

# **Práctica-3: Detección de puntos relevantes y Construcción de panoramas**

## **Visión por Computador.**

**UNIVERSIDAD DE GRANADA  
E.T.S.I. INFORMÁTICA Y TELECOMUNICACIÓN**



**UNIVERSIDAD  
DE GRANADA**

**Departamento de Ciencias de la  
Computación e Inteligencia Artificial**

**Grado en Ingeniería Informática.  
Curso 2019-2020.**

*Daniel Bolaños Martínez  
76592621-E  
[danibolanos@correo.ugr.es](mailto:danibolanos@correo.ugr.es)  
Grupo 2 - Jueves 11:30h*

# Índice

<b>1. Detectar puntos Harris sobre una pirámide Gaussiana de la imagen y presentar dichos puntos haciendo uso de la función drawKeyPoints.</b>	<b>2</b>
1.1. Detectar los puntos Harris en cada nivel de la pirámide a partir de la información aportada por cornerEigenValsAndVecs() . . . . .	2
1.2. Variar los valores de umbral de la función de detección de puntos hasta obtener un conjunto numeroso de puntos HARRIS y justificar la elección de los parámetros. . . . .	6
1.3. Identificar cuantos puntos se han detectado dentro de cada octava. . . . .	10
1.4. Calcular las coordenadas subpixel de cada KeyPoint usando la función cornerSubPix de OpenCV y mostrar 3 submatrices con el refinamiento obtenido. . . . .	14
<b>2. Detectar y extraer los descriptores AKAZE de OpenCV. Establecer las correspondencias existentes entre cada dos imágenes y usar los criterios “BruteForce+crossCheck” y “Lowe-Average-2NN”.</b>	<b>16</b>
2.1. Valorar la calidad de los resultados obtenidos a partir de un par de ejemplos aleatorios de 100 correspondencias. . . . .	19
<b>3. Escribir una función que genere un Mosaico de calidad a partir de N = 2 imágenes relacionadas por homografías.</b>	<b>21</b>
3.1. Definir una imagen en la que pintaremos el mosaico. . . . .	21
3.2. Definir la homografía que lleva cada una de las imágenes a la imagen del mosaico. . . . .	22
3.3. Usar la función cv2.warpPerspective() para trasladar cada imagen al mosaico. . . . .	23
<b>4. Realizar un mosaico usando todas las imágenes proporcionadas.</b>	<b>25</b>
<b>5. Referencias</b>	<b>27</b>

# 1. Detectar puntos Harris sobre una pirámide Gaussiana de la imagen y presentar dichos puntos haciendo uso de la función drawKeyPoints.

Para la realización de este apartado, hemos hecho uso de algunas funciones implementadas en la práctica 1 y que definiremos a continuación:

- **leeImagen**: Lee una imagen con el flag del color especificado. 1 para color y 0 para blanco y negro.
- **normalizaRGB**: Normaliza una matriz de flotantes (en escala de grises o a color) y la escala al rango 0-255.
- **pintaI**: Normaliza la imagen a escala RGB, la muestra por pantalla y le pone un título.
- **funcionGaussiana**: calcula la convolución de una imagen con una Gaussiana 2D haciendo uso de la función **getGaussianKernel** de *OpenCV*.
- **convMasks1D**: calcula la convolución de una imagen usando máscaras de derivadas haciendo uso de la función **getDerivKernels** de *OpenCV*.

## 1.1. Detectar los puntos Harris en cada nivel de la pirámide a partir de la información aportada por **cornerEigenValsAndVecs()**.

La cabecera de la función utilizada para realizar este apartado es la siguiente:

```
def ejercicio1A(im, blockSize, threshold, winsize, ksizeSobel  
=3, ksizeDeriv=3, niveles=4):
```

Donde cada parámetro viene definido como:

- **im**: imagen en escala de grises a la que se van a calcular los puntos Harris para cada nivel de la pirámide.

- **blockSize**: tamaño de la ventana de entorno para la función `cornerEigenValsAndVecs`.
- **threshold**: valor del umbral a partir del cual se suprime ese píxel de la matriz.
- **winsize**: tamaño de la ventana de búsqueda del máximo en la supresión de no máximos.
- **ksizeSobel**: tamaño de la máscara de Sobel en la función `cornerEigenValsAndVecs`. Por defecto, toma el valor de 3.
- **ksizeDeriv**: tamaño de la máscara para las derivadas en la función `cornerEigenValsAndVecs`. Por defecto, toma el valor de 3.
- **niveles**: niveles de la pirámide gaussiana. Por defecto, toma el valor de 4.

Para detectar los puntos Harris sobre cada nivel de la pirámide de una imagen, procederemos de la siguiente forma:

- Convertimos los valores de la matriz de la imagen de **uint8** a **float32**.
- Generamos la pirámide Gaussiana de la imagen haciendo uso de la función **pirDown** de *Open CV* a la que le pasamos como parámetro una imagen y nos devuelve el siguiente nivel de la pirámide a la que aplica una reducción de tamaño y un suavizado Gaussiano.

Obtendremos un vector con tantas imágenes como **niveles** hayamos establecido como parámetro.

- Para cada nivel de la pirámide, calculamos los valores y vectores propios usando la función **cornerEigenValsAndVecs** de *Open CV*.

Esta función, recibe como parámetros una **imagen** (que será en cada iteración del bucle, un nivel de la pirámide), **blockSize** y **ksizeSobel**. Para cada píxel de la imagen, la función considera un entorno  $S(p)$  de tamaño (**blockSize** × **blockSize**) y calcula la matriz de covarianza de las derivadas sobre el entorno  $S(p)$  como:

$$M = \begin{bmatrix} \sum_{S(p)} \left( \frac{dI}{dx} \right)^2 & \sum_{S(p)} \frac{dI}{dx} \cdot \frac{dI}{dy} \\ \sum_{S(p)} \frac{dI}{dx} \cdot \frac{dI}{dy} & \sum_{S(p)} \left( \frac{dI}{dy} \right)^2 \end{bmatrix}$$

Donde las derivadas se calculan usando el operador de Sobel con tamaño de máscara **ksizeSobel**. La función devuelve los valores y vectores propios de  $M$ .

- Nos quedamos con las matrices que contienen los valores propios  $\lambda_1$  y  $\lambda_2$  para cada píxel de la imagen.
- Para cada nivel de la imagen crearemos una matriz (**fim**) que contiene para cada píxel  $p$ , el valor de  $\frac{\lambda_{1p} \cdot \lambda_{2p}}{\lambda_{1p} + \lambda_{2p}}$ , si  $\lambda_{1p} + \lambda_{2p} = 0$  el valor en ese píxel será 0.
- Ponemos a 0 los valores de los píxeles de **fim** menores al umbral **threshold**.
- Aplicamos la supresión de no máximos sobre la matriz resultante, teniendo en cuenta el tamaño de entorno (**winsize**) en el cálculo de los máximos de la nueva imagen.

La función **supresionNoMax** recibe como parámetros la matriz **fim** y el valor de **winsize** y para cada píxel calcula los valores de su entorno de vecinos en una matriz de tamaño (**winsize**  $\times$  **winsize**). Si el valor del píxel central no es máximo del entorno, se pone a 0 en una copia de la matriz original.

Para calcular los **KeyPoints** en cada escala, deberemos especificar: la posición de los píxeles que han resultado después de la supresión de no máximos, el tamaño (que vendrá determinado por el nivel de la pirámide y **blockSize**) y la orientación, que calcularemos de la siguiente manera:

- Calculo las derivadas en x e y de la imagen original haciendo uso de la función **convMasks1D** de la práctica 1. Para ello especificaremos los siguientes parámetros:

```
dxIm = convMasks1D(im, dx=1, dy=0, ksize=ksizeDeriv)
dyIm = convMasks1D(im, dx=0, dy=1, ksize=ksizeDeriv)
```

- Calculo el suavizado Gaussiano sobre las derivadas haciendo uso de la función **funcionGaussiana** de la práctica 1. Usamos  $\sigma = 4,5$  y calculamos el valor de **ksize** ( $2\lceil 3\sigma \rceil + 1$ ) a partir del sigma como ya especificamos en la práctica inicial.

```
dxIms = funcionGaussiana(dxIm, sigmaX=4.5)
dyIms = funcionGaussiana(dyIm, sigmaX=4.5)
```

- Calculo la pirámide Gaussiana de la imagen **dxIms** y **dyIms** para los **niveles** especificados.
- Almacenamos un vector de **KeyPoints** para cada nivel de la pirámide donde para cada píxel mayor que 0, calculamos su posición relativa a la matriz original ( $x \cdot 2^{nivel}, y \cdot 2^{nivel}$ ), su tamaño de escala (**blockSize** · nivel) y su ángulo de orientación  $\arctan(\frac{\sin \theta}{\cos \theta})$  en grados.

Es importante usar una función arco tangente que devuelva los valores en los cuatro cuadrantes (usaremos **atan2**) y pasar a grados el ángulo resultante.

Los valores de coseno y seno utilizados para calcular el ángulo  $\theta$  vendrán determinados por los valores obtenidos en las pirámides de **dxIms** y **dyIms**. Para cada nivel **n** y píxel **p**:

$$\cos \theta = \frac{dxIms_{n,p}}{\sqrt{dxIms_{n,p}^2 + dyIms_{n,p}^2}}, \quad \sin \theta = \frac{dyIms_{n,p}}{\sqrt{dxIms_{n,p}^2 + dyIms_{n,p}^2}}$$

El valor de  $\theta$  es devuelto por la función **atan2** en radianes, por lo que debemos pasarlo a grados (multiplicando por  $360$  y diviendo por  $2\pi$ ) y asegurarnos que el valor de  $\theta$  en grados sea positivo (sumando  $360^\circ$  si el ángulo es negativo).

- Finalmente los **KeyPoints** vendrán definidos como:

```
cv2.KeyPoint(y*2**n, x*2**n, _size=blockSize*(n+1), _angle=theta)
```

Se cambian las posiciones de x e y por la implementación que usa *Open CV*.

La función **ejercicio1A** devuelve un vector de vectores de **KeyPoints** donde cada vector incluye los **KeyPoints** para cada escala de la pirámide, de esta forma será más fácil hacer el recuento de puntos para cada octava.

## 1.2. Variar los valores de umbral de la función de detección de puntos hasta obtener un conjunto numeroso de puntos HARRIS y justificar la elección de los parámetros.

La cabecera de la función utilizada para realizar este apartado es la siguiente:

```
def ejercicio1B(im, imC, blockSize, threshold, winsize):
```

Donde cada parámetro viene definido como:

- **im**: imagen en escala de grises a la que se van a calcular los puntos Harris para cada nivel de la pirámide.
- **imC**: imagen en color sobre la que se dibujarán los **KeyPoints** haciendo uso de la función **drawKeypoints** de *Open CV*.
- **blockSize**: tamaño de la ventana de entorno para la función **cornerEigenValsAndVecs**.
- **threshold**: valor del umbral a partir del cual se suprime ese píxel de la matriz.
- **winsize**: tamaño de la ventana de búsqueda del máximo en la supresión de no máximos.

Para este apartado dibujaremos todos los **KeyPoints** sobre la misma imagen usando color rojo y veremos como afectan las variaciones de los parámetros sobre el número de **KeyPoints** obtenidos en cada caso.

Para dibujar los **KeyPoints** hacemos uso de la siguiente función:

```
for i in range(len(v_kp)):  
    cv2.drawKeypoints(imNew, v_kp[i], outImage=np.array([]), color  
                      =(0, 0, 255), flags=4)
```

Dibujaremos en color rojo sobre la imagen *imNew* los **KeyPoints** de todas las escalas. El flag 4 (DRAW\_RICH\_KEYPOINTS) indica que para cada

**KeyPoint**, el círculo dibujado a su alrededor tendrá el tamaño y la orientación que se hayan especificado en el constructor.

Las pruebas las haremos sobre la imagen Yosemite1.jpg, calcularemos los **KeyPoints** sobre la imagen en blanco y negro y los dibujaremos sobre la imagen a color. A continuación mostraremos las pruebas realizadas con la modificación de parámetros y extraeremos conclusiones sobre cómo afecta cada parámetro al número de **KeyPoints** obtenidos.

1. ksizeSobel=3; ksizeDeriv=3; niveles=4				
Imagen	blockSize	threshold	winsize	KeyPoints
1.1	7	5	5	2404
1.2	7	50	5	2133
1.3	3	5	5	4963
1.4	5	5	5	3060
1.5	5	5	3	6873
1.6	5	5	7	2056

Tabla 1: Tabla comparativa parámetros.

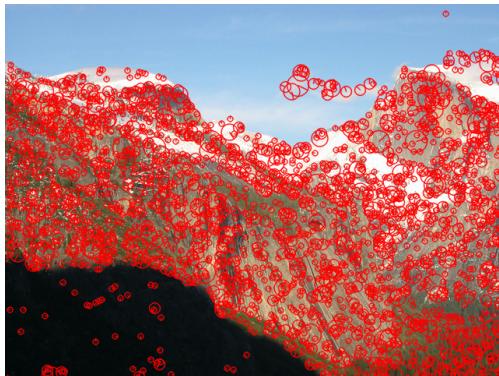


Figura 1: Imagen 1.1.

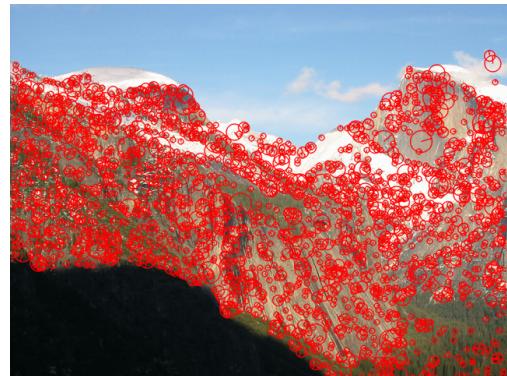


Figura 2: Imagen 1.2.

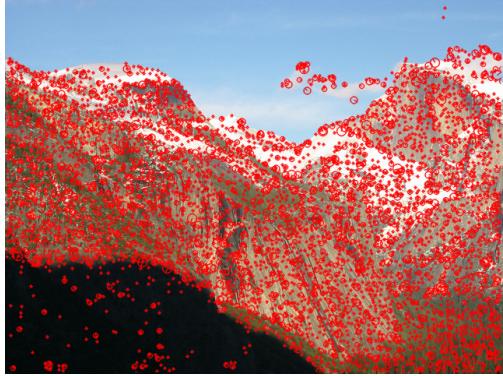


Figura 3: Imagen 1.3.

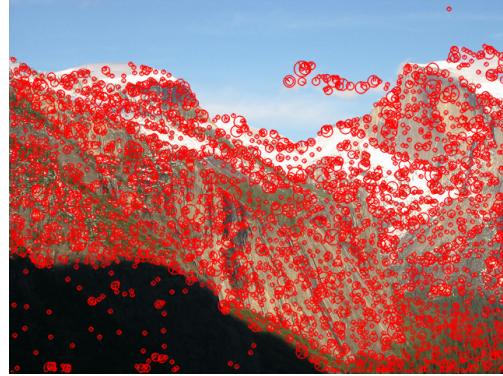


Figura 4: Imagen 1.4.

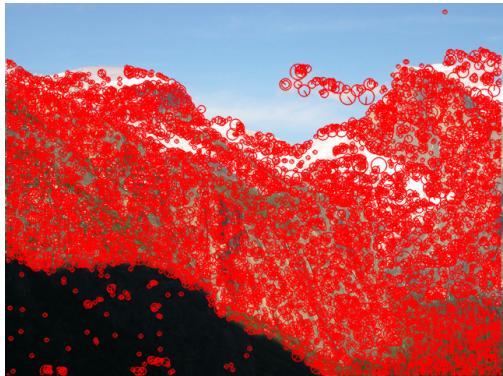


Figura 5: Imagen 1.5.

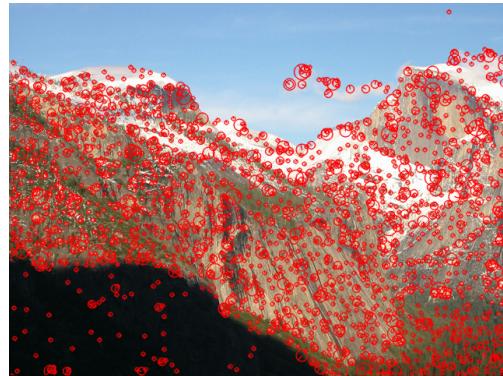


Figura 6: Imagen 1.6.

Dejando fijos los parámetros **niveles**, **ksizeSobel** y **ksizeDeriv** y modificando el resto, podemos extraer las siguientes conclusiones:

- Aumentar el umbral **threshold** disminuye como es obvio la cantidad de puntos obtenidos, ya que serán menos puntos los que pasen la criba.
- Aumentar el valor de **blockSize** también disminuye el número de puntos. Podemos ver como, sean **threshold** y **winsize** fijos, variando el tamaño de bloque en el que se calculan los valores propios de 3 a 7, obtenemos una gran diferencia entre las tres versiones.

- Aumentar el valor de **winsize** disminuye la cantidad de puntos, esto es evidente ya que si hacemos la supresión de no máximos sobre un entorno del píxel mayor, habrá más probabilidad de que el píxel candidato a ser suprimido, no sea máximo de su entorno y por tanto será eliminado.

Podemos observar como la elección del valor del umbral es clave a la hora de descartar puntos que no pertenecen a los bordes de la imagen, ya que con el umbral de 50 obtenemos mejores resultados que con el de 5. Además, usaremos valores de **blockSize** y **winsize** medio-bajos, ya que con valores altos, descartamos muchos puntos.

A continuación fijaremos el valor de estos tres parámetros y realizaremos algunos experimentos modificando los valores de **niveles**, **ksizeSobel** y **ksizeDeriv**.

2. blockSize=3; threshold=50; winsize=5				
Imagen	ksizeSobel	ksizeDeriv	niveles	KeyPoints
2.1	3	3	4	4301
2.2	3	7	4	4301
2.3	5	3	4	4946
2.4	3	3	6	4329

Tabla 2: Tabla comparativa parámetros.

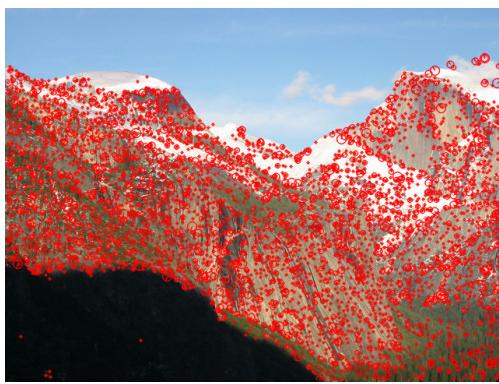


Figura 7: Imagen 2.1.

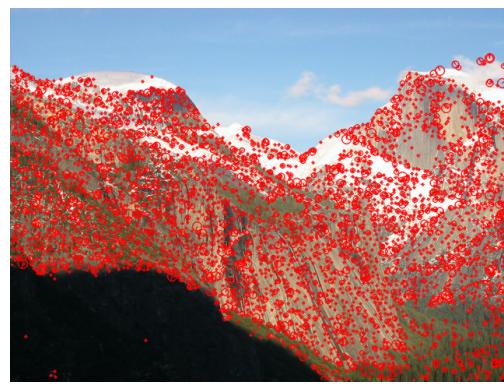


Figura 8: Imagen 2.2.

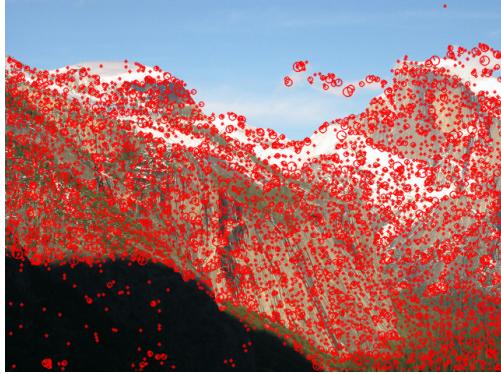


Figura 9: Imagen 2.3.

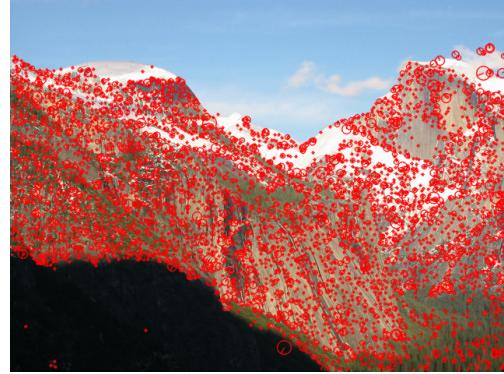


Figura 10: Imagen 2.4.

Si dejamos fijos los parámetros **blockSize**, **threshold** y **winsize** y modificamos el resto, podemos extraer las siguientes conclusiones:

- Aumentar el valor de **ksizeSobel** aumenta el número de puntos, pero también empeora la precisión de los mismos.
- Aumentar el valor de **ksizeDeriv** deja invariantes los puntos, ya que este parámetro solo afecta al cálculo de la orientación.
- Aumentar los **niveles** aumenta el número de puntos, lo que es razonable ya que al haber más escalas, también aumentarán los **KeyPoints**, aunque por cada nivel añadido, el número de puntos que se sumen con cada nivel se irá reduciendo.

### 1.3. Identificar cuantos puntos se han detectado dentro de cada octava.

La cabecera de la función utilizada para realizar este apartado es la siguiente:

```
def ejercicio1C(im, kpoints):
```

Donde cada parámetro viene definido como:

- **im**: imagen sobre la que se van a dibujar los puntos Harris para cada nivel de la pirámide.

- **kpoints**: conjunto de **KeyPoints** obtenidos en el apartado A con los valores de los parámetros que se hayan especificado.

Para este apartado, mostraremos los puntos que se han detectado en cada octava sobre un ejemplo dado que tenga más de 2000 puntos representativos en cada escala. Para ello, he calculado los **KeyPoints** con los valores de los parámetros siguientes: **blockSize=5**, **threshold=50**, **winsize=ksizeSobel=ksizeDeriv=3**, **niveles=4**.

A continuación, se mostrará una imagen para los **KeyPoints** de cada octava donde se dibujan los **KeyPoints** en color rojo.

Para obtener la Figura 15 simplemente generamos un color aleatorio diferente a la hora de pintar los **KeyPoints** de cada escala cuando llamamos a la función **drawKeypoints**.

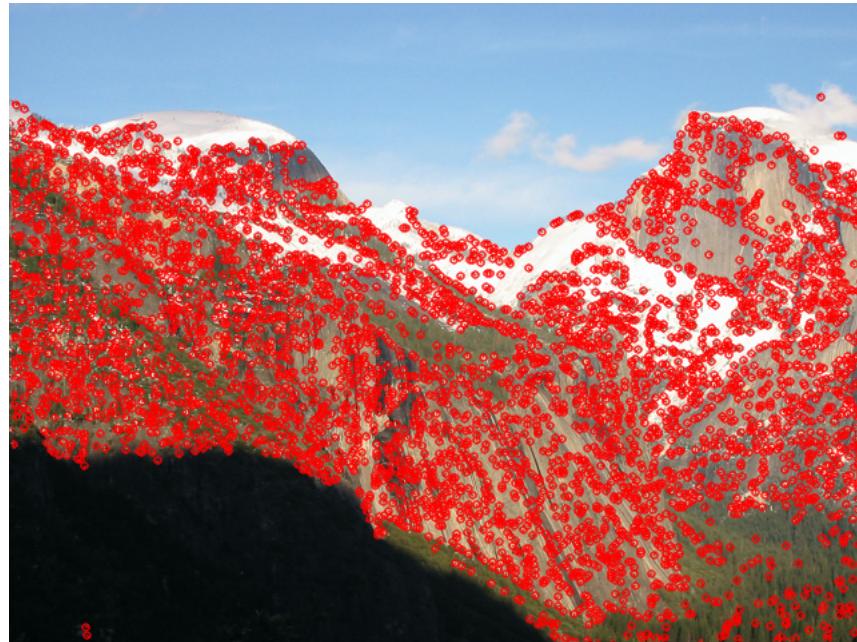


Figura 11: Octava 1. 4360 KeyPoints.

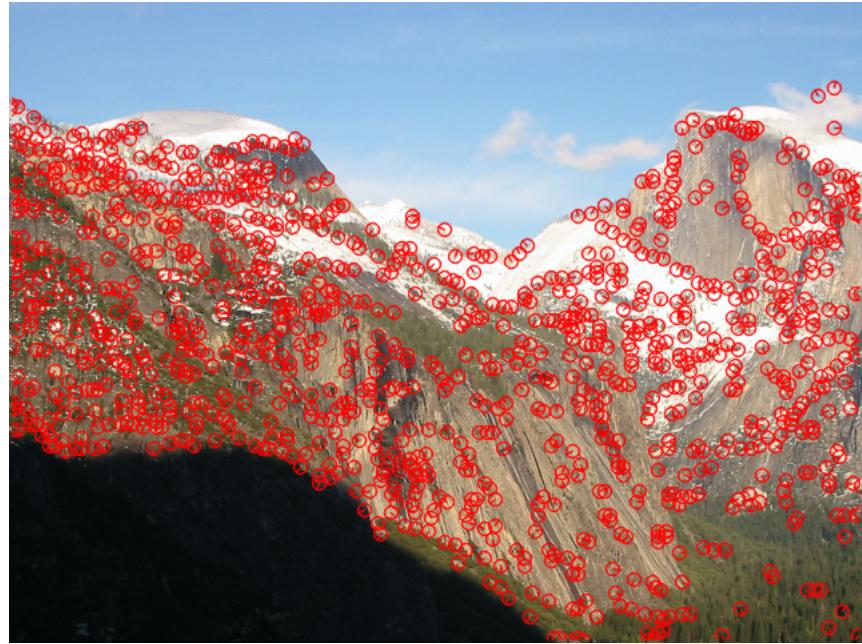


Figura 12: Octava 2. 1061 KeyPoints.



Figura 13: Octava 3. 260 KeyPoints.



Figura 14: Octava 4. 81 KeyPoints.

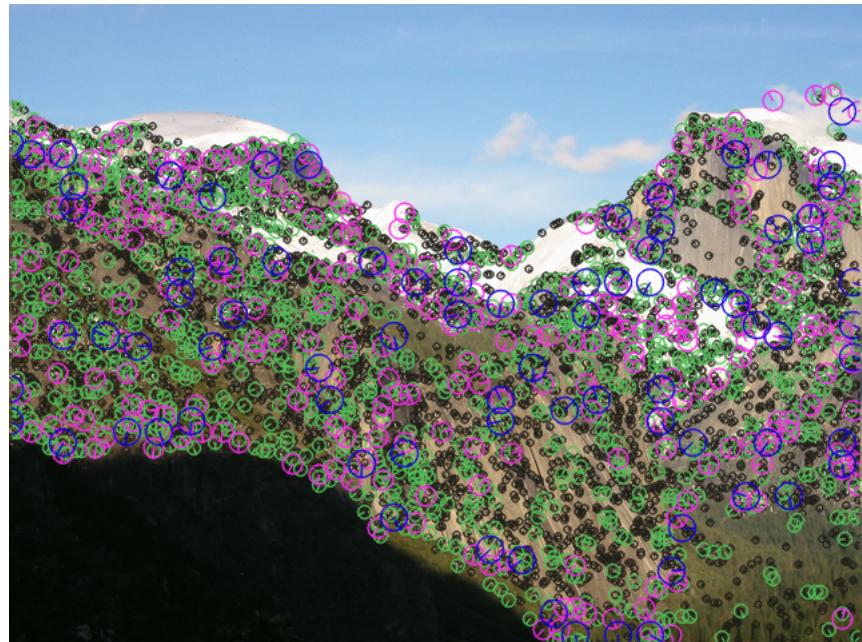


Figura 15: Representación de las 4 escalas. 5762 KeyPoints.

#### 1.4. Calcular las coordenadas subpixel de cada Key-Point usando la función `cornerSubPix` de OpenCV y mostrar 3 submatrices con el refinamiento obtenido.

La cabecera de la función utilizada para realizar este apartado es la siguiente:

```
def ejercicio1D(im, kpoints, n=3):
```

Donde cada parámetro viene definido como:

- **im**: imagen sobre la que se van a dibujar los puntos Harris para cada nivel de la pirámide.
- **kpoints**: conjunto de **KeyPoints** obtenidos en el apartado A con los valores de los parámetros que se hayan especificado.
- **n**: número de submatrices (10x10) con zoom x5 que muestran un píxel original y uno refinado con la función **cornerSubPix**. Por defecto toma el valor de 3, que son el número de muestras que se nos pide en el enunciado.

He creado una función **refKeyPts** que recibe además de **im** y **kpoints**, los siguientes parámetros:

- **winSize**: la mitad de la longitud lateral de la ventana de búsqueda. Por defecto toma el valor de 5. Por lo que el tamaño de la ventana será de  $(5 \cdot 2 + 1) \times (5 \cdot 2 + 1) = 11 \times 11$
- **zeroZone**: la mitad del tamaño de la región muerta en el medio de la zona de búsqueda sobre la cual no se realiza la suma en el cálculo de los gradientes para el refinamiento. Normalmente se usa para evitar posibles singularidades de la matriz de autocorrelación. Por defecto toma el valor de -1, para indicar que no existe tal tamaño.
- **stop\_criteria**: criterio para la terminación del proceso iterativo de refinamiento de esquinas. En nuestro caso usamos por defecto el criterio (`cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER`,

30, 0.001), indicando que la búsqueda se detendrá después de 30 iteraciones o cuando la posición de la esquina se mueva en menos de 0.001 en alguna iteración.

La función **refKeyPts** realiza las siguientes operaciones:

- Almacenamos las posiciones de los **KeyPoints** en un vector (**new\_points**) para cada octava, no sin antes hacer una conversión a **float32** para que pueda operar con ellas la función **cornerSubPix**.
- Creamos la pirámide gaussiana de la imagen **im**.
- Calculamos el refinamiento de **KeyPoints** para los puntos clave de cada escala con la función **cornerSubPix** y los parámetros que hemos especificado por defecto.

```
cv2.cornerSubPix(pirIm[ i ] , new_points[ i ] , (winSize ,  
winSize) , (zeroZone , zeroZone) , stop_criteria)
```

Una vez obtenemos el vector de posiciones ajustadas por la función **refKeyPts**, seleccionamos de forma aleatoria **n·10** píxeles, en nuestro caso 30. Y de todos los seleccionados, nos quedamos con 3 que cumplan que sus posiciones sean diferentes a las de los **KeyPoints** obtenidos en el apartado A. De esta forma, cuando mostremos ambos píxeles, podremos observar la posición original y la ajustada.

Para poder hacer uso de la función **drawKeypoints** y dibujar los puntos. Crearemos 6 (3 originales y sus 3 refinamientos) **KeyPoints** con valor nulo de **\_size** y **\_angle**. Coloreamos en rojo los 3 **KeyPoints** originales y en verde los 3 **KeyPoints** refinados.

Finalmente con la función **resize** extraeremos 3 muestras de matrices 10x10 centradas en el píxel del **KeyPoint** original y con un zoom de 5.

```
cv2.resize(subim[y-5:y+5,x-5:x+5] , (0,0) , fx=5, fy=5)
```

A continuación mostraremos los resultados obtenidos donde podemos ver la pequeña diferencia de precisión entre las posiciones de los **KeyPoints** originales y los obtenidos con el ajuste.

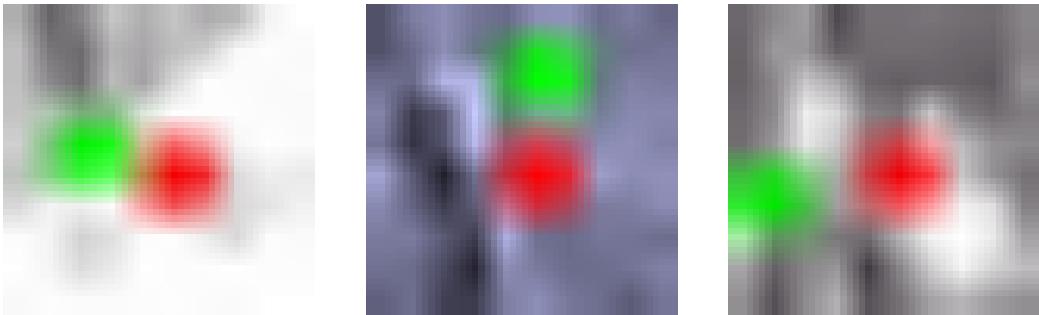


Figura 16: Refinamiento KeyPoints. Originales (rojo), Ajustados (verde).

**2. Detectar y extraer los descriptores AKA-ZE de OpenCV. Establecer las correspondencias existentes entre cada dos imágenes y usar los criterios “BruteForce+crossCheck” y “Lowe-Average-2NN”.**

La cabecera de la función utilizada para realizar este apartado es la siguiente:

```
def ejercicio2(im1, im2, k=[0.7]):
```

Donde cada parámetro viene definido como:

- **im1**: imagen 1 para la correspondencia.
- **im2**: imagen 2 para la correspondencia.
- **k**: constante diferencia de distancias para el criterio ”**Lowe-Average-2NN**” que puede especificarse como lista para obtener varios resultados

para el mismo criterio y que usaremos para comparar su funcionamiento.

Los criterios a partir de los cuales obtenemos las correspondencias, están implementados como funciones independientes **bruteForceCC** y **loweAvg2NN**. Cada una implementa la detección de correspondencias con los criterios **”BruteForce+CrossCheck”** y **”Lowe-Average-2NN”** respectivamente.

Para este ejercicio, necesitamos extraer los **KeyPoints** y los descriptores, para ello creamos el objeto de la clase que implementa el detector y extractor AKAZE de *Open CV* con los parámetros por defecto.

```
akaze = cv2.AKAZE_create()
```

Una vez creado el objeto, debemos obtener los **KeyPoints** y descriptores de la imagen, para ello hacemos uso de la función **detectAndCompute** que se encarga de hacerlo para cada imagen. No es necesario especificar el parámetro **mask**, por ello lo dejaremos con valor **None**. Este paso y el anterior serán comunes en ambos criterios.

```
kpts1, desc1 = akaze.detectAndCompute(im1, None)
kpts2, desc2 = akaze.detectAndCompute(im2, None)
```

Para la función que implementa el criterio **”BruteForce+CrossCheck”** utilizamos el matcher **BMatcher** con el parámetro **crossCheck** activado y usaremos la norma L2 que viene por defecto (**normType = NORM\_L2**).

```
bf = cv2.BFMatcher(crossCheck=True)
matches = bf.match(desc1, desc2)
```

El método **match** compara cada punto de una imagen con todos los de la otra, repitiendo este proceso para ambas imágenes. Para cada punto, elige como pareja aquel que obtenga menos valor en el cálculo de la distancia especificada, que en nuestro caso, al activar **crossCheck** será aquella que se corresponda en ambos sentidos.

Para la función que implementa el criterio **”Lowe-Average-2NN”** utilizamos el matcher **BFMatcher** esta vez con el parámetro **crossCheck** desactivado. Además utilizaremos la función **knnMatch** con k=2 para establecer las correspondencias.

```
bf = cv2.BFMatcher(crossCheck=False)
matches = bf.knnMatch(desc1, desc2, k=2)
```

El método **knnMatch** con k=2 solamente calcula las correspondencias de una imagen con la otra, obteniendo como pareja el valor más cercano. Por ello, es necesario aplicar el criterio **”Lowe-Average-2NN”** para filtrar los resultados “correctos”, considerando que una correspondencia es correcta cuando el ratio entre la distancia de esa correspondencia y la siguiente no exceda un cierto umbral al que llamaremos **k**.

```
correct_matches = []
for a,b in matches:
    if a.distance < k*b.distance:
        correct_matches.append(a)
```

Finalmente y haciendo uso de la función **drawMatches** dibujaremos en el mismo canvas las dos imágenes junto con sus correspondencias. Para cada criterio, nos quedaremos con un subconjunto de 100 correspondencias (en el caso de que haya más de 100) obtenidas de forma aleatoria y sin repetición usando:

```
if len(matches) > 100:
    subset = random.sample(matches, 100)
else:
    subset = matches
im3 = cv2.drawMatches(im1, kpts1, im2, kpts2, matches1to2=
    subset, outImg=np.array([]), flags=2)
```

El flag 2 o NOT\_DRAW\_SINGLE\_POINTS especifica que no se dibujen **KeyPoints** individuales.

## 2.1. Valorar la calidad de los resultados obtenidos a partir de un par de ejemplos aleatorios de 100 correspondencias.

Obtendremos los resultados para ambos criterios y mostraremos también dos versiones para el criterio **"Lowe-Average-2NN"**, usando  $k=0.7$  y  $k=0.8$  y valorando las correspondencias en cada caso haciendo uso de las imágenes Yosemite1.jpg y Yosemite2.jpg

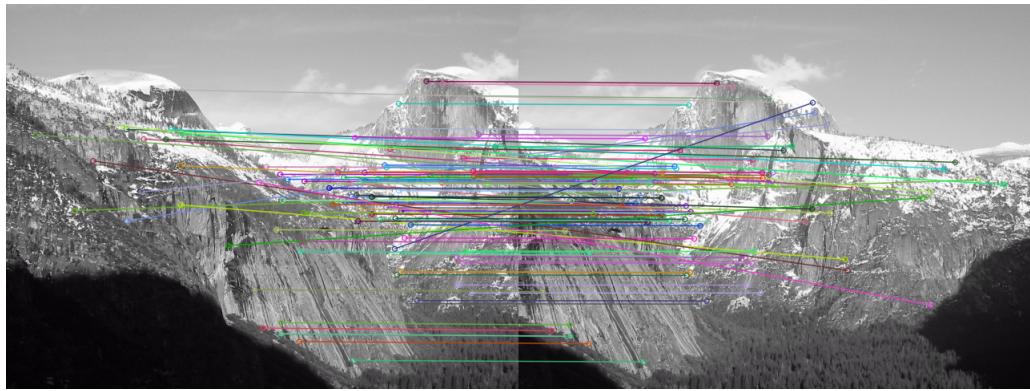


Figura 17: Correspondencias criterio "BruteForce+CrossCheck"



Figura 18: Correspondencias criterio "Lowe-Average-2NN" con  $k=0.7$

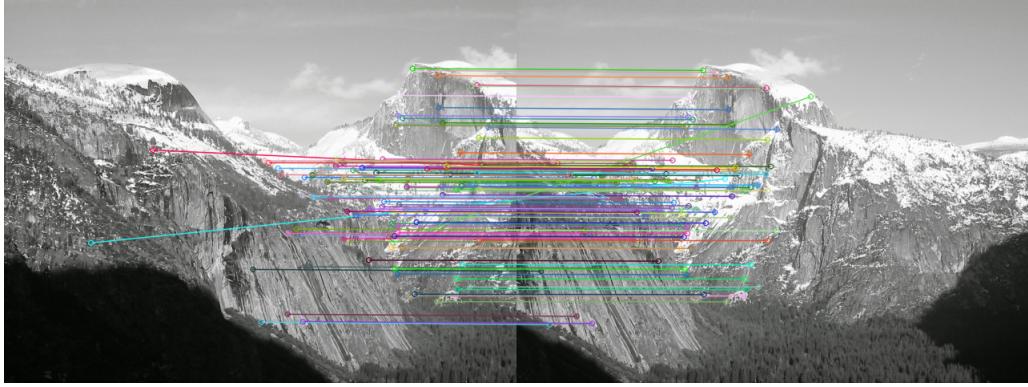


Figura 19: Correspondencias criterio ”Lowe-Average-2NN” con  $k=0.8$

El criterio **”BruteForce+CrossCheck”** obtiene un gran número de correspondencias correctas (visualmente podemos ver que casi todas tienen la misma dirección) aunque hay un porcentaje observable de correspondencias erróneas (como podemos apreciar en algunas líneas oblicuas que atraviesan diagonalmente ambas imágenes).

Su porcentaje de acierto es alto a pesar de la simplicidad del criterio y si en vez de obtener 100 correspondencias de manera aleatoria obtuviesemos las 100 mejores, podríamos obtener unos buenos resultados para estas imágenes.

El criterio **”Lowe-Average-2NN”** obtiene casi todas las correspondencias correctas, ya que es visible que los puntos de salida y llegada, se corresponden con la región de solapamiento en cada imagen. Además, obtenemos un menor porcentaje de correspondencias erróneas que utilizando el criterio anterior.

Si modificamos el parámetro  $k$  para el criterio **”Lowe-Average-2NN”** observamos la mejora del rendimiento conforme disminuimos el valor del ratio. En el caso con  $k=0.7$ , obtenemos menos correspondencias erróneas que con 0.8, pero debemos tener en cuenta que disminuyendo el valor de  $k$ , aunque las correspondencias sean mejores, también obtendremos un menor número de ellas, por lo que a veces es preferible obtener un pequeño número de correspondencias erróneas que no quedarnos con un conjunto muy reducido de correspondencias correctas.

### **3. Escribir una función que genere un Mosaico de calidad a partir de $N = 2$ imágenes relacionadas por homografías.**

La cabecera de la función utilizada para realizar este apartado es la siguiente:

```
def ejercicio3(vim, k=0.7, eps=1, name="Mosaico_ej3"):
```

Donde cada parámetro viene definido como:

- **vim**: vector de imágenes para generar el mosaico. Para este apartado el tamaño del vector será de 2.
- **k**: constante diferencia distancias para el criterio **"Lowe-Average-2NN"**. Usaremos por defecto  $k=0.7$  ya que ha obtenido buenos resultados en el apartado anterior.
- **eps**: Error de reproyección máximo permitido para tratar un par de puntos como inlier usando RANSAC. Por defecto 1.
- **name**: nombre que recibirá el mosaico. Por defecto **"Mosaico\_ej3"**.

#### **3.1. Definir una imagen en la que pintaremos el mosaico.**

Para la realización del mosaico es necesario definir un canvas de color negro (matriz de ceros **uint8**) con unas dimensiones lo suficientemente grandes para que se puedan representar todas las imágenes de las que disponemos.

Para ello, definiremos el canvas con altura y anchura equivalentes a la suma de las alturas y anchuras de todas las imágenes del vector **vim** respectivamente. Además añadiremos 1 o 3 canales al canvas dependiendo de si las imágenes son en escala de grises o a color.

```
height = sum([im.shape[0] for im in vim])
width = sum([im.shape[1] for im in vim])
```

```

if len(vim[0].shape) == 3:
    canvas = np.zeros((height, width, 3), dtype=np.uint8)
else:
    canvas = np.zeros((height, width), dtype=np.uint8)

```

A continuación, tenemos que definir la homografía que lleva la primera imagen al centro del canvas que hemos creado. La homografía vendrá definida como una traslación haciendo que la primera imagen se situe en el punto medio del canvas. Construimos la matriz de la homografía con los siguientes valores de x e y. Este apartado está implementado en la función **homografía\_centro**.

```

x = (width_canvas - width_im)/2
y = (height_canvas - height_im)/2
M0 = np.array([[1, 0, x], [0, 1, y], [0, 0, 1]])

```

### 3.2. Definir la homografía que lleva cada una de las imágenes a la imagen del mosaico.

La función utilizada para implementar este apartado se denomina **homografía**, la cual recibe dos imágenes y extrae sus correspondencias a partir de la función **loweAvg2NN** del ejercicio 2. Además obtenemos los **KeyPoints** de cada imagen que también establecimos como valores de retorno de la función **loweAvg2NN**.

Para calcular la homografía es necesario separar en dos listas (**src\_pts** y **dst\_pts**) los orígenes y destinos de las correspondencias. Para ello hacemos uso de los campos **queryIdx** y **trainIdx** de cada correspondencia del vector de matches obtenido.

```

akaze = cv2.AKAZE_create()
(kpts1, desc1), (kpts2, desc2), matches, im = loweAvg2NN(im1,
    im2, akaze, k)
src_pts = np.array([kpts1[m.queryIdx].pt for m in matches])
dst_pts = np.array([kpts2[m.trainIdx].pt for m in matches])

```

Finalmente, calculamos la matriz de la homografía con la función **findHomography** de *Open CV* usando el método RANSAC y nos quedamos

con la primera componente del valor devuelto por la función que es el que contiene la matriz  $3 \times 3$  que define la homografía entre ambas imágenes.

El método RANSAC se basa en la idea de utilizar unas cuantas correspondencias escogidas de forma aleatoria y estimar una homografía con ellas. Con esta homografía, se valoran los resultados obtenidos calculando cuantos puntos dentro de todas las correspondencias escogidas son compatibles. Si la cantidad de puntos compatibles es suficiente con un error de reprojeción `eps`, se utilizan estos puntos para estimar mejor la homografía que habíamos aproximado.

```
H = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, eps)[0]
```

Utilizamos un bucle para obtener todas las homografías que iremos almacenando en una lista para aplicarlas al final. Añadimos la homografía que lleva la primera imagen al centro como primer elemento de la lista e iteramos calculando el resto de homografías como la composición de las matrices de las homografías calculadas.

```
M0 = homografia_centro(vim[0], canvas)
homografias = [M0]
for i in range(len(vim)-1):
    H = homografia(vim[i+1], vim[i], k, eps)
    M = np.dot(homografias[i], H)
    homografias.append(M)
```

### 3.3. Usar la función `cv2.warpPerspective()` para trasladar cada imagen al mosaico.

Finalmente y una vez que tenemos la lista con todas las homografías calculadas, usamos la función `warpPerspective` para trasladar las imágenes al mosaico. Usaremos el flag `cv.BORDER_TRANSPARENT` tal y como se indica en el guión de la práctica, para evitar la interpolación en los puntos no asignados del canvas y no obtener un mosaico con el resultado de aplicar solamente la última homografía.

La función además tiene como parámetros la homografía (**homografias[i]**) a aplicar sobre la imagen (**vim[i]**) y el ancho y largo (**width** y **height**, respectivamente) del canvas sobre el que nos movemos.

```
for i in range(len(vim)):
    canvas = cv2.warpPerspective(vim[ i ] , homografias [ i ] , (width ,
height ) , dst=canvas , borderMode=cv2.BORDER_TRANSPARENT)
```

Además se ha implementado una función **autocrop** que elimina las filas y columnas de ceros de la imagen pasada por parámetro. Esta función realiza un recorte al canvas, obteniendo un resultado que se ajusta directamente sobre las regiones que contienen las imágenes del mosaico.

El mosaico que obtenemos utilizando las imágenes Yosemite1.jpg y Yosemite2.jpg con valores de  $k=0.7$  y  $\text{eps}=1.0$ , es el siguiente:

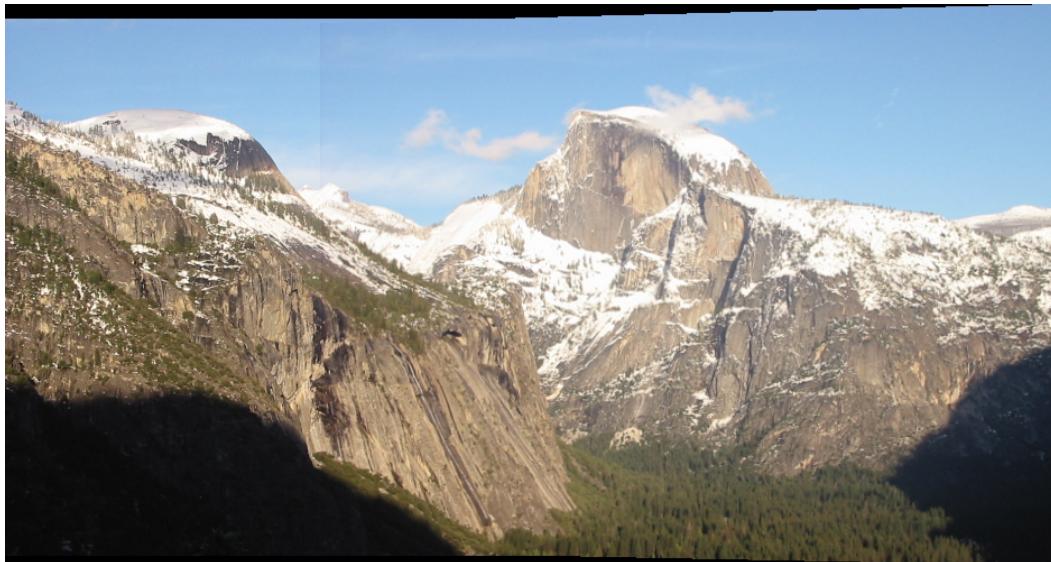


Figura 20: Mosaico con imágenes Yosemite1 y Yosemite2.

## 4. Realizar un mosaico usando todas las imágenes proporcionadas.

La cabecera de la función utilizada para realizar este apartado es la siguiente:

```
def ejercicio4(vim, k=0.7, eps=1, name="Mosaico_ej4"):
```

La función es equivalente en parámetros a la que hemos usado en el ejercicio anterior. Nos limitaremos a crear los mosaicos para todas las imágenes aportadas en DECSAI y visualizarlos.

Crearemos un total de 3 mosaicos. El primero formado por imágenes tomadas desde la fachada frontal de la ETSIIT y los dos siguientes usando los subconjuntos de imágenes yosemite que cuenten con las correspondencias necesarias para realizar el mosaico.

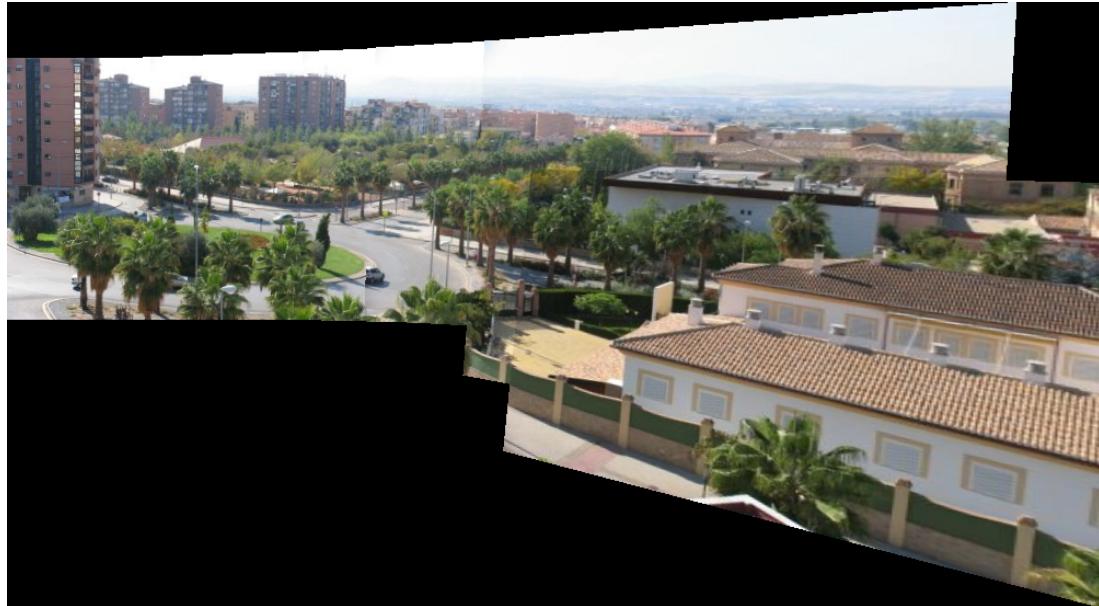


Figura 21: Mosaico con imágenes mosaico0i con  $i \in \{2..,11\}$ .

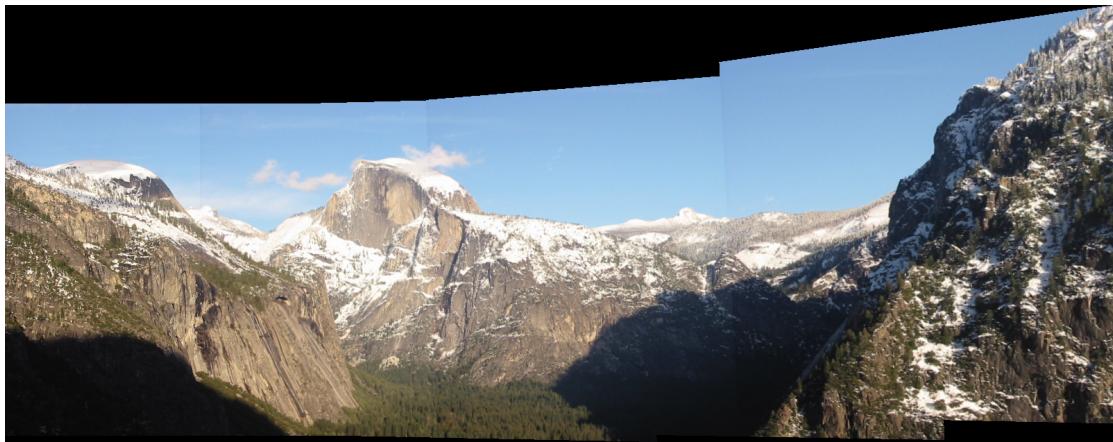


Figura 22: Mosaico con imágenes yosemitei con  $i \in \{1,..,4\}$ .

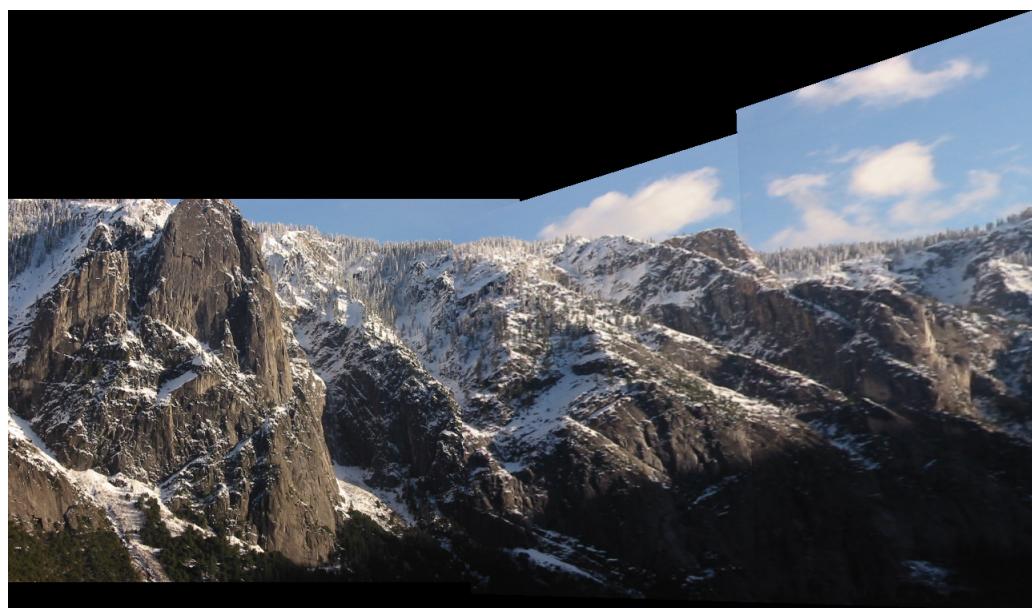


Figura 23: Mosaico con imágenes yosemitei con  $i \in \{5,..,7\}$ .

## **5. Referencias**

- [1] Diapositivas de clase y guión de prácticas.
- [2] Documentación Open CV : <https://docs.opencv.org/4.1.0/index.html>
- [3] PABLO FLORES, JUAN BRAUN. Algoritmo RANSAC: fundamento teórico.