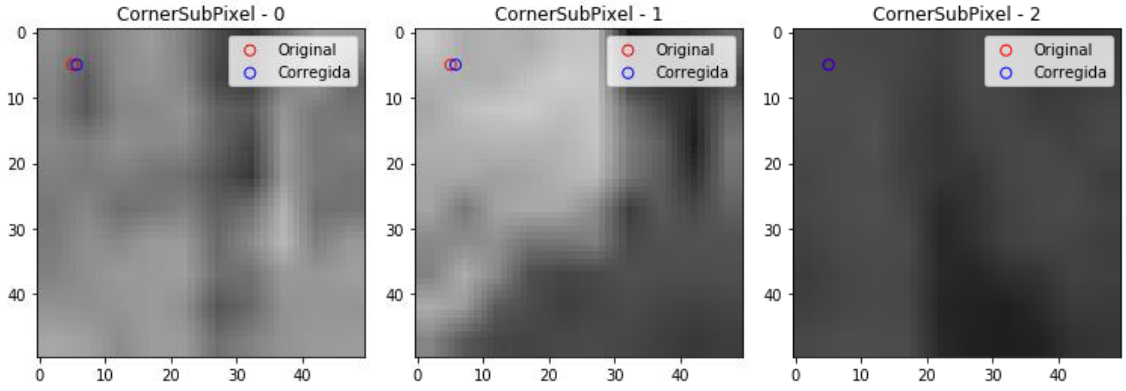
3. Modificamos el tamaño de la imagen en para quedarnos solo con la zona del punto.

```
for i in range (0, 3):
 x = int(nuevos_puntos[i].pt[1])
   y = int(nuevos_puntos[i].pt[0])

   x = [x-5, x+5]
   y = [y-5, y+5]

   if x[1] < 0:
      x[0] -= x[1]
      x[1] = 0

  if x[0] > img.shape[0]:
     x[1] -= (x[0] - img.shape[0])
     x[0] = img.shape[0]

 # Mismo procedimiento para los valores de y

 original = [nuevos_puntos[i].pt[1]-x[0], nuevos_puntos[i].pt[0]-y[0]]
 corregida = [nuevas_coordenadas[i].pt[1]-x[0],
nuevas_coordenadas[i].pt[0]-y[0]]

imagen = img[x[0]:x[1],y[0]:y[1]]
alto, largo = imagen.shape

imagen = cv2.resize(imagen, (alto*zoom, largo*zoom))
```

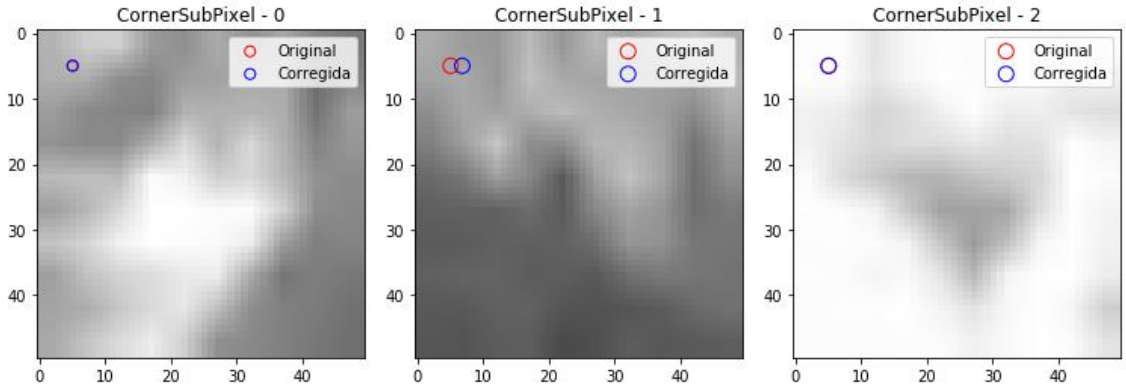
4. Por último, se pintan los dos puntos conseguidos.

Los resultados que se consiguen en este caso son los siguientes:

- Fichero *Yosemite1.jpg*:



- Fichero *Yosemite2.jpg*:



No en todos los casos hace falta corregir los puntos, como es el caso de las coordenadas 0 y 2 del segundo fichero. En cambio, en otros casos, como el de la coordenada 1, la diferencia entre la coordenada conseguida y la corregida con la coordenadas subpíxeles es grande. En este caso, el punto casi ni tienen área compartida.

Tras haber ejecutado el programa varias veces, se ha comprobado que la probabilidad de que un punto esté mal colocado es mayor que de que esté bien. Suele desplazarse casi siempre.

2. DETECCIÓN Y EXTRACCIÓN DE DESCRIPTORES AKAZE

En este segundo ejercicio se va a buscar una correlación entre puntos de interés, como los que se sacaron en el ejercicio anterior, de dos imágenes. Por supuesto, las dos imágenes tienen que estar relacionadas, teniendo elementos comunes entre ellas.

Por tanto, la solución de esta parte puede dividirse en 3 secciones distintas:

1. Conseguir los puntos y descriptores por medio de AKAZE.
2. Conseguir la correspondencia entre los descriptores de las dos imágenes.
3. Pintar los puntos conseguidos.

Para la primera parte, se utiliza una función de OpenCV, **`detectAndCompute`**, que extrae los puntos clave y los descriptores de las imágenes.

```
akaze = cv2.AKAZE_create()
keypoints1, descriptor1 = akaze.detectAndCompute(imagen1, None)
keypoints2, descriptor2 = akaze.detectAndCompute(imagen2, None)
```

Conseguir la correspondencia entre los descriptores es igual de fácil, pero esta vez tenemos que dividir dos casos distintos. Por un lado, vamos a utilizar la fuerza bruta, por lo que utilizamos:

```
matches = cv2.BFMatcher(cv2.NORM_L2, True)
matches.match(descriptor1, descriptor2)
```

En cambio, si queremos aplicar *Lowe's Ratio Distance,* tenemos que hacerlo de esta manera:

```
matches = cv2.BFMatcher(cv2.NORM_L2, False)
matches.knnMatch(descriptor1, descriptor2, k=2)
```

El método del primer caso nos da una lista de puntos posibles y en el segundo, una lista de parejas de puntos posibles, el primer y segundo mejor.

Si queremos las correspondencias por medio de fuerza bruta, solo quedaría pintar los resultados que conseguimos. En cambio, si queremos utilizamos la segunda forma, tenemos que aplicar la siguiente fórmula para filtrar los puntos interesantes.

$$\frac{d_1}{d_2} < umbral \Longrightarrow d_1 < umbral \times d_2$$

d_1 se corresponde con la distancia conseguida con el primer descriptor seleccionado, y d_2 la distancia conseguida con el segundo menor. El valor del umbral determinará la precisión que se va a tener en los puntos seleccionados: cuanto más bajo sea este umbral, más correctos se espera que sean los puntos correlacionados.

Para conseguir esto basta con hacer:

```
puntos = []
for k in matches:
   # Se utiliza la fórmula: (d1/d2) < umbral => d1 < umbral * d2
   k[0].distance < umbral * k[1].distance:
      puntos.append(k[0])
```

Consiguiendo en “puntos” la lista de correspondencias que han conseguido al pasar la poda.

Ya por último queda pintar las correspondencias conseguidas. Para ello se utiliza la función de OpenCV **cv2.drawMatches** que nos devuelve una imagen con la correspondencia representada.

```
# Flag = 2 -> NOT_DRAW_SINGLE_POINTS
imagen = cv2.drawMatches(imagen1, kp1, imagen2, kp2, matches,
outImg=np.array([]), flags=2)
```

Para probar la implementación anterior, se utilizan dos conjuntos de imágenes distintas: *Yosemite3.jpg* y *Yosemite4.jpg*, y *Tablero1.jpg* y *Tablero2.jpg*. A su vez, se prueban dos valores de umbral distintos, de 0.4 el primero y 0.7 el segundo. Se consigue, para el primer conjunto, el siguiente resultado:

Ilustración 1 - Fuerza Bruta

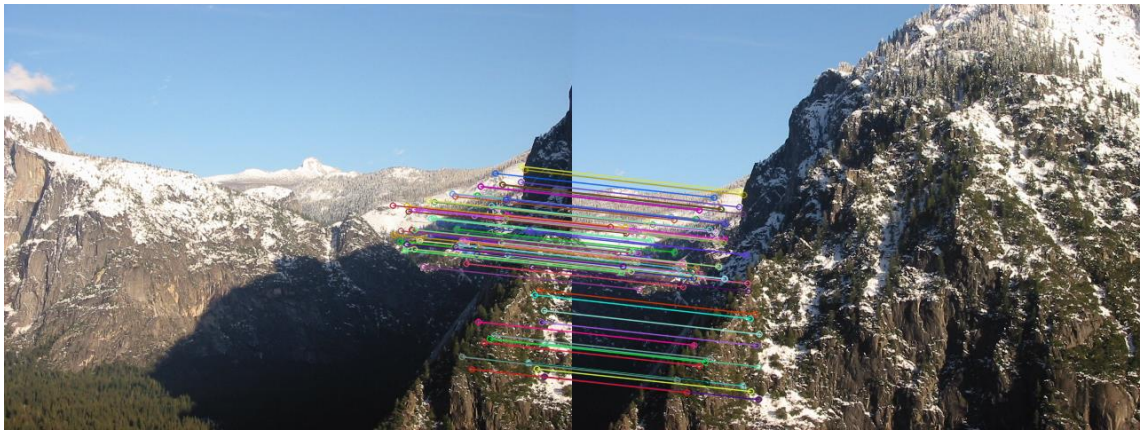


Ilustración 2 – Lowe-Average-2NN (Umbral 0.4)

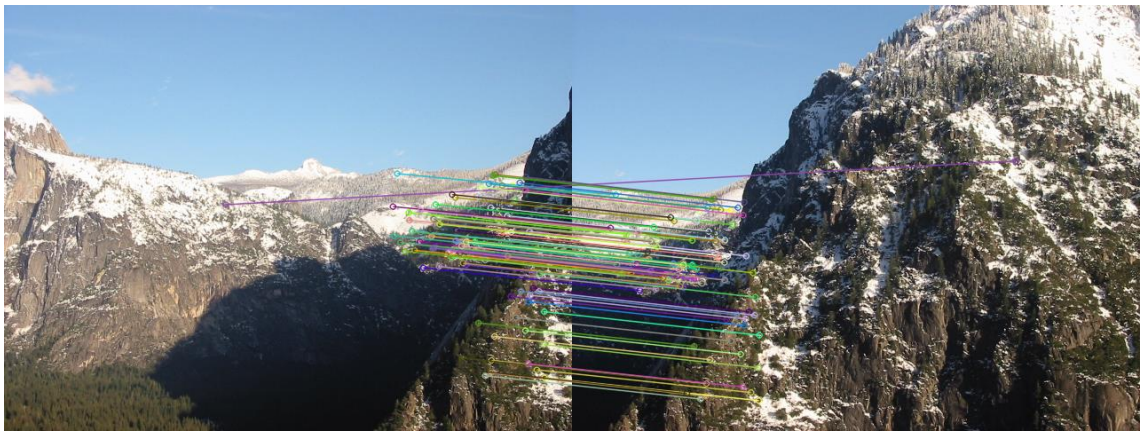


Ilustración 3 - Lowe-Average-2NN (Umbral 0.7)

Lo primero que destaca es el filtro de puntos que se consigue con respecto a la fuerza bruta. Con este método conseguimos correlaciones de objetos de la parte baja de a imagen con objetos de la parte alta, porque se intenta asociar cada punto con uno de la otra imagen con la que guarde una distancia mínima. Por eso, hay puntos repartidos por toda la imagen. Esta correlación es errónea, ya que muestra puntos incorrectos, por lo que usamos el segundo método para intentar mejorar el resultado.

Los puntos correlacionados que se consiguen en el segundo caso se centran una zona de cada imagen, coincidiendo con la parte de la escena que comparten. Los umbrales son importantes ya que, como se explicó, cuanto menor sea el umbral, más seguros se consideran esos puntos. Este efecto puede observarse en las imágenes anteriores: con umbral 0.4 todos los puntos parecen ser correcto, estudiándolos a ojo, pero con umbral 0.7, a parte de los puntos conseguidos con umbral 0.4, se consigue una nueva correlación incorrecta. Con respecto a l fuerza bruta se consiguen filtrar muchísimas correlaciones (se pasa de tener 425 correlaciones a tener 162), pero siguen escapándose algunos errores, como el de este caso.

Si ahora ejecutamos las imágenes de *Tablero1.jpg* y *Tablero2.jpg*, un conjunto más complicado porque se basa en una repetición geométrica constante, conseguimos los siguientes *matches*:

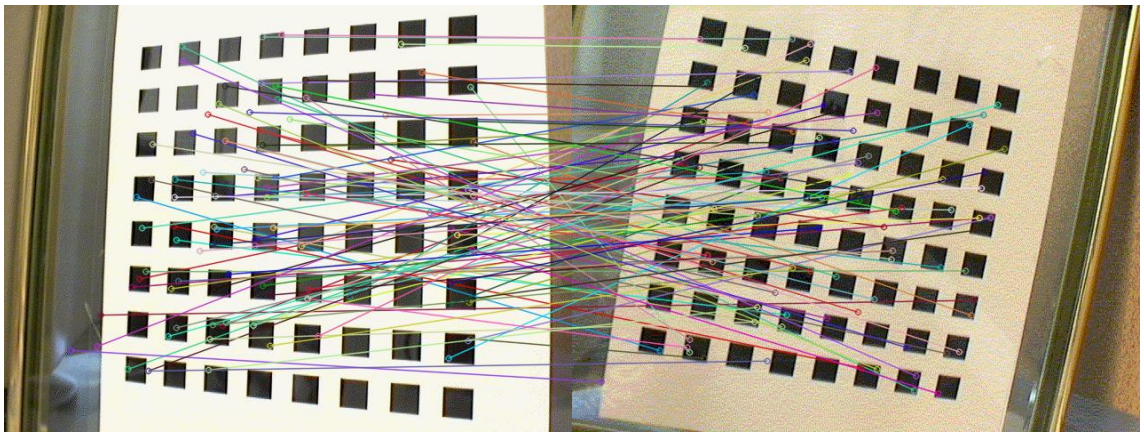


Ilustración 4 - Fuerza Bruta

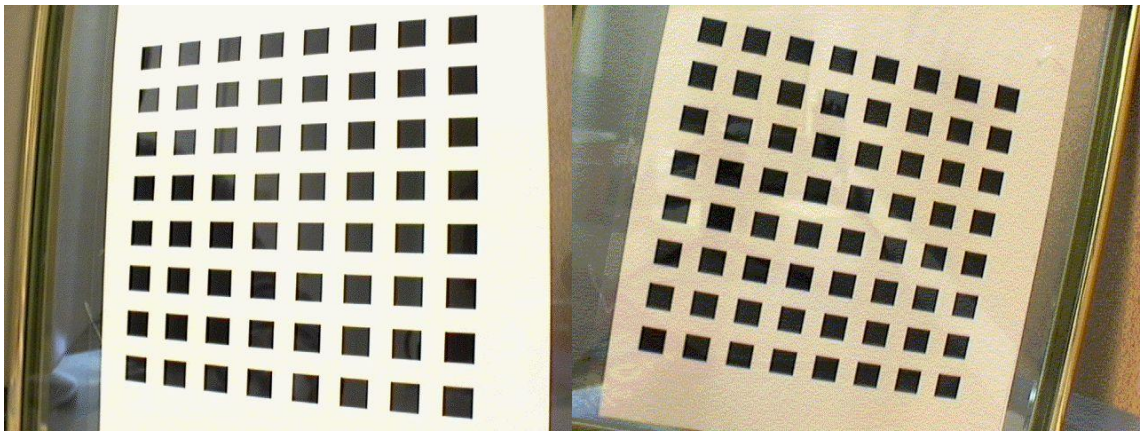


Ilustración 5 - Lowe-Average-2NN (Umbral 0.4)

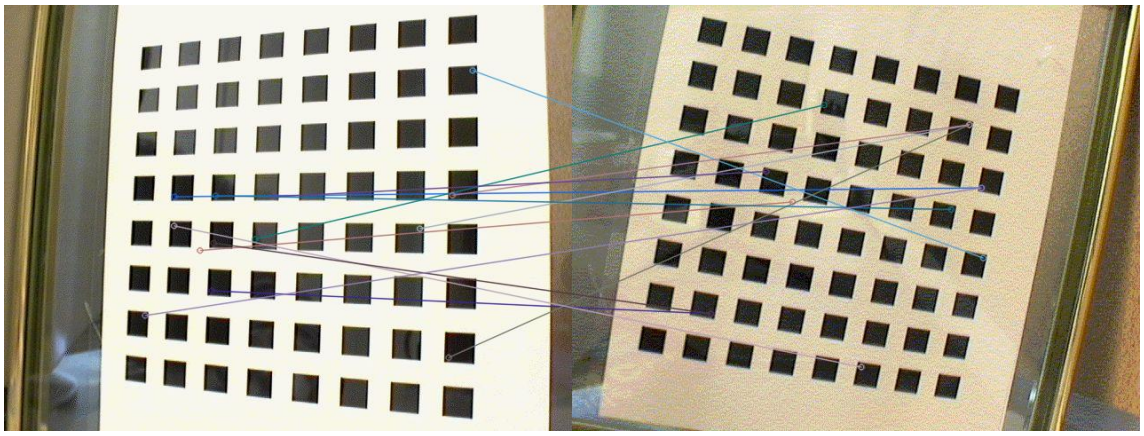


Ilustración 6 - Lowe-Average-2NN (Umbral 0.7)

En este caso, no parece que se haya conseguido ninguna correlación. Por eso, cuando aplicamos el umbral suave (0.7) son quedamos con muy pocos puntos, solo 13 de los 173 conseguidos con fuerza bruta, y al aplicar el umbral de 0.4 desaparecen todos. No se consigue ninguna correspondencia segura.

Si nos fijamos en las correspondencias conseguidas con el umbral 0.7, vemos que lo que se relacionan son esquinas con otras esquinas, pero no con la que le correspondería. Esto muestra que este método es muy bueno para algunos casos, como en el del primer conjunto con un umbral de 0.4, y muy malo para otros, en el que todos son esquinas, que al final es lo que se busca.

```
- Usando un umbral de  0.4

-> Numeros de matches con Fuerza Bruta:  425
-> Numero de matches con Lowe-Arange-2NN:  87

- Usando un umbral de  0.7

-> Numeros de matches con Fuerza Bruta:  425
-> Numero de matches con Lowe-Arange-2NN:  162
```

Ilustración 7 - Ejecución Yosemite

```
- Usando un umbral de  0.4

-> Numeros de matches con Fuerza Bruta:  173
-> Numero de matches con Lowe-Arange-2NN:  0

- Usando un umbral de  0.7

-> Numeros de matches con Fuerza Bruta:  173
-> Numero de matches con Lowe-Arange-2NN:  13
```

Ilustración 8 - Ejecución Tablero

3. GENERACIÓN DE MOSAICOS (N=2)

Para este ejercicio se utiliza lo que se consiguió en el ejercicio 2. Esta vez queremos combinar imágenes a partir de las correspondencias conseguidas, 2 en este ejercicio, de manera que formen una única imagen. Para conseguir nuestro objetivo utilizamos dos funciones claves de OpenCV: **`cv2.findHomography`**, para conseguir la homografía de las imágenes que queremos incorporar, y **`cv2.wrapPerspective`**, para montar el mosaico.

Por tanto, para conseguir la imagen final, lo primero que tenemos que hacer es montarnos una imagen de base ficticia. Ficticia en el sentido de que es una matriz que nos montamos y en la que "pegaremos" todas las imágenes del mosaico. La matriz se inicializa toda a 0 y su dimensiones son `(imagen1.shape[0]*4, int(imagen1.shape[1]*1.5)`, donde `imagen1` es la primera imagen que se va a montar.

Una vez tenemos nuestro fondo del mosaico, pasamos a copiar la primera foto en él. Para ello, sacamos los `keyPoints` de la primera imagen y creamos dos listas de puntos: una para la imagen, que tendrá las coordenadas de sus `keyPoints`, y otra para la matriz, que tendrá los mismos puntos que los de la imagen, pero con un incremento, para centrar la imagen en el fondo. Para este paso se utilizan las funciones creadas para el ejercicio anterior.

```
x = int(mosaico_base.shape[0]/3)
y = int(mosaico_base.shape[1]/3)

keypoints1,descriptor1,keypoints2,descriptor2 = Akaze(imagen1,mosaico_base)

# Puntos para montar la primera imagen en el mosaico
puntos1 = np.empty((len(keypoints1),2), dtype=np.float32)
puntos2 = np.empty((len(keypoints1),2), dtype=np.float32)

# Se añade la primera imagen en el mosaico
for i in range (0, x):
   puntos1[i,0] = keypoints1[i].pt[0]
   puntos1[i,1] = keypoints1[i].pt[1]

   puntos2[i,0] = keypoints1[i].pt[0] + x
   puntos2[i,1] = keypoints1[i].pt[1] + y

# Adaptamos las listas de puntos al formato que admite fidHomografy
puntos1.reshape(-1,1,2)
puntos2.reshape(-1,1,2)
```

Una vez tenemos la lista de interés, sacamos su homografía y montamos el mosaico.

```
# Encontramos la homografía entre las dos imágenes
homografias, mask = cv2.findHomography(puntos1, puntos2, cv2.RANSAC)

# Montamos la primera imagen del mosaico
mosaico = cv2.warpPerspective(src=imagen1,dst=mosaico_base,M=homografias,
                    dsize=(mosaico_base.shape[0], mosaico_base.shape[1]),
                    borderMode=cv2.BORDER_TRANSPARENT)
```

Hay que tener en cuenta, a la hora de usar la función **cv2.findHomography** que primero se ponen los puntos de la imagen que se va a modificar y después los de la imagen de referencia.

Con esto conseguimos montar la imagen central sobre el fondo, por lo que solo queda acoplar la segunda imagen al mosaico. El procedimiento es similar al anterior, pero en este caso, primero tenemos que buscar los puntos con correspondencia entre la imagen que queremos incorporar y el propio mosaico. De esta manera, al calcular la homografía asociada a la imagen será en función a la posición que tiene que ocupar en el mosaico.

Basándonos en los resultados anteriores, utilizamos el método *Lowe-Average-2NN* con un umbral de 0.4 para la correspondencia, consiguiendo puntos más fiables.

```
# Sacamos la correspondencia de la imagen que queremos añadir y el mosaico
actual
keypoints1, descriptor1, keypoints2, descriptor2 = Akaze(mosaico, imagen2)
matches = Correspondecia (descriptor1, descriptor2, False)

# Puntos seleccionados por medio de Lowe's ratio distance
matches = LoweRatio (matches, 0.4)

# Puntos para encontrar la homografía
puntos1 = np.empty((len(matches),2), dtype=np.float32)
puntos2 = np.empty((len(matches),2), dtype=np.float32)

for i in range(0, len(matches)):
   puntos1[i,0] = keypoints2[matches[i].trainIdx].pt[0]
   puntos1[i,1] = keypoints2[matches[i].trainIdx].pt[1]

   puntos2[i,0] = keypoints1[matches[i].queryIdx].pt[0]
   puntos2[i,1] = keypoints1[matches[i].queryIdx].pt[1]
```

El resto del procedimiento sería el mismo.

Este ejercicio lo hemos probado con dos pares de fotos distintos, dos del mosaico *Yosemite* y dos del mosaico *mosaico*. En concreto, para este último, se han utilizado las imágenes 002 y 006 para que se notara la diferencia al montarlas.

- *Yosemite:*

Con esta imagen, en la parte del cielo se distingue la línea que separa las dos imágenes. En cambio, en la parte de la montaña es mucho más difícil saber dónde se juntan las dos imágenes.

- *Mosaico:*

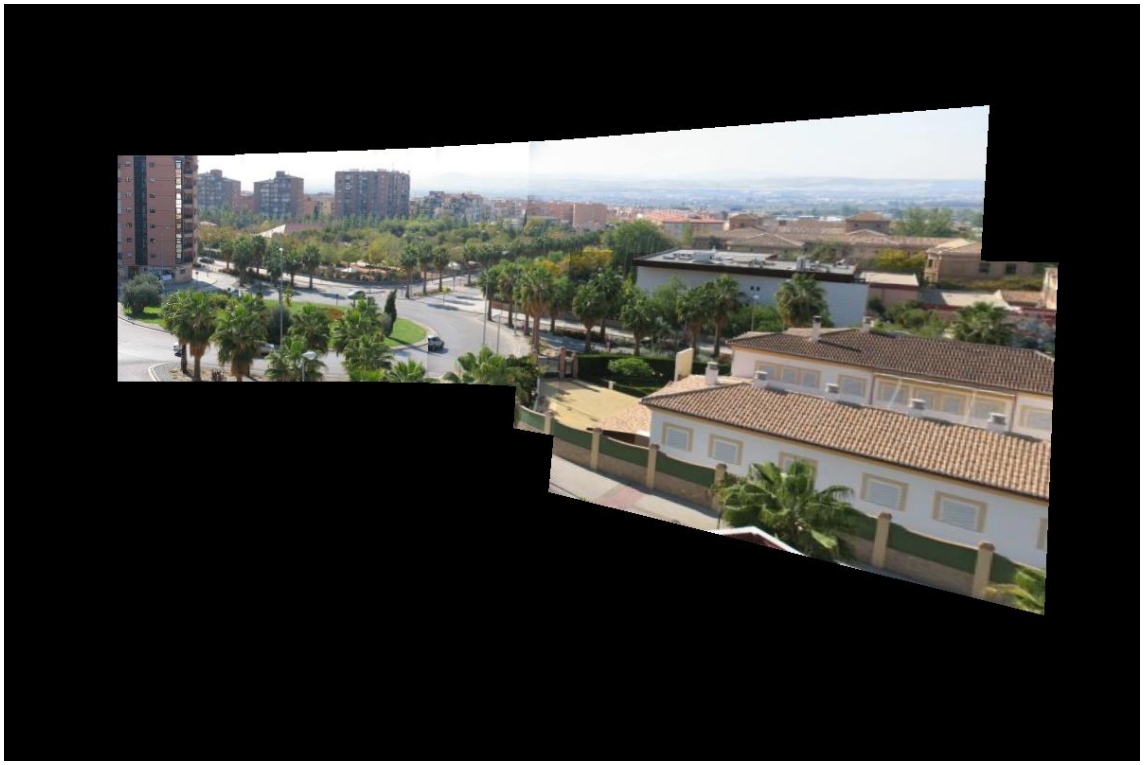
En este caso es más difícil descubrir donde está el corte de la imagen.

A partir de las imágenes anteriores, podría decirse que cuanto más plana y lisa sea una imagen, más refinamiento debe tener la técnica para conseguir un acoplamiento prácticamente perfecto.

4. GENERACIÓN DE MOSAICOS (N=10)

Este ejercicio es una ampliación del anterior. Para realizarlo se implementa exactamente el ejercicio anterior pero dentro de un bucle, añadiendo todas las imágenes al mosaico. La única diferente es la primera, por la manera en la que se tiene que añadir al mosaico.

Para crear el mosaico se utilizan las 10 imágenes de *mosaico*, iterando por todas ellas hasta conseguir el siguiente resultado.



Como pasaba en el ejercicio anterior, dónde más se nota las imágenes del centro. En las imágenes del final y las del principio, a penas se nota el paso de una imagen a otra.

Otro impedimento son los objetos móviles, ya que no se mantienen constante en todas las imágenes. Eso puede verse en el coche, que se distorsiona al crear el mosaico.

5. REFERENCIAS

AKAZE local features matching. (2014, Noviembre 10). Retrieved from OpenCV 3.0.0-dev documentation: https://docs.opencv.org/3.0-beta/doc/tutorials/features2d/akaze_matching/akaze_matching.html

cv::BFMatcher Class Reference. (2019, Diciembre 21). Retrieved from OpenCV: https://docs.opencv.org/3.4/d3/da1/classcv_1_1BFMatcher.html

cv::DescriptorMatcher Class Reference. (2019, Julio 26). Retrieved from OpenCV: https://docs.opencv.org/4.1.1/db/d39/classcv_1_1DescriptorMatcher.html#a378f35c9b1a5dfa4022839a45cdf0e89

cv::DMatch Class Reference. (2019, Diciembre 21). Retrieved from OpenCV: https://docs.opencv.org/3.4/d4/de0/classcv_1_1DMatch.html#a47475576327f6ba9cd39250881b4d330

cv::Feature2D Class Reference. (2019, Diciembre 21). Retrieved from OpenCV: https://docs.opencv.org/3.4/d0/d13/classcv_1_1Feature2D.html#a8be0d1c20b08eb867184b8d74c15a677

cv::KeyPoint Class Reference. (2019, Diciembre 21). Retrieved from OpenCV: https://docs.opencv.org/3.4/d2/d29/classcv_1_1KeyPoint.html#acfcc8e0dd1a634a7583686e18d372237

DataCamp. (n.d.). *Python For Data Science Cheat Sheet Numpy*. Retrieved from DataCamp: https://s3.amazonaws.com/assets.datacamp.com/blog_assets/Numpy_Python_Cheat_Sheet.pdf

Drawing Function of Keypoints and Matches. (2019, Julio 26). Retrieved from OpenCV: https://docs.opencv.org/4.1.1/d4/d5d/group__features2d__draw.html#ga2c2ede79cd5141534ae70a3fd9f324c8

Drawing Function of Keypoints and Matches. (2019, Diciembre 21). Retrieved from OpenCV: https://docs.opencv.org/3.4/d4/d5d/group__features2d__draw.html

Feature Detection. (2019, Diciembre 19). Retrieved from OpenCV: https://docs.opencv.org/2.4/modules/imgproc/doc/feature_detection.html?highlight=cornersubpix

Feature Matching. (2019, Diciembre 21). Retrieved from OpenCV: https://docs.opencv.org/3.4/dc/dc3/tutorial_py_matcher.html

Features2D + Homography to find a known object. (2019, Diciembre 22). Retrieved from OpenCV: https://docs.opencv.org/2.4/doc/tutorials/features2d/feature_homography/feature_homography.html

Geometric Transformations of Images. (2019, Diciembre 22). Retrieved from OpenCV: https://docs.opencv.org/trunk/da/d6e/tutorial_py_geometric_transformations.html

Mallick, S. (2016, Enero 3). *Homography Examples using OpenCV (Python / C ++)*. Retrieved from Learn OpenCV: https://www.learnopencv.com/homography-examples-using-opencv-python-c/