Universidad de Granada

Ingeniería Informática

Computación y Sistemas Inteligentes

Practica 2

Autor: José Antonio Ruiz Millán
Asignatura: Visión por Computador
24 de noviembre de 2018

1. **(3 puntos) Detección de puntos SIFT y SURF. Aplicar la detección de puntos SIFT y SURF sobre las imágenes, representar dichos puntos sobre las imágenes haciendo uso de la función drawKeyPoints. Presentar los resultados con las imágenes Yosemite.rar.**

   Para este ejercicio hago uso de algunas de las funciones de *OpenCV* para crear el objeto *SIFT* y *SURF* y así poder calcular los puntos dentro de cada una de las imágenes. Antes de poner directamente el código utilizado, voy a poner las distintas funciones propias que utilizo, estas funciones ya fueron definidas en la practica anterior pero aún así las pondré de nuevo, en primer lugar, esta función permite dada una imagen y definiendo si la queremos en color o blanco y negro, cargar la imagen.

```
1               def leeimagen(filename,flagColor):
                    img = cv2.imread(filename,flagColor)
3
                    return img
5
```

   Por otra parte, tengo la función que permite visualizar un conjunto de imágenes juntas.

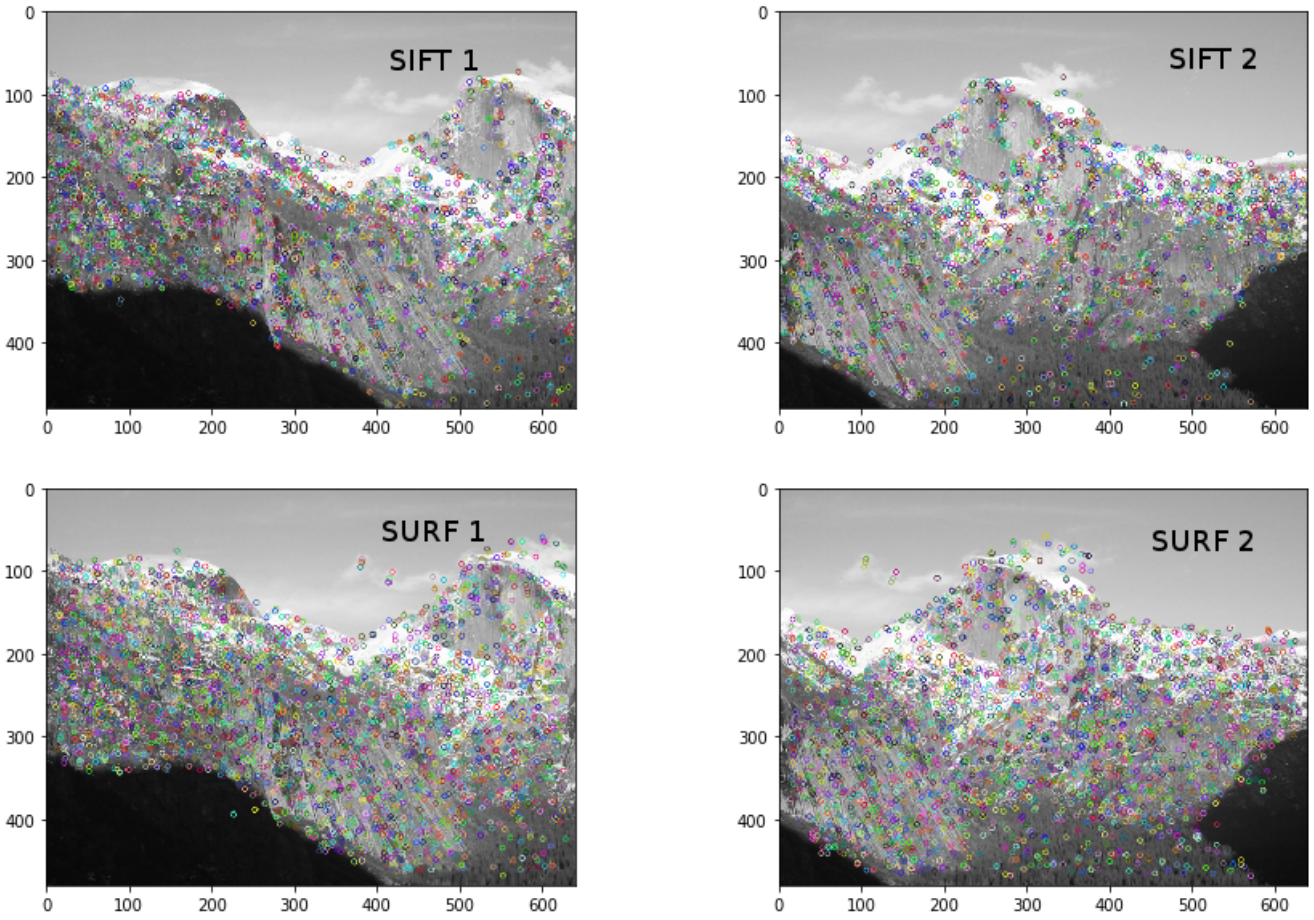
```
def multIM(img,nfil,ncol,tamx,tamy,color=True):
2       fig=plt.figure( figsize =(tamx, tamy))
        for i,im in enumerate(img):
4           fig.add_subplot(nfil, ncol, i+1)
            if color:
6               nimg = cv2.cvtColor(im, cv2.COLOR_BGR2RGB)
            else:
8               nimg = cv2.cvtColor(im,cv2.COLOR_GRAY2RGB)
            plt.imshow(nimg)
10      plt.show()
```

   Una vez enseñado esto, paso a mostrar la resolución de esta parte del ejercicio.

```
img1 = leeimagen("Yosemite1.jpg",1)
2 gray1 = leeimagen("Yosemite1.jpg",0)

4 img2 = leeimagen("Yosemite2.jpg",1)
  gray2 = leeimagen("Yosemite2.jpg",0)
6
  # SIFT
8
  sift  = cv2.xfeatures2d.SIFT_create()
10 kpsi1 = sift.detect(gray1,None)
  kpsi2 = sift.detect(gray2,None)
12
  imgsi1 = cv2.drawKeypoints(gray1,kpsi1,None)
14 imgsi2 = cv2.drawKeypoints(gray2,kpsi2,None)

16 # SURF

18 surf = cv2.xfeatures2d.SURF_create()
  kpsf1 = surf.detect(gray1,None)
20 kpsf2 = surf.detect(gray2,None)
```

```
imgsf1 = cv2.drawKeypoints(gray1,kpsf1,None)
imgsf2 = cv2.drawKeypoints(gray2,kpsf2,None)

#Miestro el resultado
multIM([imgsi1,imgsi2,imgsf1,imgsf2 ],2,2,15,10)

input()
```

Como vemos en el código, en este apartado lo que he realizado es en primer lugar, cargar cada una de las imágenes que voy a utilizar, posteriormente, haciendo uso de la funcion *detect*[1] tanto para *SIFT* como para *SURF* calculamos los *KeyPoints* de cada una de las imágenes para posteriormente mostrarlos utilizando la función *drawKeypoints*[1]



a) **Variar los valores de umbral de la función de detección de puntos hasta obtener un conjunto numeroso ($\geq$ 1000) de puntos SIFT y SURF que sea representativo de la imagen. Justificar la elección de los parámetros en relación a la representatividad de los puntos obtenidos.**

Para el caso de *SIFT* tenemos dos valores para modificar:

- **contrastThreshold:** Este umbral es utilizado para la detección de las características en zonas de bajo contraste dentro de la imagen, por lo que si el valor de este umbral es muy pequeño, estamos detectando muchas características para la detec-

ción de zonas con poco contraste, mientras que si el valor de este umbral es muy alto, eliminaremos características y utilizaremos menos características para las zonas de poco contraste. El valor por defecto en *OpenCV* es 0,04 pero podemos variarlo mas o menos entre 0,02 y 0,1. En mi caso, como tenemos muchos puntos en este tipo de zonas, he decido aumentar el valor y usar 0,06 para así obtener menos puntos en estas zonas.

- **edgeThreshold:** Al contrario que el caso anterior, este umbral se encarga de detectar las características en los bordes, es decir, en zonas donde hay un alto contraste. El valor de este umbral es directamente proporcional a las características que se calculan es este tipo de zonas, es decir, si el valor de este umbral es bajo, obtendremos menos características, y por el contrario, si el valor de este umbral es alto, obtendremos más características. El valor por defecto en *OpenCV* es de 10. En mi caso, también he decidido reducir el valor de este umbral para eliminar algunos puntos más en estas zonas. Este valor se puede mover entre 2 y 15, en mi caso, he decidido usar 6 para eliminar algunos de estos puntos.

Por otra parte, para el caso de *SURF* tenemos:

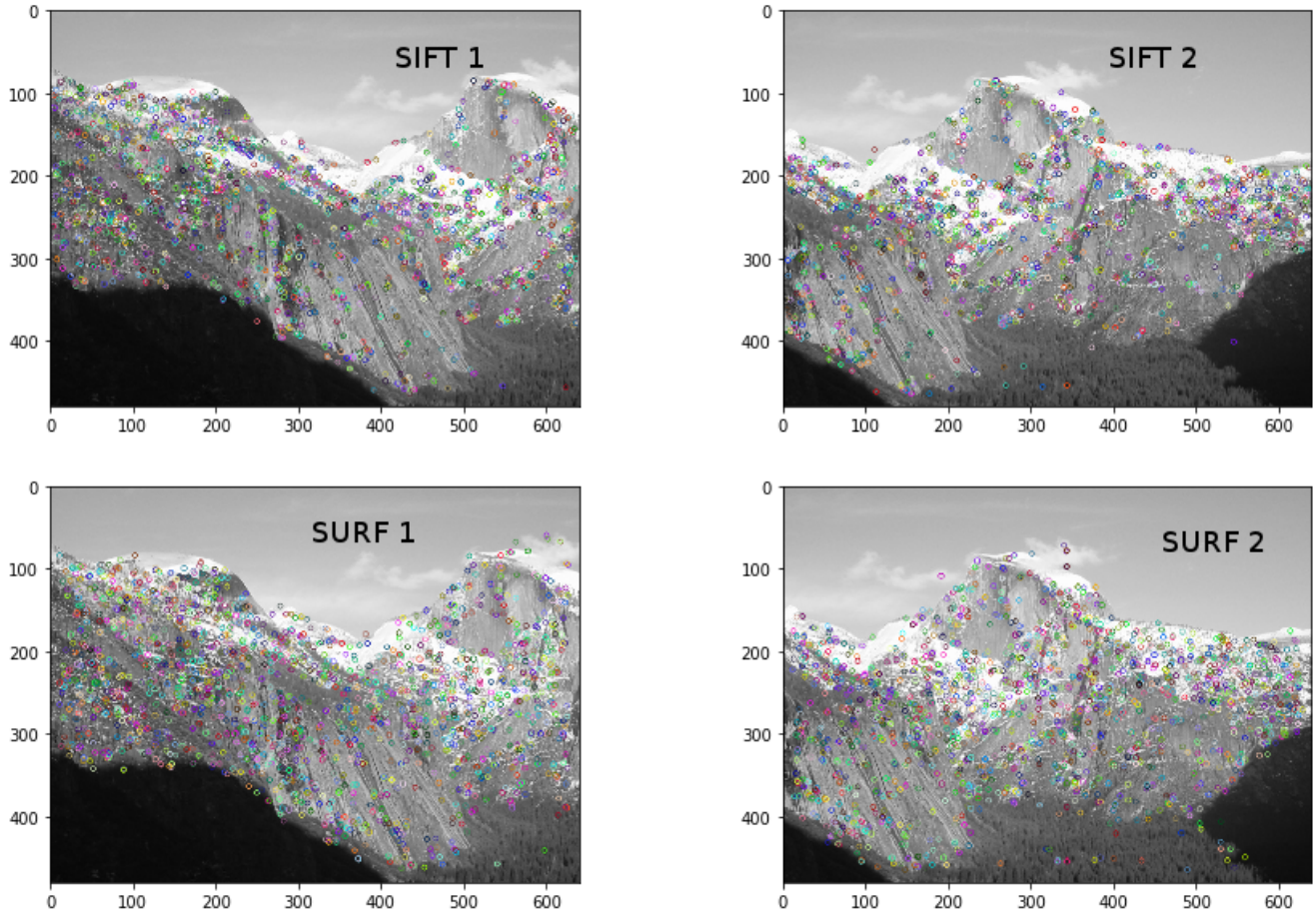
- **hessianThreshold:** Este valor como su nombre indica hace referencia al valor de la hessiana que calcular al obtener los puntos. Modificando este umbral lo que hacemos es quedarnos únicamente con los puntos que su valor de hessian está por encima del umbral. El valor por defecto en *OpenCV* es 100, sin embargo, un valor recomendado para este umbral está entre 300 y 500 dependiendo del contraste de la imagen. En mi caso he decidido usar un valor de 400.

El codigo resultante para la resolución de este apartado es el siguiente:

```
# SIFT

sift  = cv2.xfeatures2d.SIFT_create(contrastThreshold=0.06,edgeThreshold=6)
kpsi1 = sift.detect(gray1,None)
kpsi2 = sift.detect(gray2,None)

imgsi1 = cv2.drawKeypoints(gray1,kpsi1,None)
imgsi2 = cv2.drawKeypoints(gray2,kpsi2,None)

# SURF

surf  = cv2.xfeatures2d.SURF_create(hessianThreshold=400)
kpsf1 = surf.detect(gray1,None)
kpsf2 = surf.detect(gray2,None)

imgsf1 = cv2.drawKeypoints(gray1,kpsf1,None)
imgsf2 = cv2.drawKeypoints(gray2,kpsf2,None)

#Muestro los resultados
multIM([imgsi1,imgsi2,imgsf1,imgsf2],2,2,15,10)

input()
```

Como vemos en el código, la estructura es exactamente la misma que en paso anterior,

sólo que para este caso, tenemos los parámetros modificados.



b) **Identificar cuántos puntos se han detectado dentro de cada octava. En el caso de SIFT, identificar también los puntos detectados en cada capa. Mostrar el resultado dibujando sobre la imagen original un círculo centrado en cada punto y de radio proporcional al valor de sigma usado para su detección (ver circle()) y pintar cada octava en un color.**

Para este apartado crearé dos nuevos modelos de *SIFT* y *SURF* porque para poder representarlos en la imagen con su radio y demás opciones, tantos puntos como los que tenemos calculados en el apartado anterior no son visualizables para poder comentar los resultados. Antes de mostrar el código para este apartado, mostraré las distintas funciones utilizadas.

En primer lugar muestro la función que nos devuelve cuantos puntos se han detectado dentro de cada octava. Esta función solo necesita los puntos en sí y un booleano para indicar si los puntos que le pasas han sido detectado mediante *SIFT* o *SURF*.

```
def puntosPorOctava(kp,surf=True):
    puntos = np.zeros(20)

    for p in kp:
        if surf:
            oc = p.octave
        else:
```

```
            oc,l,s = unpackSIFTOctave(p)
            oc+=1

        puntos[oc] += 1

    return puntos[puntos>0]

```

Como podemos ver, lo que hago es crear un array con un valor suficientemente grande relleno de 0, luego voy recorriendo cada uno de los puntos y en caso de ser puntos obtenidos mediante *SURF*, guardamos el valor de la octava a la que pertenece y en el array creado sumamos uno al valor que pertenece a esa octava. Sin embargo, si el punto que estamos consultando es un punto obtenido con *SIFT*, debemos de realizar un cálculo para obtener la octava ya que ésta viene comprimida. La función que permite descomprimir la octava es el siguiente.

```
def unpackSIFTOctave(kpt):
    """unpackSIFTOctave(kpt)->(octave,layer,scale)
    @created by Silencer at 2018.01.23 11:12:30 CST
    @brief Unpack Sift Keypoint by Silencer
    @param kpt: cv2.KeyPoint (of SIFT)
    """
    _octave = kpt.octave
    octave = _octave&0xFF
    layer  = (_octave>>8)&0xFF
    if octave>=128:
        octave |= -128
    if octave>=0:
        scale = float(1/(1<<octave))
    else:
        scale = float(1<<-octave)
    return (octave, layer, scale)

```

Por otra parte, uno de los métodos utilizados en este ejercicio también debe permitirnos obtener los puntos obtenidos en cada una de las capas para el caso de los puntos *SIFT*, por lo que he definido la siguiente función. Para el funcionamiento de la misma debemos de pasarle los puntos y la octava que queremos saber.

```
def puntosPorCapa(kp,octava):
    puntos = np.zeros(20)

    for p in kp:
        oc,l,s = unpackSIFTOctave(p)
        oc+=1
        if oc == octava:
            puntos[l-1] += 1

    return puntos[puntos>0]

```

En definición es idéntico a la función para saber cuantos puntos tenemos por octava, con la diferencia de que cuando tenemos que un punto pertenece a la octava que estamos

evaluando, comprueba cual es la capa a la que pertenece y suma 1 a la correspondiente capa en el array.

Por último tenemos la función que permite dibujar cada uno de los puntos en la imagen teniendo en cuenta su radio y la octava a la que pertenece. Para el funcionamiento de la misma tenemos que pasarle la imagen en la que queremos que se pinten los puntos (no se modifica), lo puntos, una lista con los colores e indicar si son puntos *SIFT* o *SURF*.

```
def pintaPuntos(img,kp,colores,surf=True):
    img2 = img.copy()

    for p in kp:
        if(surf):
            oc = p.octave
        else:
            oc,l,s = unpackSIFTOctave(p)
        cv2.circle(img2,(int(p.pt[0]),int(p.pt[1])),int(p.size/2),list(map(int,colores[oc])),1,cv2
    .LINE_AA)

    return img2

```

Por lo que el código para la resolución de este ejercicio es el siguiente:

```
# SIFT

siftm = cv2.xfeatures2d.SIFT_create(1000)
kpsi1m = siftm.detect(gray1,None)
kpsi2m = siftm.detect(gray2,None)

print("La imagen 1 con SIFT tiene "+str(len(kpsi1m))+" KeyPoints.")
pp = puntosPorOctava(kpsi1m,False)
for i,p in enumerate(pp):
    print("La octava "+str(i)+" tiene "+str(int(pp[i]))+" KeyPoints.")
    cp = puntosPorCapa(kpsi1m,i)
    for j in range(cp.size):
        print("La capa "+str(j)+" de la octava "+str(i)+" tiene "+str(int(cp[j]))+" KeyPoints.")

print("\nLa imagen 2 con SIFT tiene "+str(len(kpsi2m)))
pp = puntosPorOctava(kpsi2m,False)
for i,p in enumerate(pp):
    print("La octava "+str(i)+" tiene "+str(int(pp[i]))+" KeyPoints.")
    cp = puntosPorCapa(kpsi2m,i)
    for j in range(cp.size):
        print("La capa "+str(j)+" de la octava "+str(i)+" tiene "+str(int(cp[j]))+" KeyPoints.")

nimg1 = cv2.cvtColor(gray1,cv2.COLOR_GRAY2BGR)
nimg2 = cv2.cvtColor(gray2,cv2.COLOR_GRAY2BGR)

colores = np.array([[0,  0,   255],
    [  0, 255,  0],
    [ 255,  0,  0],
    [ 255,  0,  255],
    [255,  255,  0],
    [0,  255,  255],
    [255,  0,  125],
```

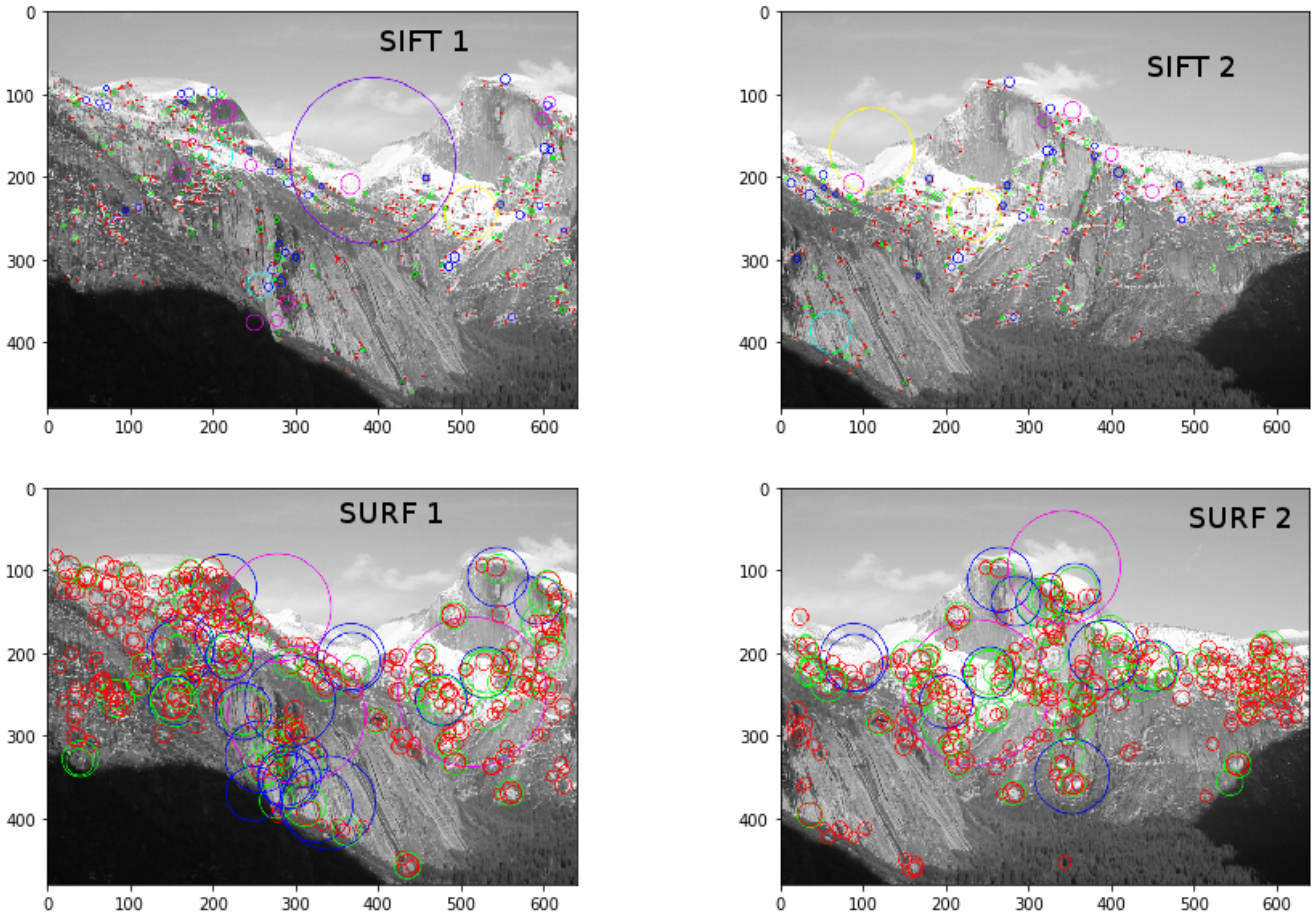
```
      [125,  0,   255],
       [0,0,0],
      [0,  150,  255]],  dtype=np.uint8)

imgsi1 = pintaPuntos(nimg1,kpsi1m,colores,False)
imgsi2 = pintaPuntos(nimg2,kpsi2m,colores,False)

# SURF

surfm = cv2.xfeatures2d.SURF_create(2000)
kpsf1m = surfm.detect(gray1,None)
kpsf2m = surfm.detect(gray2,None)

print("\nLa imagen 1 con SURF tiene "+str(len(kpsf1m)))
pp = puntosPorOctava(kpsf1m)
for i,p in enumerate(pp):
    print("La octava "+str(i)+" tiene "+str(int(pp[i]))+" KeyPoints.")

print("\nLa imagen 2 con SURF tiene "+str(len(kpsf2m)))
pp = puntosPorOctava(kpsf2m)
for i,p in enumerate(pp):
    print("La octava "+str(i)+" tiene "+str(int(pp[i]))+" KeyPoints.")

imgsf1 = pintaPuntos(nimg1,kpsf1m,colores)
imgsf2 = pintaPuntos(nimg2,kpsf2m,colores)

multIM([imgsi1,imgsi2,imgsf1,imgsf2],2,2,15,10)

input()
```

Básicamente lo que realizo es una vez tengo los puntos, (que como he comentado anteriormente he disminuido el número de puntos para que puedan ser representados pero estos puntos no serán utilizandos en los ejercicios siguientes para realizarlos, si no que utilizaré los anteriores a estos) calculo los puntos por octavas y por capas, los muestro en texto y luego muestro las imágenes con estos puntos utilizando las funciones comentadas anteriormente.

Como resultado, en mi caso obtengo lo siguiente:

```
La imagen 1 con SIFT tiene 1000 KeyPoints.
La octava 0 tiene 775 KeyPoints.
La capa 0 de la octava 0 tiene 418 KeyPoints.
La capa 1 de la octava 0 tiene 215 KeyPoints.
La capa 2 de la octava 0 tiene 142 KeyPoints.
La octava 1 tiene 166 KeyPoints.
La capa 0 de la octava 1 tiene 81 KeyPoints.
La capa 1 de la octava 1 tiene 57 KeyPoints.
La capa 2 de la octava 1 tiene 28 KeyPoints.
La octava 2 tiene 44 KeyPoints.
La capa 0 de la octava 2 tiene 22 KeyPoints.
La capa 1 de la octava 2 tiene 12 KeyPoints.
La capa 2 de la octava 2 tiene 10 KeyPoints.
La octava 3 tiene 11 KeyPoints.
La capa 0 de la octava 3 tiene 6 KeyPoints.
La capa 1 de la octava 3 tiene 1 KeyPoints.
```

```
La capa 2 de la octava 3 tiene 4 KeyPoints.
La octava 4 tiene 2 KeyPoints.
La capa 0 de la octava 4 tiene 2 KeyPoints.
La octava 5 tiene 1 KeyPoints.
La capa 0 de la octava 5 tiene 1 KeyPoints.
La octava 6 tiene 1 KeyPoints.
La capa 0 de la octava 6 tiene 1 KeyPoints.

La imagen 2 con SIFT tiene 1000
La octava 0 tiene 767 KeyPoints.
La capa 0 de la octava 0 tiene 428 KeyPoints.
La capa 1 de la octava 0 tiene 210 KeyPoints.
La capa 2 de la octava 0 tiene 129 KeyPoints.
La octava 1 tiene 191 KeyPoints.
La capa 0 de la octava 1 tiene 94 KeyPoints.
La capa 1 de la octava 1 tiene 63 KeyPoints.
La capa 2 de la octava 1 tiene 34 KeyPoints.
La octava 2 tiene 34 KeyPoints.
La capa 0 de la octava 2 tiene 16 KeyPoints.
La capa 1 de la octava 2 tiene 8 KeyPoints.
La capa 2 de la octava 2 tiene 10 KeyPoints.
La octava 3 tiene 5 KeyPoints.
La capa 0 de la octava 3 tiene 3 KeyPoints.
La capa 1 de la octava 3 tiene 1 KeyPoints.
La capa 2 de la octava 3 tiene 1 KeyPoints.
La octava 4 tiene 1 KeyPoints.
La capa 0 de la octava 4 tiene 1 KeyPoints.
La octava 5 tiene 2 KeyPoints.
La capa 0 de la octava 5 tiene 1 KeyPoints.
La capa 1 de la octava 5 tiene 1 KeyPoints.

La imagen 1 con SURF tiene 422
La octava 0 tiene 329 KeyPoints.
La octava 1 tiene 68 KeyPoints.
La octava 2 tiene 22 KeyPoints.
La octava 3 tiene 3 KeyPoints.

La imagen 2 con SURF tiene 350
La octava 0 tiene 274 KeyPoints.
La octava 1 tiene 63 KeyPoints.
La octava 2 tiene 11 KeyPoints.
La octava 3 tiene 2 KeyPoints.

```



c) **Mostrar cómo con el vector de keyPoint extraídos se pueden calcular los descriptores SIFT y SURF asociados a cada punto usando OpenCV.**

En este caso únicamente tenemos que utilizar la función *compute*[1] de *OpenCV* para obtener los descriptores de la imagen, dados unos *KeyPoints*.

```
#SIFT

kpsi1,desi1 = sift.compute(gray1,kpsi1)
kpsi2,desi2 = sift.compute(gray2,kpsi2)

#SURF

kpsf1,desf1 = sift.compute(gray1,kpsf1)
kpsf2,desf2 = sift.compute(gray2,kpsf2)

input()
```

2. **(2.5 puntos) Usar el detector-descriptor SIFT de OpenCV sobre las imágenes de Yosemite.rar (cv2.xfeatures2d.SIFT create()). Extraer sus listas de keyPoints y descriptores asociados. Establecer las correspondencias existentes entre ellos usando el objeto BFMatcher de OpenCV y los criterios de correspondencias "BruteForce+crossCheck y "Lowe-Average-2NN". (NOTA: Si se usan los resultados propios del puntos anterior en lugar del cálculo de SIFT de OpenCV se**

añaden 0.5 puntos)

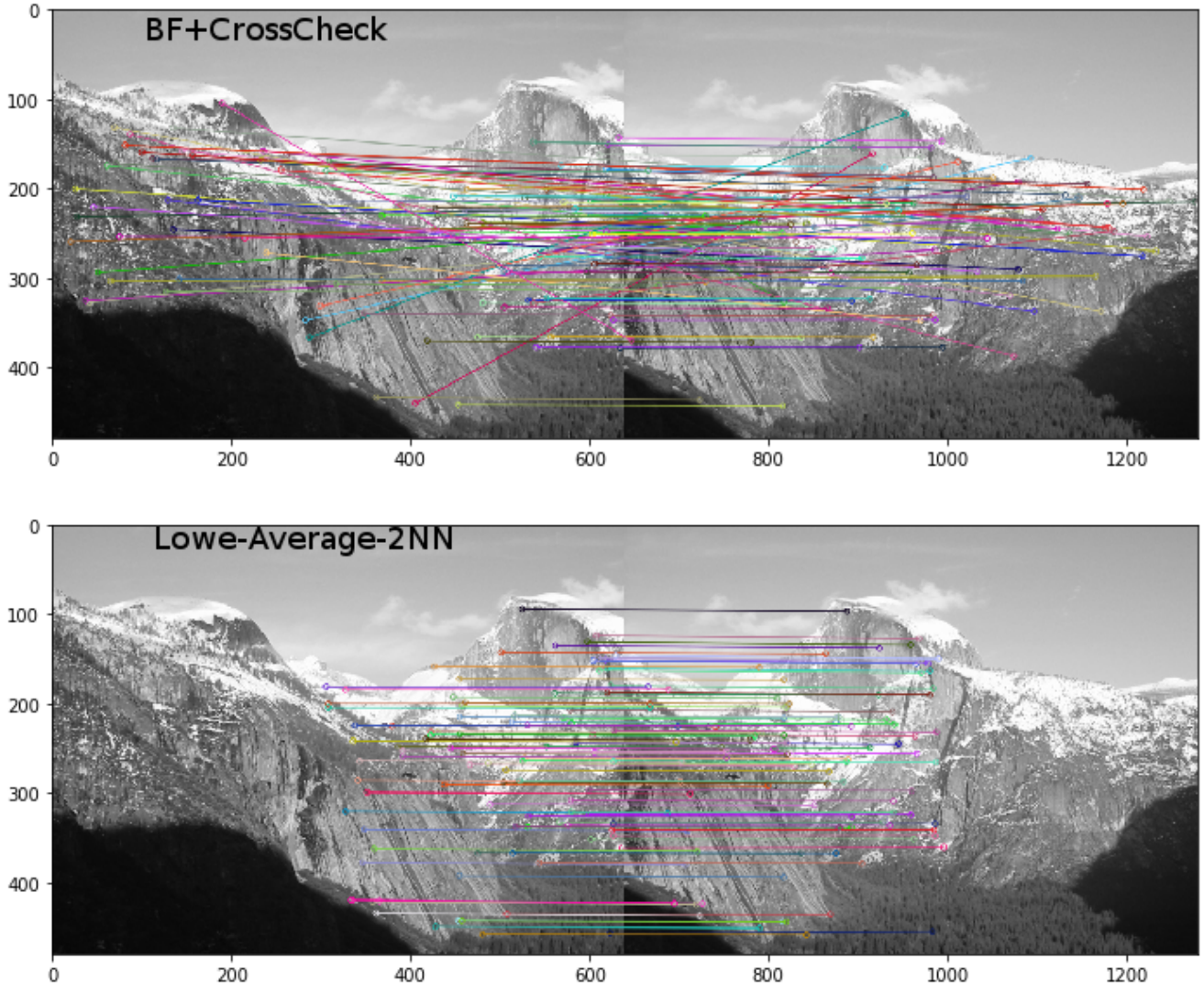
Como ya tenemos calculados los *KeyPoints* y los descriptores de los ejercicios anteriores, no hay que realizar ningún proceso para calcularlos y podemos pasar directamente a los siguientes apartados.

a) **Mostrar ambas imágenes en un mismo canvas y pintar líneas de diferentes colores entre las coordenadas de los puntos en correspondencias. Mostrar en cada caso 100 elegidas aleatoriamente.**

Para este apartado utilizaré únicamente funciones de *OpenCV*[1] y no funciones definidas por mi, por lo que la resolución del ejercicio es la siguiente.

```
# ========================================================
#  BruteForce+crossCheck
# ========================================================

bf = cv2.BFMatcher(crossCheck=True)

matches = bf.match(desi1,desi2)

matches = sorted(matches, key = lambda x:x.distance)
sample = random.sample(matches,100)

bfc = cv2.drawMatches(gray1,kpsi1,gray2,kpsi2,sample,None,flags=2)

# ========================================================
#  Lowe-Average-2NN
# ========================================================

bf = cv2.BFMatcher(crossCheck=False)

matches = bf.knnMatch(desi1,desi2,k=2)

good = []
for m,n in matches:
    if m.distance < 0.7*n.distance:
        good.append([m])

sample = random.sample(good,100)

bfnn = cv2.drawMatchesKnn(gray1,kpsi1,gray2,kpsi2,sample,None,flags=2)

multIM([bfc,bfnn],2,1,15,10)

input()
```

Obteniendo como resultado lo siguiente:



b) **Valorar la calidad de los resultados obtenidos en términos de las correspondencias válidas observadas por inspección ocular y las tendencias de las líneas dibujadas.**

Respondo en el apartado (c) ya que preguntan lo mismo.

c) **Comparar ambas técnicas de correspondencias en términos de la calidad de sus correspondencias (suponer 100 aleatorias e inspección visual).**

Como podemos ver claramente en las imágenes, el método *BF+CrossCheck* obtiene peores resultados que el método *Lowe-Average-2NN*, esto se debe claramente a que el segundo método está mas elaborado que el primero, siendo una mejora del mismo, es decir, en el segundo método estamos quedandonos con los mejores para cada match y con esto conseguimos obtener un conjunto de matches de calidad, sin embargo, para el primer caso, al no hacer ningún tipo de discriminación, obtenemos muchos más matches que el segundo caso pero algunos de ellos de peor calidad. Si en vez de obtener 100 puntos aleatorios ordenamos estos puntos y nos quedamos con los 100 primeros, obtendríamos un resultado notablemente bueno por parte de ambos métodos. En nuestro caso, como hemos decidido realizar la conclusión respecto a 100 puntos aletorios y

mediante inspección ocular, diré que el segundo método es mejor que el primero.

3. **(2.5 puntos) Escribir una función que genere un mosaico de calidad a partir de N = 3 imágenes relacionadas por homografías, sus listas de keyPoints calculados de acuerdo al punto anterior y las correspondencias encontradas entre dichas listas. Estimar las homografías entre ellas usando la función cv2.findHomography(p1,p2, CV RANSAC,1). Para el mosaico será necesario.**

   a) **Definir una imagen en la que pintaremos el mosaico.**

   b) **Definir la homografía que lleva cada una de las imágenes a la imagen del mosaico.**

   c) **Usar la función cv2.warpPerspective() para trasladar cada imagen al mosaico (Ayuda: Usar el flag BORDER TRANSPARENT de warpPers- pective).**

   Para este ejercicio he tenido que crear una funcion que he llamado *mosaico3* que será la encargada de crear un mosaico pasándole 3 imágenes, los *KeyPoints* de cada una de ellas y los correspondientes matches. También he decidio incluirle un parámetro booleano para indicar si se quiere eliminar los bordes sobrantes de la imágen o no.

```
def mosaico3(imgs,kps,matches,r=True):
    maxx = 0
    maxy = 0
    for im in imgs:
        maxy = im.shape[0] if im.shape[0] > maxy else maxy
        maxx = im.shape[1] if im.shape[1] > maxx else maxx

    mosaico_size = [maxx*3,maxy*3]

    H = np.eye(3)
    H[1][2] = (mosaico_size[1]/2-(maxy/2))

    result = cv2.warpPerspective(imgs[0],H,(mosaico_size[0], mosaico_size[1]) )

    p1 = np.array([kps[0][m.queryIdx].pt for m in matches[0]])
    p2 = np.array([kps[1][m.trainIdx].pt for m in matches[0]])
    homography,masc = cv2.findHomography(p2,p1, cv2.RANSAC,1)
    cv2.warpPerspective(imgs[1],H.dot(homography),(mosaico_size[0],mosaico_size[1]),dst=result,
     borderMode=cv2.BORDER_TRANSPARENT)

    p1 = np.array([kps[1][m.queryIdx].pt for m in matches[1]])
    p2 = np.array([kps[2][m.trainIdx].pt for m in matches[1]])
    H = H.dot(homography)
    homography,masc = cv2.findHomography(p2,p1, cv2.RANSAC,1)
    cv2.warpPerspective(imgs[2],H.dot(homography),(mosaico_size[0],mosaico_size[1]),dst=result,
     borderMode=cv2.BORDER_TRANSPARENT)

    if(r):
        result = recortar(result)

    return result
```

Esta función es un poco más compleja de explicar, no obstante intentaré explicarla lo más

simple posible.

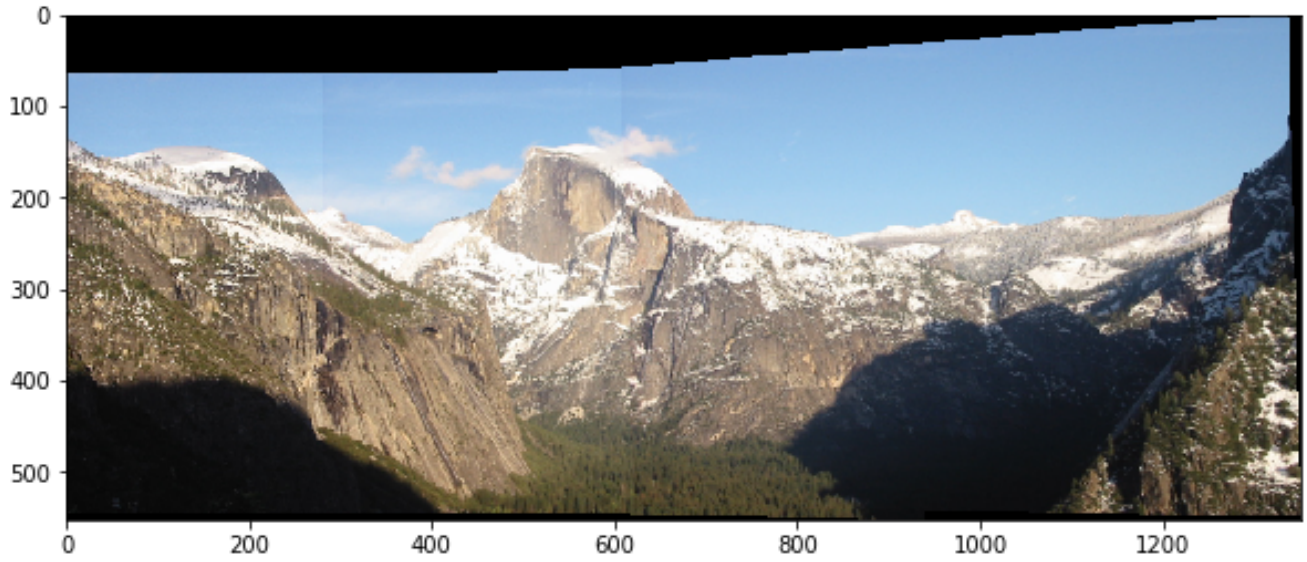
- Lo primero que hago es calcular el tamaño del lienzo final, lo que hago en mi caso es coger de las 3 imágenes, la más grande, y hacer un lienzo que sea esta imágen multiplicada por 3 y con esto asegurar que no podremos salirnos del lienzo.
- Despues, creo la matriz de traslación que en mi caso sólo traslado la imágen en el eje Y para centrarla en el centro del lienzo final. Este proceso se lo realizo a la primera imagen, para después ir rellenando el lienzo hacia la derecha.
- Usando *warpPerspective* posiciono la primera imágen en el centro del lienzo pegada a la izquierda.
- Obtengo los puntos *query* de la primera imagen y los puntos *train* de la segunda imagen que es la que vamos a unir al lienzo.
- Una vez tengo los puntos, calculo la homografía entre estas dos imágenes, y multiplicada por la matriz de traslacion y usando *warpPerspective*, la añadimos al lienzo.
- Para añadir la tercera imagen es exactamente lo mismo pero tenemos que tener en cuenta que tenemos ir acumulando las multiplicaciones de las homografías para así colocar cada imagen en su correcto lugar, por lo que antes de realizar la colocación de esta tercera imagen, calculamos la homografía conjunta y la multiplicamos por la homografía que obtenemos en ese momento y finalmente colocamos la imagen en el lienzo.

Por lo que el código necesario para realizar este mosaico es el siguiente:

```
img1 = leeimagen("yosemite1.jpg",1)
img2 = leeimagen("yosemite2.jpg",1)
img3 = leeimagen("yosemite3.jpg",1)
gray1 = leeimagen("yosemite1.jpg",0)
gray2 = leeimagen("yosemite2.jpg",0)
gray3 = leeimagen("yosemite3.jpg",0)

imgs = [img1,img2,img3]

sift  = cv2.xfeatures2d.SIFT_create()
kp1 = sift.detect(gray1,None)
kp2 = sift.detect(gray2,None)
kp3 = sift.detect(gray3,None)

kp1,des1 = sift.compute(gray1,kp1)
kp2,des2 = sift.compute(gray2,kp2)
kp3,des3 = sift.compute(gray3,kp3)

kps = [kp1,kp2,kp3]

matches12 = BFL2NN(des1,des2)
matches23 = BFL2NN(des2,des3)

matches = [matches12,matches23]

mosaico = mosaico3(imgs,kps,matches)

multIM([mosaico],1,1,10,10)

```

```
input()
```

Y obtenemos como resultado:



4. **(2.5 puntos) Lo mismo que en el punto anterior pero para $N > 5$ (usar las imágenes para mosaico).**

   Para este ejercicio tenemos que hacer lo mismo que para el anterior aunque modificando mínimamente la función.

```
def mosaicoN(imgs,kps,matches,r=True):
    maxx = 0
    maxy = 0
    for im in imgs:
        maxy = im.shape[0] if im.shape[0] > maxy else maxy
        maxx = im.shape[1] if im.shape[1] > maxx else maxx

    mosaico_size = [maxx*len(imgs),maxy*len(imgs)]

    H = np.eye(3)
    H[1][2] = (mosaico_size[1]/2-(maxy/2))

    result = cv2.warpPerspective(imgs[0],H,(mosaico_size[0],mosaico_size[1]))

    for i in range(len(imgs)-1):
        p1 = np.array([kps[i][m.queryIdx].pt for m in matches[i]])
        p2 = np.array([kps[i+1][m.trainIdx].pt for m in matches[i]])
        homography,masc = cv2.findHomography(p2,p1, cv2.RANSAC,1)
        cv2.warpPerspective(imgs[i+1],H.dot(homography),(mosaico_size[0],mosaico_size[1]),dst=result,
    borderMode=cv2.BORDER_TRANSPARENT)
        H = H.dot(homography)

    if(r):
        result = recortar(result)

    return result
```

```
26
```

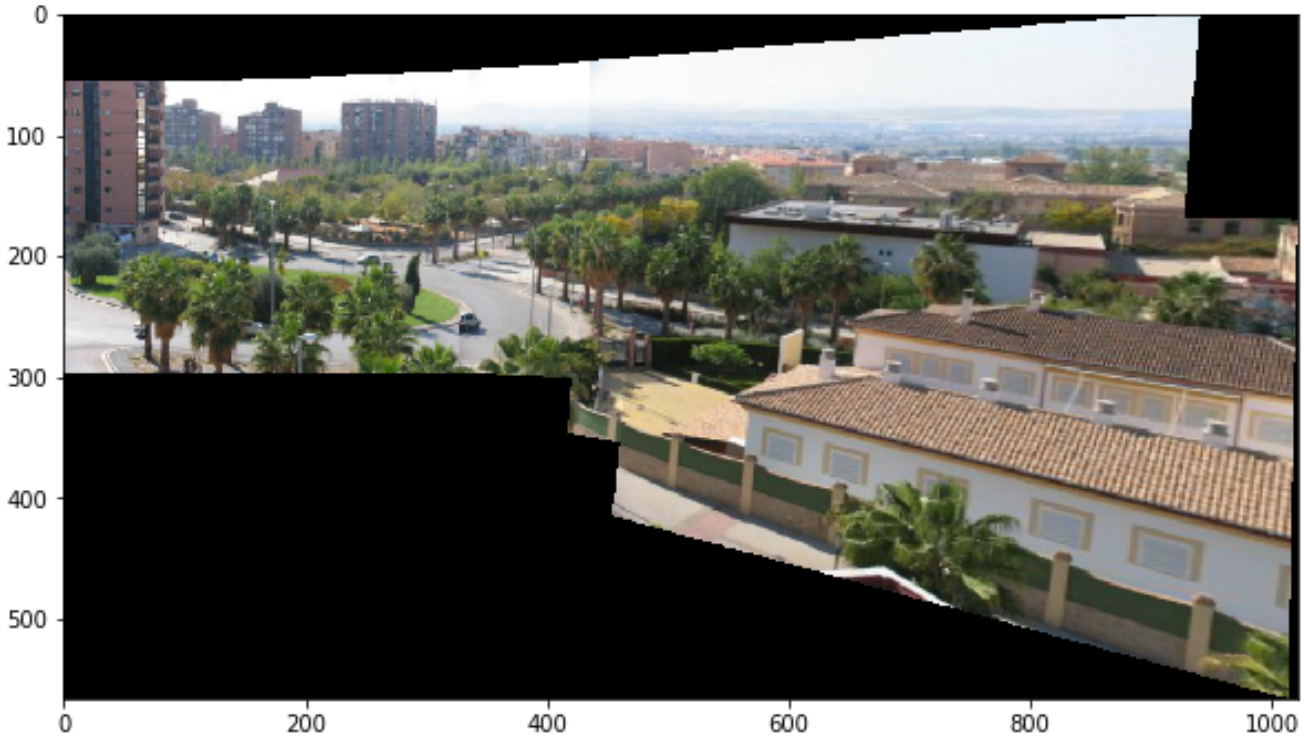
Como vemos las operaciones son exactamente las mismas, solo que ahora en vez de tener en cuenta que tenemos solo 3 imágenes, tenemos más por lo que esa es la única diferencia. Coloco la primera imagen en el centro a la izquierda y ahora voy recorriendo todas y cada una de las imágenes y voy añadiendolas a la derecha de ésta. Al final obtenemos el mosaico con todas las imágenes correctamente.

Como resolución del ejercicio he hecho lo siguiente:

```
img1 = leeimagen("mosaico002.jpg",1)
img2 = leeimagen("mosaico003.jpg",1)
img3 = leeimagen("mosaico004.jpg",1)
img4 = leeimagen("mosaico005.jpg",1)
img5 = leeimagen("mosaico006.jpg",1)
img6 = leeimagen("mosaico007.jpg",1)
img7 = leeimagen("mosaico008.jpg",1)
img8 = leeimagen("mosaico009.jpg",1)
img9 = leeimagen("mosaico010.jpg",1)
img10 = leeimagen("mosaico011.jpg",1)
gray1 = leeimagen("mosaico002.jpg",0)
gray2 = leeimagen("mosaico003.jpg",0)
gray3 = leeimagen("mosaico004.jpg",0)
gray4 = leeimagen("mosaico005.jpg",0)
gray5 = leeimagen("mosaico006.jpg",0)
gray6 = leeimagen("mosaico007.jpg",0)
gray7 = leeimagen("mosaico008.jpg",0)
gray8 = leeimagen("mosaico009.jpg",0)
gray9 = leeimagen("mosaico010.jpg",0)
gray10 = leeimagen("mosaico011.jpg",0)

imgs = [img1,img2,img3,img4,img5,img6,img7,img8,img9,img10]

 sift  = cv2.xfeatures2d.SIFT_create()
kp1 = sift.detect(gray1,None)
kp2 = sift.detect(gray2,None)
kp3 = sift.detect(gray3,None)
kp4 = sift.detect(gray4,None)
kp5 = sift.detect(gray5,None)
kp6 = sift.detect(gray6,None)
kp7 = sift.detect(gray7,None)
kp8 = sift.detect(gray8,None)
kp9 = sift.detect(gray9,None)
kp10 = sift.detect(gray10,None)

kp1,des1 = sift.compute(gray1,kp1)
kp2,des2 = sift.compute(gray2,kp2)
kp3,des3 = sift.compute(gray3,kp3)
kp4,des4 = sift.compute(gray4,kp4)
kp5,des5 = sift.compute(gray5,kp5)
kp6,des6 = sift.compute(gray6,kp6)
kp7,des7 = sift.compute(gray7,kp7)
kp8,des8 = sift.compute(gray8,kp8)
kp9,des9 = sift.compute(gray9,kp9)
kp10,des10 = sift.compute(gray10,kp10)
```

```
kps = [kp1,kp2,kp3,kp4,kp5,kp6,kp7,kp8,kp9,kp10]

matches12 = BFL2NN(des1,des2)
matches23 = BFL2NN(des2,des3)
matches34 = BFL2NN(des3,des4)
matches45 = BFL2NN(des4,des5)
matches56 = BFL2NN(des5,des6)
matches67 = BFL2NN(des6,des7)
matches78 = BFL2NN(des7,des8)
matches89 = BFL2NN(des8,des9)
matches910 = BFL2NN(des9,des10)

matches = [matches12,matches23,matches34,matches45,matches56,matches67,matches78]
matches.extend([matches89,matches910])

mosaico = mosaicoN(imgs,kps,matches)

multIM([mosaico],1,1,10,10)
```

Y como resultado obtengo lo siguiente:



Referencias

[1] OpenCV Reference, `https://docs.opencv.org/3.4.3/`, Accedido el 24 de noviembre de 2018.