

Visión por Computador: Práctica 2

**Detección de puntos relevantes y
construcción de panoramas**

Antonio Manuel Milán Jiménez

21 de noviembre de 2017

Índice

Ejercicio 1	Pag 3
Apartado A	Pag 7
Apartado B	Pag 11
Apartado C	Pag 13
Apartado D	Pag 16
Ejercicio 2	Pag 16
Parte extra	Pag 20
Ejercicio 3	Pag 23
Ejercicio 4	Pag 28

Ejercicio 1

En este primer ejercicio buscaremos la detección de puntos Harris en una imagen a multiescala, es decir, en diferentes niveles de una pirámide Gaussiana.

Para empezar importamos algunas de las funciones que utilizamos en la primera práctica, tales como, imprimir imágenes, construir una pirámide Gaussiana o realizar una convolución con núcleos derivados.

```
import cv2
import numpy as np
from matplotlib import pyplot as plt
import math
from math import sqrt
from PIL import Image
from scipy import signal
import copy

#Función encargada de imprimir una Lista de imágenes en color o escala de grises
def imprimir(imagenes,fila,col,titulos,lista_puntos,lista_segmentos,
grises=False):

    n_img=1
    for img in imagenes:
        for puntos in lista_puntos:

            cv2.circle(img,(int(puntos[1]), int(puntos[0])), 
int(puntos[2]*5), (0,255,0))

            for segmento in lista_segmentos:
                cv2.line(img,segmento[0],segmento[1],(0,0,0))

    plt.subplot(fila,col,n_img)
    plt.subplots_adjust(hspace=0.8)
    if(grises==True):
        plt.imshow(img,cmap='gray')
    else:
        plt.imshow(cv2.cvtColor(img,cv2.COLOR_BGR2RGB))
    plt.title(titulos[n_img-1])

    n_img=n_img+1

plt.show()
```

```

#Función que añade un marco de un tamaño especificado a una imagen
def anadir_marco(img,tam_borde):

    img_mod=np.zeros((img.shape[0]+tam_borde*2,img.shape[1]+tam_borde*2))

    for i in range(tam_borde,img.shape[0]+tam_borde):
        for j in range(tam_borde,img.shape[1]+tam_borde):
            img_mod[i,j]=img[i-tam_borde,j-tam_borde]

    return img_mod

#Función que construye la pirámide Gaussiana de una imagen
def piramide_Gaussiana(img,n):

    lista_img_piramides = []
    img_mod = np.array(img, dtype=np.float32)

    for i in range(1,n+1):
        lista_img_piramides.append(img_mod)
        img_mod=cv2.pyrDown(img_mod)

    return lista_img_piramides

#Función que pone todos los valores de una imagen en positivo y entre 0 y 255
def Reescalar(img,maximos=False,negativos=False):

    if(negativos):
        minimo=np.amin(img)
        img=img+minimo*-1

    maximo=np.amax(img)
    if(maximo>255 or maximos==True):
        coef=(float)(255/maximo)
        img=img*coef

    return img

```

```

#Función que aplica un filtro Gaussiano mediante nucleos separables para
imagenes en escala de grises
def Gauss_kernel_Grey(img,sigma,ind_kernel,kernel,borde):

    img_mod = np.array(img, dtype=np.float32)

    if(ind_kernel==0):
        if(sigma<1):
            tam=7
        else:
            tam=int(sigma*6+1)

    kernel = cv2.getGaussianKernel(tam,sigma)

    for i in range(len(img)):
        img_mod[i]=cv2.filter2D(img[i],cv2.CV_32F,
np.flipud(kernel)).ravel()

    for i in range(len(img[0])):
        img_mod[:,i]=cv2.filter2D(img[:,i],cv2.CV_32F,
np.flipud(kernel)).ravel()

    if(borde!=0):
        img_mod=cv2.copyMakeBorder(img_mod,10,10,10,10,borde)

    return img_mod

#Función que realiza una convolucion con nucleos derivados, en x o y
def Gauss_kernel_deriv(img,ind_deriv,kernel,kernel_no_derivado):

    #Necesario para que no elimine los valores negativos
    img_mod = np.array(img, dtype=np.float32)

    if(ind_deriv ==0):
        for i in range(len(img)):
            img_mod[i]=cv2.filter2D(img[i],cv2.CV_32F,
np.flipud(kernel)).ravel()

        for i in range(len(img[0])):
            img_mod[:,i]=cv2.filter2D(img_mod[:,i],cv2.CV_32F,
np.flipud(kernel_no_derivado)).ravel()

```

```

if(ind_deriv ==1):
    for i in range(len(img_mod[0])):
        img_mod[:,i]=cv2.filter2D(img[:,i],cv2.CV_32F,
np.flipud(kernel)).ravel()

    for i in range(len(img)):
        img_mod[i]=cv2.filter2D(img_mod[i],cv2.CV_32F,
np.flipud(kernel_no_derivado)).ravel()

if(np.amin(img_mod) < 0):           #Hay negativos
    img_mod = Reescalar(img_mod,False,True)

return img_mod

#Función que realiza la convolucion con nucleos derivados
def kernel_derivada(img,ordenX,ordenY,var,borde,sigma):

    img_mod = np.array(img, dtype=np.float32)

    if(var==0):
        kernels=cv2.getDerivKernels(ordenX,0,int(sigma*6+1),
normalize=True)
    else:
        kernels=cv2.getDerivKernels(0,ordenY,int(sigma*6+1),
normalize=True)

    img_mod=Gauss_kernel_deriv(np.array(img,
dtype=np.float32),var,kernels[var],kernels[(var+1)%2])

    if(sigma>1):
        n_kernel=cv2.getGaussianKernel(int(sqrt(sigma-
1)*6+1),int(sqrt(sigma-1)))
        img_mod=Gauss_kernel_Grey(img_mod,int(sqrt(sigma-
1)),1,n_kernel,0)

    if(borde!=0):
        img_mod=cv2.copyMakeBorder(img_mod,10,10,10,10,borde)

return img_mod

```

Apartado A

En este primer apartado calcularemos en sí los puntos Harris de nuestra imagen. Comentábamos al principio que lo realizaremos en multiescala ya que puede ser que en unos niveles de la pirámide detectemos mejor unos puntos Harris que en otros niveles; por ello los calcularemos a lo largo de 5 niveles en una pirámide Gaussiana de la imagen.

Esta es nuestra función que calcula los puntos Harris:

```
#Calcula aquellos puntos que su valor Harris es máximo en un entorno determinado

def calcular_Harris(img,blocksize,ksize,n_piramide,escala,reescalar=True,marco=True,criterio=0.04):

    mi_imagen_harris=cv2.cornerEigenValsAndVecs(img,blocksize,ksize)

    nueva_imagen=np.zeros((mi_imagen_harris.shape[0],
    mi_imagen_harris.shape[1]), dtype=np.float32)

    nueva_imagen=pow((mi_imagen_harris[:, :, 0]*mi_imagen_harris[:, :, 1])-criterio*(mi_imagen_harris[:, :, 0]+mi_imagen_harris[:, :, 1]),2)

    lista_maximos_coordenadas=suprimir_no_maximos(nueva_imagen,blocksize,n_piramide,escala)

    return nueva_imagen,lista_maximos_coordenadas
```

Los puntos Harris los obtenemos mediante la función de OpenCV "**cornerEigenValsAndVecs()**" a la cual le proporcionamos la imagen, blocksize (el tamaño de la ventana) y ksize. Los valores de éstos son 11 y 7, determinados por los sigmas 1.5 y 1 respectivamente.

Dado que queremos quedarnos sólo con los mejores puntos Harris, tenemos que aplicar el criterio de Harris definido como:

$$R = \det(M) - k(\text{trace}(M))^2$$

$$\det(M) = \lambda_1 * \lambda_2$$

$$\text{trace}(M) = \lambda_1 + \lambda_2$$

Este criterio de Harris está definido en la documentación de OpenCV titulada "Harris Corner Detection".

A continuación realizamos una supresión de los valores no máximos, quedándonos así con los píxeles que su valor Harris es máximo local en su entorno (blocksize). Estos máximos los iremos construyendo mediante el formato "(y,x,escala,valorHarris)".

Estas son las funciones utilizadas para dicha tarea, que recorrerán la imagen buscando los máximos locales y suprimiendo a los demás valores de su entorno:

#Calcula el punto máximo en un entorno, comprobando si coincide con el indicado

```
def detectar_maximo(imagen,centroY,centroX,blocksize):  
  
    maximo=np.amax(imagen[centroY-blocksize:centroY+blocksize,centroX-blocksize:centroX+blocksize])  
  
    coincide=(maximo==imagen[centroY,centroX])  
  
    return coincide
```

#Función encargada de poner a 0 todos Los valores de una ventana en La imagen

```
def suprimir_valores(matriz,OrigventanaX,OrigventanaY,tamVX,tamVY):  
  
    matriz[OrigventanaY-tamVY:OrigventanaY+tamVY,OrigventanaX-tamVX:OrigventanaX+tamVX]=0  
  
    return matriz  
  
  
#Función encargada de obtener Los máximos de una imagen y eliminar Los que no lo sean en su entorno  
  
def suprimir_no_maximos(imagen,blocksize,nivel_piramide,escala):  
  
    lista_maximos=[]  
    m_aux=np.ones((imagen.shape[0],imagen.shape[1]))  
    m_aux.fill(255)  
  
    for i in range(blocksize,m_aux.shape[0]-blocksize):  
        for j in range(blocksize,m_aux.shape[1]-blocksize):  
            if(m_aux[i,j]==255):  
                if(detectar_maximo(imagen,i,j,int(blocksize/2))):  
                    m_aux=suprimir_valores(m_aux,j,i,  
int(blocksize/2),int(blocksize/2))  
                    lista_maximos.append([i,j,escala,imagen[i,j]])
```

```

lista_maximos=np.array(lista_maximos)

return lista_maximos

```

Dado que a la hora de visualizar las imágenes nos interesa ver los 500 mejores puntos de toda la pirámide, hemos creado estas dos funciones. La primera se encarga de unir todas las listas de puntos, ordenarlos en función de su valor Harris almacenado y obtener los 500 mejores. Los ordenamos de mayor a menor, pues los puntos con mayor valor Harris, indicarán un cambio de intensidad mayor, lo cual nos resulta más interesante.

La segunda, dado que vamos a visualizar los puntos sobre la imagen original, trata de reescalar las coordenadas de los puntos que fueron calculados en otra escala, multiplicando las coordenadas por dicha escala:

#Función encargada de concatenar varias listas de puntos y ordenarlos en función de su valor Harris

```

def concatenarOrdenar(lista,tam):
    lista_puntos=np.concatenate((lista[0],lista[1]),axis=0)

    for i in range(tam-2):
        if(len(lista[i+2]) > 0):
            lista_puntos=np.concatenate((lista_puntos,lista[i+2]),axis=0)

    lista_puntos=lista_puntos[np.lexsort((lista_puntos[:,3],))]
    lista_puntos=lista_puntos[::-1]
    lista_puntos=lista_puntos[0:500]
    return lista_puntos

```

#Función encargada de reescalar los puntos en función de su escala original

```

def reescalarCord(puntos):

    for i in range(puntos.shape[0]):
        escala=puntos[i,2]
        puntos[i,0]=puntos[i,0]*escala
        puntos[i,1]=puntos[i,1]*escala

    return puntos

```

Esta es la llamada en el "main" que realizamos para este primer apartado y su resultado:

```

mi_imagen=cv2.imread("imagenes/Yosemite1.jpg",0)

mi_lista_imagenes=piramide_Gaussiana(mi_imagen,5)

lista_imagen=[]
lista_puntos_separados=[]
lista_imagen.append(mi_imagen)

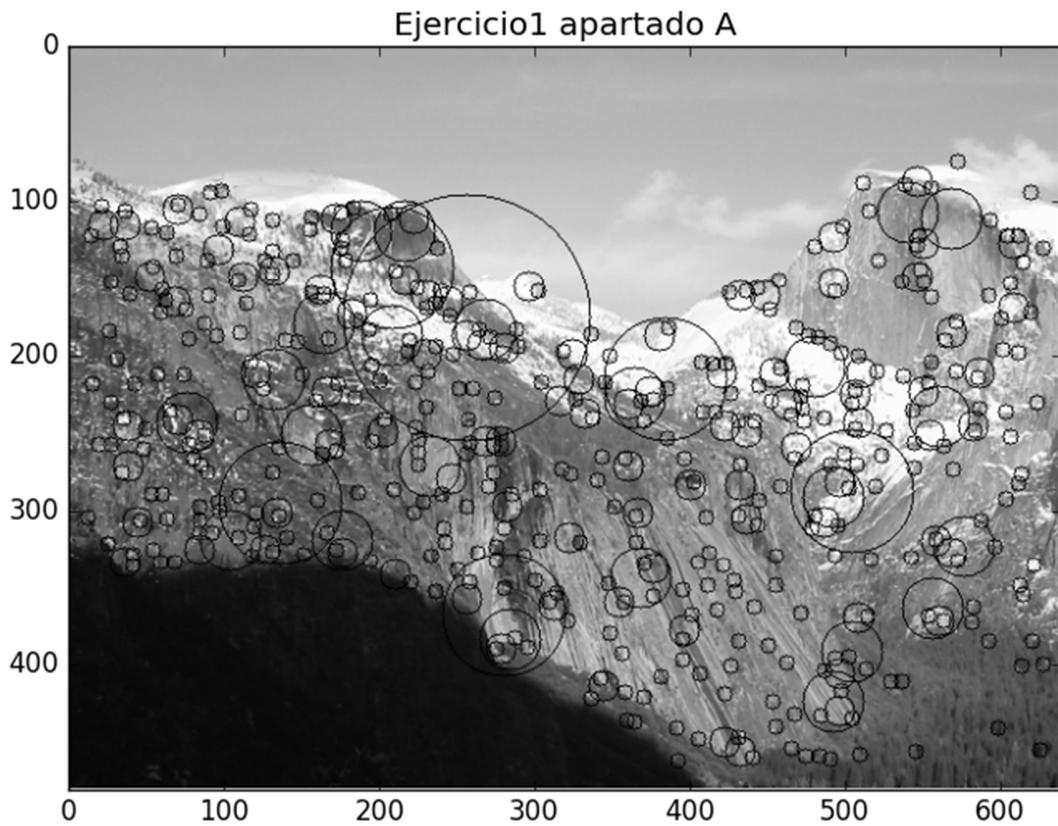
for i in range(5):
    img=calcular_Harris(mi_lista_imagenes[i],blocksize=11,ksize=7,
n_piramide=i+1,escala=pow(2,i))
    lista_puntos_separados.append(img[1])

lista_puntos=concatenarOrdenar(lista_puntos_separados,5)
lista_puntos=reescalarCord(lista_puntos)

titulos=["Ejercicio1 apartado A"]

imprimir(copy.deepcopy(lista_imagen),1,1,titulos,lista_puntos,[],True)

```



Podemos ver que los mejores puntos Harris seleccionados son los que se corresponde a los contornos principales de la imagen, es decir, donde hay una mayor diferencia entre un píxel y otro.

El tamaño del círculo nos indica el nivel de la pirámide en el que se ha detectado, siendo de mayor tamaño los puntos detectados en mayores niveles de la pirámide. Vemos entonces que la mayoría de los puntos Harris seleccionados se detectaron en el primer nivel de la pirámide, donde también tomamos más puntos en consideración.

Apartado B

En este apartado refinaremos la posición calculada de los puntos Harris a nivel de subpíxel mediante la función de OpenCV "**cornerSubPix()**". A esta función le proporcionaremos la imagen , los puntos detectados, el tamaño de ventana determinado por `blocksize`, la "zona cero" y el "criterio". El término criterio se ha establecido en función de lo mostrado en los ejemplos de la documentación de OpenCV titulada "Feature Detection":

#Función que refina La posición de una Lista de puntos de una imagen

```
def refinarPosicion(img,puntos,blocksize):

    if(len(puntos)>0):
        lista_tuplas=[]
        for punto in puntos:
            tupla=(punto[0],punto[1])
            lista_tuplas.append(tupla)

        nuevos_puntos=cv2.cornerSubPix(image=img,
                                         corners=np.array(lista_tuplas,dtype=np.float32),
                                         winSize=(int(blocksize/2),int(blocksize/2)),
                                         zeroZone=(-1,-1),
                                         criteria=(cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER,100,0.001))

        for i in range(len(puntos)):
            puntos[i,0]=nuevos_puntos[i,0]
            puntos[i,1]=nuevos_puntos[i,1]

    return puntos
```

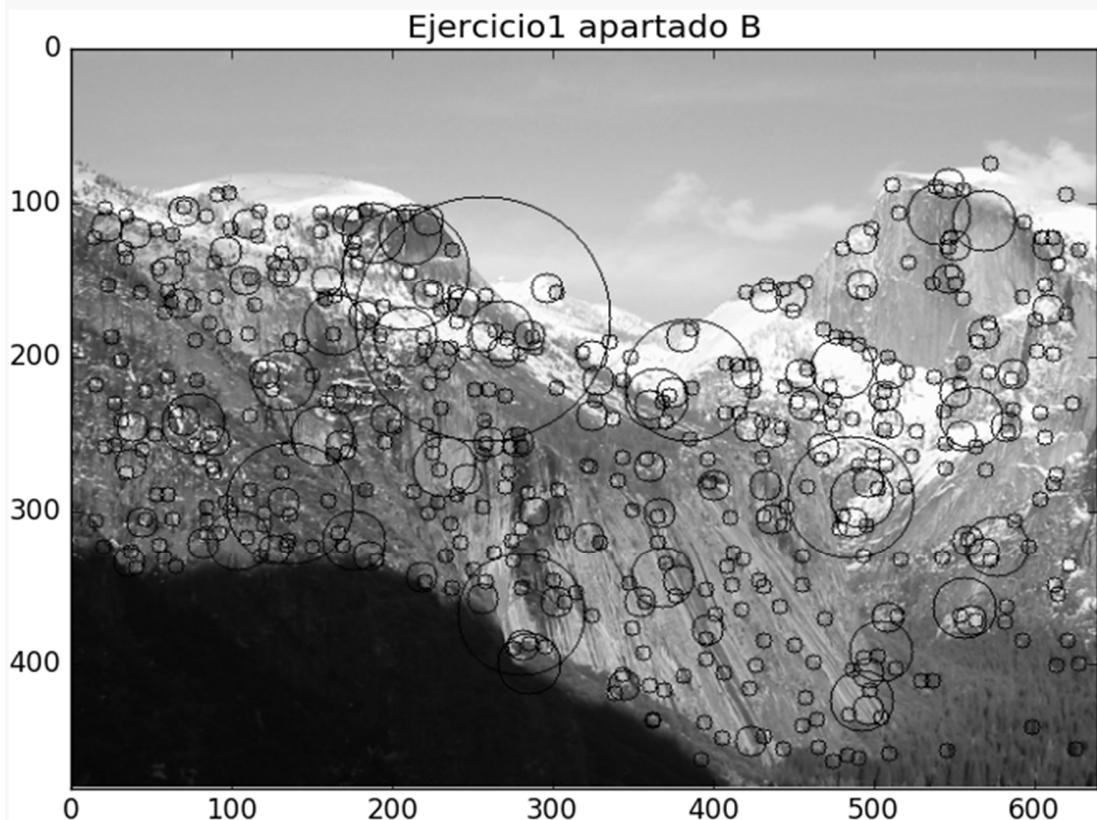
Entonces esta función nos devolverá los puntos con sus posiciones refinadas. Aplicamos la función separadamente a los puntos en función de la escala en la que se hallaron. Esta es la llamada que hacemos y la imagen resultado donde podemos apreciar que la posición de los puntos ha cambiado ligeramente respecto al anterior apartado:

```
for i in range(len(lista_puntos_separados)):
    lista_puntos_separados[i]=refinarPosicion(mi_lista_imagenes[i],
    lista_puntos_separados[i],11)

lista_puntos=concatenarOrdenar(lista_puntos_separados,5)
lista_puntos=reescalarCord(lista_puntos)

titulos=[ "Ejercicio1 apartado B"]

imprimir(copy.deepcopy(lista_imagen),1,1,titulos,lista_puntos,[],True)
```



Son pequeñas variaciones en las coordenadas de los puntos, que siguen encontrándose en los principales contornos de la imagen.

Apartado C

En este apartado calculamos la orientación de cada punto Harris con la que podremos construir finalmente nuestra lista de keypoints.

Primero calculamos las orientaciones de todos los puntos de la imagen. Calculamos para ello la imagen derivada respecto X y respecto Y de nuestra imagen. Indicaremos también el σ con el que hacer la derivada.

A continuación, dividimos la derivada respecto Y entre la derivada respecto X y calculamos la arco tangente de dichos valores. Luego convertimos los ángulos obtenido en radianes a grados. Esta función la aplicaremos también separadamente en función de la escala en la que se detectaron los puntos:

#Función que calcula Las orientaciones de todos los puntos de una imagen

```
def calcular_orientacion(img,sigma):  
  
    imgX=kernel_derivada(img,1,0,0,0,sigma)  
    imgY=kernel_derivada(img,0,1,1,0,sigma)  
  
    angulos=np.arctan(imgY/imgX)  
    angulos=np.degrees(angulos)  
  
    return angulos
```

A continuación tenemos que hacer corresponder todas las orientaciones que hemos obtenido con sólo aquellos puntos detectados que nos interesan. La lista de puntos que proporcionamos es aquella en la que se encuentran ya juntos todos los puntos de las diferentes escalas, ya que, en función de la escala del punto en cuestión, se determinará su orientación en las orientaciones de una imagen u otra de la pirámide.

#Función que encuentra Las orientaciones de una Lista de puntos proporcionada

```
def determinar_angulos(orientaciones,lista_puntos):  
  
    l_angulos=[]  
    for puntos in lista_puntos:  
        n=math.log(puntos[2],2)  
        orient_aux=orientaciones[int(n)]  
        l_angulos.append(orient_aux[int(puntos[0])],int(puntos[1]))  
  
    return l_angulos
```

En la siguiente función calculamos nuestra lista de keypoints. Proporcionando la lista de puntos y ángulos, la función de Opencv "KeyPoint()" irá construyendo los keypoints en el formato "(x,y,escala,angulo)".

#A partir de una Lista de puntos y sus angulos, crea la lista de keypoints

```
def crear_kp(angulos,lista_puntos):

    l_kp=[]
    i=0

    for puntos in lista_puntos:
        kp=cv2.KeyPoint(puntos[1],puntos[0],puntos[2],angulos[i])
        l_kp.append(kp)
        i=i+1

    return np.array(l_kp)
```

Dado que queremos que en la visualización aparezca también un segmento para cada punto indicando su orientación, hemos creado una función encargada de calcular estos segmentos.

Para cada punto, calculamos su "segundo punto" con el que dibujar el segmento. Este segundo punto se calcula de la forma:

$$x' = x + escala * \cos(\text{angulo}), y' = y + escala * \sin(\text{angulo})$$

Creamos entonces una lista de segmentos donde tendremos todos los puntos con sus "segundos puntos". Luego utilizaremos la función "line()" que dibujará una línea entre ambos puntos:

#Función que calcula Los "segundos" puntos, para poder posteriormente dibujar Los segmentos

```
def crear_segmentos(angulos,lista_puntos):

    lista_segmentos=[]

    for i in range(lista_puntos.shape[0]):
        puntoX=lista_puntos[i,1]+lista_puntos[i,2]*5*np.cos(angulos[i])
        puntoY=lista_puntos[i,0]+lista_puntos[i,2]*5*np.sin(angulos[i])
        segmento=[(int(lista_puntos[i,1]),int(lista_puntos[i,0])),(int(puntoX),int(puntoY))]
        lista_segmentos.append(segmento)

    return lista_segmentos
```

Esta es la llamada que hacemos en el "main" y el resultado, donde vemos ahora una línea para cada punto indicando su orientación, desde el centro hacia fuera:

```
orientaciones=[]
for i in range(len(mi_lista_imagenes)):
    orientaciones.append(calcular_orientacion(mi_lista_imagenes[i],
sigma=5))

lista_puntos=concatenarOrdenar(lista_puntos_separados,5)
angulos=determinar_angulos(orientaciones,lista_puntos)
lista_puntos=reescalarCord(lista_puntos)

lista_keypoints=crear_kp(angulos,lista_puntos)
segmentos=crear_segmentos(angulos,lista_puntos)

titulos=[ "Ejercicio1 apartado C"]

imprimir(copy.deepcopy(lista_imagen),1,1,titulos,lista_puntos,segmentos,
True)
```



Apartado D

En este apartado calculamos los descriptores SIFT para cada uno de los puntos.

Para ello simplemente tenemos que crear un objeto SIFT mediante la función "`cv2.xfeatures2d.SIFT_create()`". Entonces, utilizamos la función "`compute()`" proporcionándole la imagen y la lista de keypoints calculada en el apartado anterior, devolviéndonos la lista de descriptores para nuestros puntos.

```
sift=cv2.xfeatures2d.SIFT_create()  
descriptores=sift.compute(mi_imagen,lista_keypoints)
```

Ejercicio 2

Para el segundo ejercicio, calcularemos los matches entre dos imágenes. Los calcularemos con dos criterios de correspondencia diferentes, por fuerza bruta y por Knn.

Este ejercicio lo hemos recogido en la siguiente función:

#Función que calcula Los matches entre dos imágenes

```
def CalculaMatches(mi_imagen1,mi_imagen2,keypoints1,keypoints2,  
keypoints=False):  
  
    sift=cv2.xfeatures2d.SIFT_create()  
  
    if(keypoints==False):  
        keypoints1, descriptores1=sift.detectAndCompute(mi_imagen1,None)  
        keypoints2, descriptores2=sift.detectAndCompute(mi_imagen2,None)  
    else:  
        keypoints1,descriptores1=sift.compute(mi_imagen1,keypoints1)  
        keypoints2,descriptores2=sift.compute(mi_imagen2,keypoints2)  
  
    matcherBF=cv2.BFMatcher(crossCheck=True)  
    matches=matcherBF.match(descriptores1,descriptores2)  
    matches = sorted(matches, key=lambda x:x.distance)  
    img_bf = cv2.drawMatches(mi_imagen1,keypoints1,mi_imagen2,keypoints2,
```

```

matches[0:40],None,flags=2)

#Knn

matcherKnn=cv2.BFMatcher()
matchesKnn=matcherKnn.knnMatch(descriptores1,descriptores2,k=2)

mejores=[]
for a,b in matchesKnn:
    if (a.distance < 0.7*b.distance):
        mejores.append(a)

mejores = sorted(mejores, key=lambda x:x.distance)
img_knn = cv2.drawMatches(mi_imagen1,keypoints1,mi_imagen2,
keypoints2,mejores[0:40],None,flags=2)

if(len(matches)>len(mejores)):
    print("BF " + str(matches[len(mejores)-1].distance))
    print("Knn " + str(mejores[-1].distance))
else:
    print("Knn " + str(mejores[len(matches)-1].distance))
    print("Bf " + str(matches[-1].distance))

return img_bf,img_knn

```

A la función le proporcionamos las dos imágenes y opcionalmente los keypoints ya calculados. En el caso de no proporcionarlos, los calculamos, junto con los descriptores, con la función "**detectAndCompute()**".

Brute-Force

Para la fuerza bruta, creamos un objeto "BFMatcher" con la función de OpenCV del mismo nombre. Le indicamos en la construcción que realice, además, "crossCheck". Ahora, llamamos con este objeto a la función "**match()**" con los dos descriptores de ambas imágenes. Esto nos devolverá los matches que encuentre entre las dos imágenes.

A continuación, ordenamos con "**sorted()**" los matches en función del parámetro de la distancia para que visualicemos sólo los mejores.

Por último, utilizamos la función de OpenCV "**drawMatches()**" a la cual le proporcionamos las dos imágenes, las dos listas de keypoints y los matches que hemos encontrado. Como hemos ordenado los matches, sólo le pasamos los 40 mejores. Esta función nos devuelve una imagen con las dos imágenes concatenadas, con los puntos y matches representados entre ellos mediante líneas. Esta será la imagen que visualizaremos.

Knn

Para el Knn, volvemos a crear un objeto "BFMatcher", aunque esta vez sin indicarle el "crossCheck". Los matches los obtenemos mediante la función "**knnMatch()**" a la que, además de proporcionarle los descriptores de ambas imágenes, le indicamos también $k=2$, para que vaya calculando los dos mejores matches posibles para cada punto.

Dado que ahora, para cada punto tenemos dos posibles matches, tenemos que decidir si el primero de ellos es notablemente mejor que el segundo, ya que de lo contrario, podríamos estar escogiendo el primero cuando el segundo match es el correcto. Por ello en nuestra función realizamos lo siguiente:

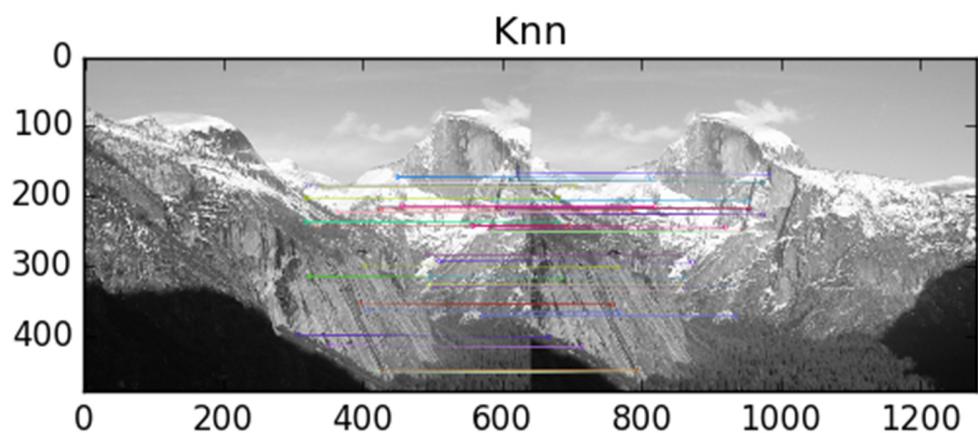
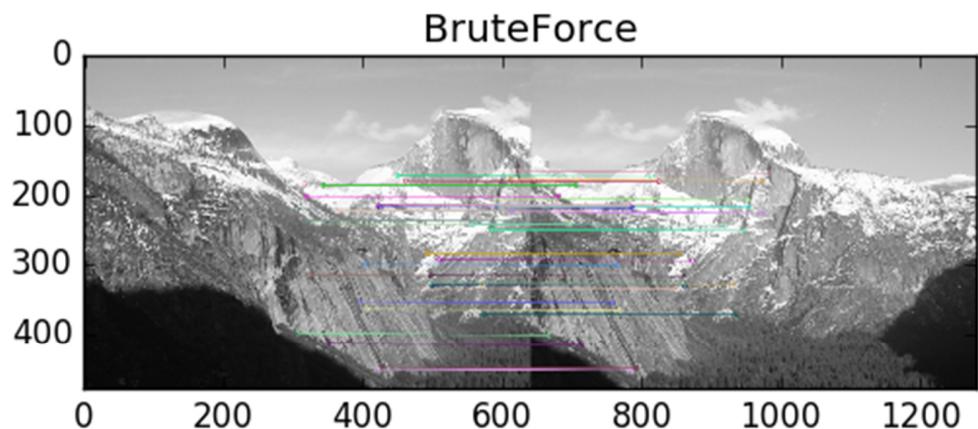
```
for a,b in matchesKnn:  
    if a.distance < 0.7*b.distance:  
        mejores.append(a)
```

Básicamente, tomamos el primer match como correcto sólo si su distancia es al menos un 30% mejor que el segundo match. De lo contrario, desechamos ambos matches.

Una vez hecho esto, ordenamos de nuevo los matches con los que nos hemos quedado en función de su distancia y llamamos a la función "**drawMatches()**" para obtener la imagen que visualizaremos más adelante.

Finalmente, esta es la llamada que realizamos en el "main", junto con la imagen obtenida, con bruteForce y Knn:

```
mi_imagen1=cv2.imread("imagenes/Yosemite1.jpg",0)  
mi_imagen2=cv2.imread("imagenes/Yosemite2.jpg",0)  
  
img_bf,img_knn=CalculaMatches(mi_imagen1,mi_imagen2,[],[])  
  
lista_imagenes_matches=[]  
lista_imagenes_matches.append(img_bf)  
lista_imagenes_matches.append(img_knn)  
  
imprimir(copy.deepcopy(lista_imagenes_matches),2,1,[ "BruteForce", "Knn"],[],[],True)
```



A simple vista, ambas imágenes parecen iguales, con los mismos matches. Es cierto que si nos centramos en los mejores matches que nos devuelve cada uno, los que a priori nos interesan más, no hay ninguna diferencia. Sin embargo, si tomamos los peores matches de cada uno, ahí sí encontramos alguna diferencia. De hecho, si tomamos el último de los matches de Knn y lo comparamos con el correspondiente en los de bruteForce (que sea el mismo índice), obtenemos que para el de Knn la distancia es 242 mientras que para bruteForce es 209.

Esto se traduce en que, si bien para los mejores matches la calidad de ambos es la misma, para los peores podemos decir que bruteForce se comporta algo mejor que el Knn.

Parte extra

Como ejercicio extra, se pedía que se utilizase la lista de keypoints que obtuvimos en el ejercicio 1. Se sigue el mismo proceso que acabamos de describir, sólo que en nuestra función "**CalculaMatches()**", indicamos que sólo se calcule los descriptores con "**compute()**", pues los keypoints ya los hemos obtenido nosotros. No obstante, recogeremos también los keypoints que nos devuelve "**compute()**", ya que en la documentación se indica que puede crear o eliminar algunos keypoints a los que no les haya encontrado descriptores.

Hemos creado una función que a partir de una imagen, nos devuelve su lista de keypoints. Básicamente, hemos recogido el proceso realizado en el ejercicio 1 en una única función:

```
#A partir de una imagen, calcula su Lista de keypoints

def ObtenerKeyPoints(imagen,n_piramide,blocksize,ksize):

    mi_lista_imagenes=piramide_Gaussiana(imagen,n_piramide)

    lista_imagen=[]
    lista_puntos_separados=[]
    lista_imagen.append(imagen)

    for i in range(5):
        img=calcular_Harris(mi_lista_imagenes[i],blocksize,ksize,i+1,
    escala=pow(2,i))
        lista_puntos_separados.append(img[1])

    lista_puntos=concatenarOrdenar(lista_puntos_separados,n_piramide)
    lista_puntos=reescalarCord(lista_puntos)

    for i in range(len(lista_puntos_separados)):
        lista_puntos_separados[i]=refinarPosicion(mi_lista_imagenes[i],
    lista_puntos_separados[i],11)

    lista_puntos=concatenarOrdenar(lista_puntos_separados,5)
    lista_puntos=reescalarCord(lista_puntos)

    orientaciones=[ ]
```

```

for i in range(len(mi_lista_imagenes)):
    orientaciones.append(calcular_orientacion(mi_lista_imagenes[i],
sigma=5))

lista_puntos=concatenarOrdenar(lista_puntos_separados,5)
angulos=determinar_angulos(orientaciones,lista_puntos)
lista_puntos=reescalarCord(lista_puntos)

keypoints=crear_kp(angulos,lista_puntos)

return keypoints

```

Y esta es la llamada que realizamos en el "main", junto con la imagen resultado:

```

mi_imagen1=cv2.imread("imagenes/Yosemite1.jpg",0)
mi_imagen2=cv2.imread("imagenes/Yosemite2.jpg",0)

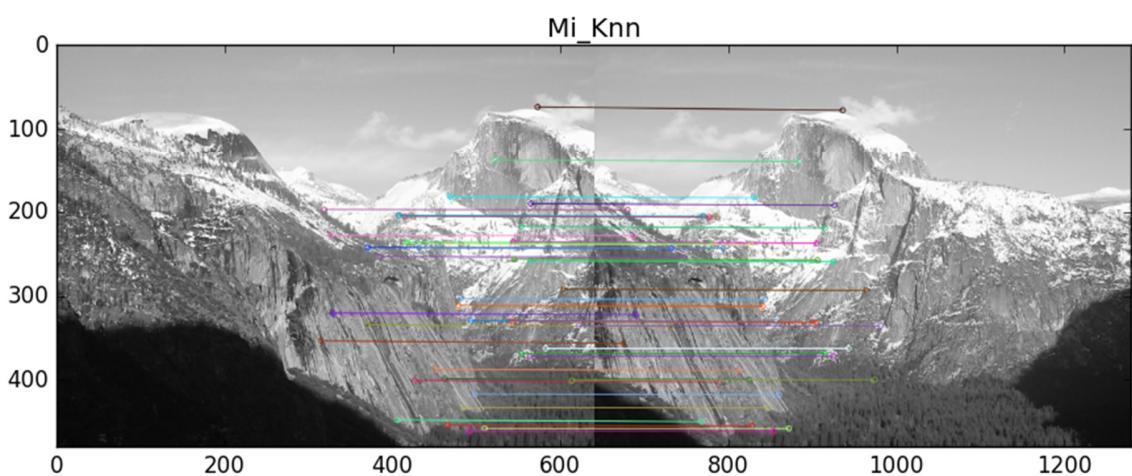
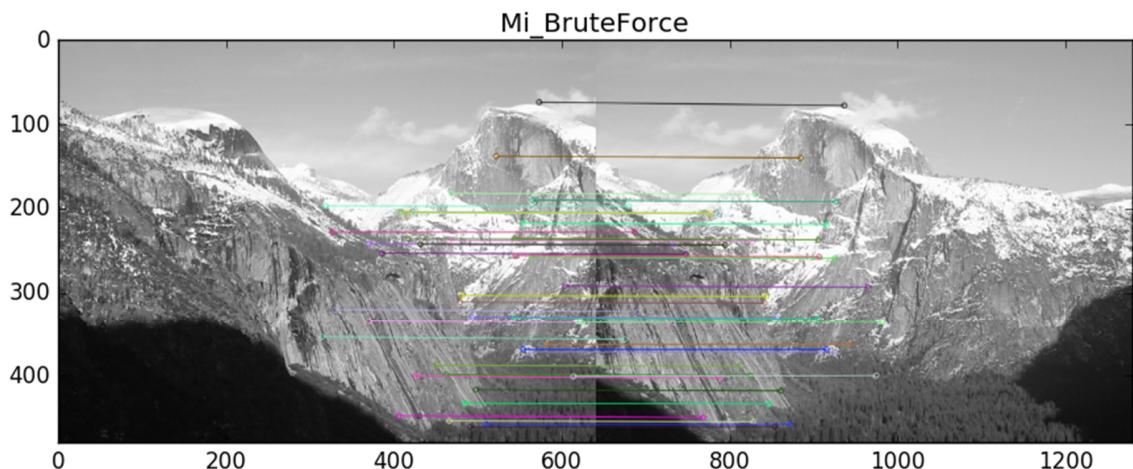
keypoints1 = ObtenerKeyPoints(mi_imagen1,5,11,7)
keypoints2 = ObtenerKeyPoints(mi_imagen2,5,11,7)

mi_img_bf,mi_img_knn=CalculaMatches(mi_imagen1,mi_imagen2,keypoints1,keyp
oints2,True)

lista_imagenes_matches.append(mi_img_bf)
lista_imagenes_matches.append(mi_img_knn)

imprimir(copy.deepcopy(lista_imagenes_matches),2,2,[ "BruteForce", "Knn", "M
i_BruteForce", "Mi_Knn"],[],[],True)

```



Al igual que en el apartado anterior, comparándolos encontramos que bruteForce se comporta algo mejor que Knn para los últimos valores.

Ejercicio 3

En este ejercicio tendremos que construir un mosaico con 3 imágenes. Esta es la función principal en la que nos apoyaremos:

```
#Función que crea el mosaico a partir de una Lista de imágenes

def calcular_mosaico(lista_imagenes):

    img_out = calcular_imgOut(np.array(lista_imagenes))
    img_central=lista_imagenes[int((lista_imagenes.shape[0])/2)]

    centroXimg=int(img_out.shape[1]/2)
    centroYimg=int(img_out.shape[0]/2)
    centroXimgCentral=int(img_central.shape[1]/2)
    centroYimgCentral=int(img_central.shape[0]/2)

    a=(centroXimg-centroXimgCentral,centroYimg-centroYimgCentral)

    h_ini=np.matrix([[1,0,a[0]],[0,1,a[1]],[0,0,1]],dtype=np.float32)
    img_out=cv2.warpPerspective(src=np.array(img_central),M=h_ini,
dsize=(img_out.shape[1],img_out.shape[0]))

    homografias=calcular_homografias(lista_imagenes,h_ini)
    print(len(homografias))

    lista_imagenes=np.delete(lista_imagenes,
int((lista_imagenes.shape[0])/2),axis=0)

    for i in range(lista_imagenes.shape[0]):
        img_out=cv2.warpPerspective(src=np.array(lista_imagenes[i]),
M=(homografias[i]),dst=img_out,
dsize=(img_out.shape[1],img_out.shape[0]),
borderMode=cv2.BORDER_TRANSPARENT)

        img_out=eliminar_filas_columnas(img_out)

    return img_out
```

Para empezar definimos una imagen sobre la que iremos construyendo el mosaico. Dado que a priori no sabemos cuál será el tamaño del mosaico, nos aseguramos definiendo el tamaño como la suma de las dimensiones (X,Y) de todas las imágenes:

```

#Función que crea una imagen con las dimensiones de todas las dimensiones
de todas las imágenes proporcionadas

def calcular_imgOut(lista_imagenes):

    intX=0
    intY=0

    for imagen in lista_imagenes:
        intX=intX+imagen.shape[1]
        intY=intY+imagen.shape[0]

    mi_img=np.zeros((intY,intX))
    mi_img.fill(255)

    return mi_img

```

A continuación, determinamos la imagen central de nuestro mosaico, básicamente la imagen central de la lista de imágenes que se nos proporciona. Sabemos que para llevar esta imagen central al mosaico realizaremos únicamente una traslación por lo que la homografía será :

1	0	x
0	1	y
0	0	1

Los valores "x" e "y" se corresponde con el punto del mosaico dónde se encontrará el (0,0) de la imagen central.

Llamando ahora a "**warpPerspective()**" con la imagen central, esta homografía que acabamos de definir y el tamaño final del mosaico, obtendremos el mosaico con la imagen central ya posicionada.

El siguiente paso será calcular las diferentes homografías que llevarán las diferentes imágenes del mosaico a éste. Las obtendremos con la siguiente función:

```

#Función que calcula las homografías entre las imágenes y el mosaico con
la imagen central

def calcular_homografias(lista_imagenes,h_ini):

    tam=lista_imagenes.shape[0]
    homografias=[]

    for i in range(tam-1):
        ptsA,ptsB = calcular_matches(lista_imagenes[i],
        lista_imagenes[i+1])
        if(i<int(tam/2)):
            homografia, m=cv2.findHomography(ptsA,ptsB,cv2.RANSAC,1)

```

```

    else:
        homografia, m=cv2.findHomography(ptsB,ptsA,cv2.RANSAC,1)
        homografias.append(homografia)

tamHom=len(homografias)

i=int(tamHom/2)
while(i!=0):
    i=i-1
    if((i+1)==int(tamHom/2)):
        homografias[i]=h_ini*homografias[i]
    else:
        homografias[i]=homografias[i+1]*homografias[i]

i=int(tamHom/2)
while(i!=(tamHom)):

    if(i==int(tamHom/2)):
        homografias[i]=h_ini*homografias[i]
    else:
        homografias[i]=homografias[i-1]*homografias[i]
    i=i+1

return homografias

```

Lo primero es ir calculando en un bucle los puntos de los matches entre dos imágenes consecutivas. Hemos creado una función que se encargará calcular estos puntos. Para ello, de igual forma que vimos en el ejercicio 2, calculamos los matches mediante fuerza bruta de dos imágenes dadas. Calculamos por fuerza bruta ya que vimos en el ejercicio 2 que nos daba mejores resultados.

Ya que queremos obtener los puntos de los matches, tenemos que, para cada uno de los matches, "buscarlos" dentro de la lista de keypoints de la forma: keypoints[match.queryIdx] o keypoints[match.trainIdx] si se trata de la imagen "destino". A partir de los keypoints, ya sí obtenemos los valores de sus puntos. Estos valores serán los que devolvamos. Esta es la función en cuestión:

#Calcula Los matches entre dos imágenes y devuelve Los puntos correspondientes

```

def calcular_matches(imgA,imgB):

    sift=cv2.xfeatures2d.SIFT_create()
    keypoints1, descriptores1=sift.detectAndCompute(imgA,None)
    keypoints2, descriptores2=sift.detectAndCompute(imgB,None)

```

```

matcherBF=cv2.BFMatcher(crossCheck=True)
matches=matcherBF.match(descriptores1,descriptores2)

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

matches=matches[0:30]

pts_A=[]
pts_B=[]

for m in matches:

    pts_A.append(np.array(keypoints1[m.queryIdx].pt,dtype=np.float32))

    pts_B.append(np.array(keypoints2[m.trainIdx].pt,dtype=np.float32))

    pts_A=np.array(pts_A).reshape(-1,1,2)
    pts_B=np.array(pts_B).reshape(-1,1,2)

return pts_A,pts_B

```

Una vez que tenemos entonces las dos listas de puntos de los matches entre dos imágenes, calculamos la homografía entre ellas con la función "**findHomography()**" a la que le proporcionamos las dos listas de puntos. Notar que en función de si estamos a la izquierda o a la derecha de la imagen central, calculamos en un sentido u otro la homografía, cambiando el orden de las listas.

Sin embargo, ahora mismo sólo tendríamos las homografías entre imágenes consecutivas, pero nosotros buscamos obtener las homografías que lleven a cada una de las imágenes al mosaico. Estas homografías serán una composición (multiplicando) de todas las homografías que haya entre todas las imágenes que haya entre la imagen en cuestión y la imagen central en el mosaico.

Eso es lo que hacemos en la segunda parte de la función:

```

i=int(tamHom/2)
while(i!=0):
    i=i-1
    if((i+1)==int(tamHom/2)):
        homografias[i]=h_ini*homografias[i]
    else:
        homografias[i]=homografias[i+1]*homografias[i]

i=int(tamHom/2)
while(i!=(tamHom)):
```

```

if(i==int(tamHom/2)):
    homografias[i]=h_ini*homografias[i]
else:
    homografias[i]=homografias[i-1]*homografias[i]
i=i+1

```

Recalculamos las homografías con las homografías que tenga por delante hasta llegar a la imagen central, teniendo en cuenta que estas homografías con las que calculamos tienen que estar ya recalculadas.

En este punto tenemos ya calculadas todas las homografías que llevan a cada una de nuestras imágenes al mosaico.

Ya sólo queda utilizar estas homografías para ir colocando las diferentes imágenes en el mosaico mediante la función "**warpPerspective()**". Entonces a esta función le proporcionamos la imagen en cuestión, la homografía correspondiente y el tamaño de la imagen destino. Además, le indicamos en el parámetro de "borderMode" el valor "**BORDER_TRANSPARENT**" para que mantenga también en el mosaico los imágenes que ya habíamos colocado.

Por último, eliminamos los bordes en negro alrededor del mosaico final, eliminando aquellas filas y columnas que estén completamente en negro:

#Función que elimina las filas y columnas de la imagen que estén completamente a 0

```

def eliminar_filas_columnas(img):

    filas_a_eliminar=[]
    columnas_a_eliminar=[]

    for i in range(img.shape[0]):
        if (np.all(img[i] == [0,0,0])):
            filas_a_eliminar.append(i-len(filas_a_eliminar))

    for i in filas_a_eliminar:
        img=np.delete(img,i,axis=0)

    for i in range(img.shape[1]):
        if (np.all(img[:,i] == [0,0,0])):
            columnas_a_eliminar.append(i-len(columnas_a_eliminar))

    for i in columnas_a_eliminar:
        img=np.delete(img,i,axis=1)

return img

```

Ya sólo tenemos que hacer la llamada en el "main" con nuestras 3 imágenes:

```

lista_imgs=[]

mi_img1=cv2.imread("imagenes/mosaico002.jpg")
mi_img2=cv2.imread("imagenes/mosaico003.jpg")
mi_img3=cv2.imread("imagenes/mosaico004.jpg")

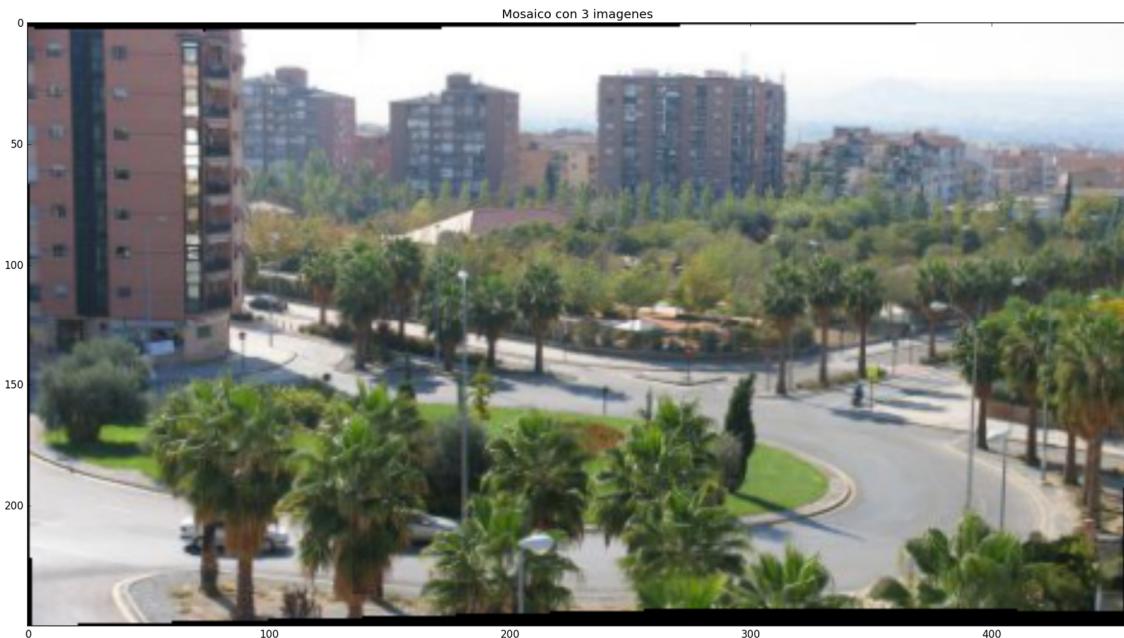
lista_imgs.append(mi_img1)
lista_imgs.append(mi_img2)
lista_imgs.append(mi_img3)

mosaico=calcular_mosaico(np.array(lista_imgs))

imprimir([copy.deepcopy(mosaico)],1,1,['Mosaico con 3
imagenes'],[],[])

```

Y éste es el mosaico resultado:



Ejercicio 4

Gracias a que hemos hecho el ejercicio 3 de forma general para otros tamaños de la lista de imágenes, simplemente en este ejercicio tenemos que llamar a la función que habíamos construido con la lista de 5 imágenes:

```

lista_imgs=[]

mi_img1=cv2.imread("imagenes/mosaico002.jpg")
mi_img2=cv2.imread("imagenes/mosaico003.jpg")
mi_img3=cv2.imread("imagenes/mosaico004.jpg")
mi_img4=cv2.imread("imagenes/mosaico005.jpg")
mi_img5=cv2.imread("imagenes/mosaico006.jpg")

lista_imgs.append(mi_img1)
lista_imgs.append(mi_img2)
lista_imgs.append(mi_img3)
lista_imgs.append(mi_img5)
lista_imgs.append(mi_img4) #Estan en diferente orden

mosaico=calcular_mosaico(np.array(lista_imgs))

imprimir([copy.deepcopy(mosaico)],1,1,[ "Mosaico con 5
imagenes"],[],[])

```

Y aquí tenemos el mosaico que se construye:

