

Visión por Computador (2017-2018)
DOBLE GRADO EN INGENIERÍA INFORMÁTICA Y MATEMÁTICAS
UNIVERSIDAD DE GRANADA

Trabajo 2:
Detección de puntos relevantes y construcción de
panoramas

M^a del Mar Alguacil Camarero

Índice

1. Ejercicio 1	3
2. Ejercicio 2	14
3. Ejercicio 3	19
4. Ejercicio 4	23

1. Ejercicio 1

Detección de puntos Harris multiescala.

- a) Escribir una función que extraiga la lista potencial de puntos Harris a distintas escalas de una imagen de nivel de gris. Para ello construiremos una Pirámide Gaussiana de 5 niveles.

Para la implementación de esta función nos ayudaremos de otras auxiliares para facilitar la estructura y compresión de la función final que nos permite calcular los puntos Harris multiescala de una imagen determinada.

En primer lugar, calculamos los autovalores y autovectores de la matriz de Harris en cada nivel de la pirámide Gaussiana (teniendo en cuenta la imagen original), para ello, usamos la función `cornerEigenValsAndVecs` de OpenCV fijando los valores de `blocksize` y `ksize` equivalentes al uso de máscaras gaussiana de $\sigma_I = 1,5$ y $\sigma_D = 1$ respectivamente, que nos devuelve una imagen con $(\lambda_1, \lambda_2, x_1, y_1, x_2, y_2)$, donde:

- λ_1 y λ_2 son los autovalores de H no ordenados.
- x_1 e y_1 son los autovectores de λ_1
- x_2 e y_2 son los autovectores de λ_2

Para calcular los puntos Harris usando operador Harris:

$$R = \det(H) - k \cdot \text{traza}(H)^2 = \lambda_1 \lambda_2 - k \cdot (\lambda_1 + \lambda_2)^2$$

siendo k es una constante calibrada, $k \in [0,04, 0,06]$, que en este caso tomaremos $k = 0,04$. Y seleccionamos aquellos puntos que son mayores que cero (en este caso, hemos empleado un umbral para evitar el posible ruido de los datos).

Criterio de Harris y $R > 0$

```
# R=det(H)-k·traza(H)2=landa1·landa2-k(landa1+landa2)2
# siendo landa1 y landa2 los autovalores.
def harris(landa1, landa2, k=0.04):
    return landa1*landa2-k*(landa1+landa2)

# Matriz con el valor del criterio selección de Harris asociado a cada pixel
def seleccion(matrix, corner):
    # Matriz con los valores Harris obtenidos
    mharris = []

    # Calculamos los valores Harris correspondientes
    for sextupla in corner:
        c = cv2.split(sextupla)
        mharris.append(harris(c[0], c[1]))
```

```

# Seleccionamos los puntos con R>0(en este caso, R>umbral=0.01) ya que
# estos son los que representan las esquinas
seleccionados = [] # Matriz de 1=True y 0=False (valores de la matriz
# que son mayores que el umbral)

for i in range(len(mharris)):
    # umbral=0.01 para evitar ruido
    seleccionados.append((mharris[i]>0.01)*1)

# Devolvemos dichas matrices
return mharris, seleccionados

```

A continuación se muestra la implementación de la fase de supresión de valores no-máximos. Esta fase del algoritmo elimina como candidatos aquellos píxeles que teniendo un valor alto de criterio Harris no son máximos locales de su entorno para un tamaño de ventana prefijado.

Para la selección de máximos locales nos ayudamos de la función `maxLocal` que tomando como entrada los valores de una ventana se devuelve `True` si el valor del centro es máximo local y `False` en caso contrario. Además se implementa otro función auxiliar `pintarNegro` que dada una imagen binaria modifica a cero todos los píxeles del rectángulo deseado.

Seguidamente implementamos la función `noMaximos` que fijada un tamaño de ventana recorre la matriz binaria creada, en cada posición de valor 255, si el valor del píxel correspondiente de la matriz Harris es máximo local o no. En caso afirmativo, ponemos a cero en la imagen binaria todos los píxeles de la ventana y copiamos las coordenadas del punto central a la lista de salida junto con su escala (nivel de la pirámide).

```

Supresión de valores no-máximos
# Máximo local: tomando como entrada los valores de una ventana devolver
# True si el valor del centro es máximo local y False en caso contrario.
def maxLocal(ventana):
    # Calculamos el máximo de la ventana
    maximo = np.amax(ventana)

    dim = ventana.shape # dimensiones de la ventana

    # Comprobamos si el valor del centro corresponde con el máximo
    if maximo == ventana[int(dim[0]/2), int(dim[1]/2)]:
        return True
    else:
        return False

# Sobre una imagen binaria inicializada a 255 modificamos a 0 todos

```

```

# los píxeles de un rectángulo dado
def pintarNegro(binaria, inicio_row, fin_row, inicio_col, fin_col):
    binaria[inicio_row:fin_row+1, inicio_col:fin_col+1] = 0

# Esta fase del algoritmo elimina como candidatos aquellos píxeles que teniendo
# un valor alto de criterio Harris no son máximos locales de su entorno para un
# tamaño de entorno prefijado (parámetro de entrada).
# mHarris: matriz de Harris
# seleccionados: matriz binaria que vale uno si cumple que el criterio de
#         Harris es mayor que cero y cero en caso contrario
# levels: niveles de la pirámide Gaussiana
# tam: tamaño deseado de la ventana (2*tam[0]+1, 2*tam[1]+1)
def noMaximos(mHarris, seleccionados, levels, tam):
    # Fijamos un tamaño de entorno/ventana y recorremos la matriz binaria ya creada
    # preguntando, en cada posición de valor 255, si el valor del píxel
    # correspondiente de la matriz Harris es máximo local o no
    # En caso negativo, no hacer nada
    # En caso afirmativo, poner a cero en la imagen binaria todos los píxeles
    # de la ventana y copiar las coordenadas del punto central a la lista de
    # salida junto con su escala (nivel de la pirámide).

    # Matrices binarias correspondientes a cada nivel
    matrices = []

    # Información de los puntos que son máximos locales en cada nivel: punto
    # central y escala
    informacion = []

    # Comprobamos si son máximos locales los seleccionados
    for nivel in range(levels):
        # Seleccionamos los elementos de Harris que sean mayores de cero
        # (en este caso, que el umbral prefijado)
        # np.where devuelve un vector con los índices fila y otro con las columnas)
        indices = np.where(seleccionados[nivel] == 1)

        # Dimensiones de la matriz Harris
        dim = mHarris[nivel].shape

        # Matriz de 255's de dimensión igual que la matriz de Harris asociada
        binaria = np.ones(dim, int)*255

        # Eliminamos los puntos que no corresponden a máximos locales
        for i in range(len(indices[0])):

```

```

row = indices[0][i]
col = indices[1][i]

# Si no hemos encontrado un máximo local cercano a él,
# comprobamos si lo es
if( binaria[row,col] == 255 ) :
    inicio_row = max(row-tam[0],0)
    fin_row = min(row+tam[0],dim[0]-1)
    inicio_col = max(col-tam[1],0)
    fin_col = min(col+tam[1],dim[1]-1)

    if maxLocal(mHarris[nivel][inicio_row:fin_row+1,
                                inicio_col:fin_col+1]):
        pintarNegro(binaria, inicio_row, fin_row,
                    inicio_col, fin_col)
    # Guardamos las coordenadas y el nivel correspondiente
    informacion.append([(row,col), nivel])

matrices.append(binaria)

# Devolvemos las matrices binarias y la lista de coordenadas con su
# nivel correspondiente
return matrices, informacion

```

Además implementamos otra función que nos permite dibujar círculos centrados en los puntos Harris con un radio dependiente del nivel de la gaussiana y proporcional a un sigma dado. Permite la visualización de la imagen en color con dichos círculos dibujados en diferentes colores, los cuales dependen del nivel al que pertenezca el centro.

```

# Dibujar sobre la imagen original un círculo centrado en cada punto y de radio
# proporcional al valor del sigma usado para su detección.
def circulos(filename, ptos, sigma=1, orientaciones=[]):
    img = cv2.imread(filename)
    niveles = len(ptos)

    # Colores de los distintos puntos correspondientes a cada nivel
    color=[(0,0,255), (100, 255, 70), (255,0,0),
           (127,127,0), (127,0,127), (0,255,255)]

    for nivel in range(niveles):
        # Pasamos las coordenadas de la escala a las de la imagen original
        if nivel != 0:
            coords = (2**nivel)*np.array(ptos[nivel])

```

```

else:
    coords = ptos[nivel]

    # Dibujamos los círculos y sus orientaciones si disponemos de ellas
    for j,pto in enumerate(coords):
        # Cálculamos el radio dependiendo de la escala y proporcional
        # a sigma
        radio = int((nivel+10)*sigma)

        # Representamos el círculo con centro el pto y radio calculado
        # anteriormente
        cv2.circle(img=img, center=(pto[1],pto[0]), radius=radio,
                   color=color[nivel%len(color)], thickness = 1)

    if len(orientaciones)!=0:
        # Hallamos el ángulo en grados a partir de la orientación obtenida
        angulo = (orientaciones[nivel][j]*180)/np.pi

        # Representamos las líneas correspondientes a la orientación
        cv2.line(img=img, pt1=(pto[1], pto[0]),
                  pt2=(int(pto[1]+np.sin(angulo)*radio),
                        int(pto[0]+np.cos(angulo)*radio)),
                  color=color[nivel%len(color)]))

    # Dibujamos la representación obtenida
    visualization([img], [','], 1, 1, color = True)

```

Y por último creamos la función que nos calcula la lista potencial de puntos Harris a distintas escalas de una imagen de nivel de gris, seleccionando los N puntos de mayor valor, teniendo en cuenta un tamaño de ventana (para asegurarnos de que se impar el tamaño de la ventana será $2 \cdot tam_ventana[0] + 1 \times 2 \cdot tam_ventana[1] + 1$)

```

def ptosHarris(filename, levels=5, tam_ventana=(3,3), sigmaI=1.5,
                sigmaD=1, dibujar=True, N=500):
    # Construimos la pirámide Gaussiana de la imagen
    gauss = pyrGauss(filename, levels=levels, visualize=False)

    # Sobre cada nivel de la pirámide usamos la función de OpenCV
    # cornerEigenValsAndVecs para extraer la información de autovalores y
    # autovectores de la matriz Harris en cada nivel (fijamo valores de
    # blockSize y ksize equivalentes al uso de las máscaras gaussianas de
    # sigmaI y sigmaD respectivamente).
    blockSize=round(6*sigmaI+1)
    ksize=round(6*sigmaD+1)

```

```

corner = []
for i in range(len(gauss)):
    corner.append(cv2.cornerEigenValsAndVecs(gauss[i], blockSize, ksize))

# Usamos uno de los criterios estudiados a partir de los autovalores
# y creamos una matriz con el valor del criterio selección asociado
# a cada pixel (para el criterio de Harris usamos k=0.04).
mharris, seleccionados = seleccion(gauss, corner)

# Implementamos la fase de supresión de valores no-máximos sobre dicha matriz.
matrices, informacion = noMaximos(mharris, seleccionados, levels, tam_ventana)

# Ordenamos de mayor a menor los puntos resultantes de acuerdo a su valor
# arg[1]->nivel
# arg[0]->pto
informacion.sort(key=lambda arg: mharris[arg[1]][arg[0]], reverse=True)

# Seleccionamos los N puntos de mayor valor.
del(informacion[N:len(informacion)])

# Separamos los ptos dependiendo de la escala
informacion.sort(key=lambda arg: arg[1])
ptos = []
ptos_nivel = []
nivel = 0
for info in informacion:
    if nivel!=info[1]:
        ptos.append(ptos_nivel)
        nivel += 1
        ptos_nivel = []

    ptos_nivel.append(info[0])
ptos.append(ptos_nivel)

# Mostramos el resultado dibujando sobre la imagen original un círculo
# centrado en cada punto y de radio proporcional al valor del sigma
# usado para su detección
if dibujar:
    circulos(filename, ptos, sigmaI)



---


return ptos, gauss

```

A continuación la imagen *Yosemite1* con los 500 puntos Harris de mayor valor encontrados con la función explicada e implementada anteriormente.

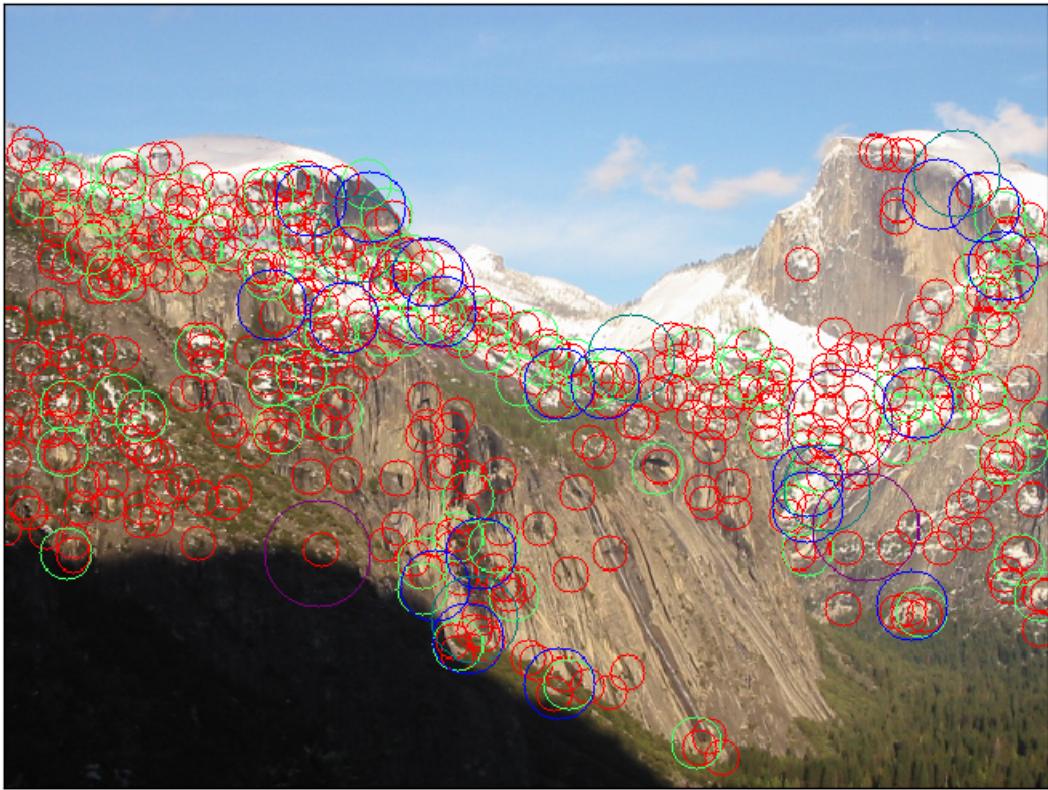


Figura 1.1: 500 puntos Harris de mayor valor con tamaño de ventana 7×7 .

- b) Extraer los valores (cx , cy , escala) de cada uno de los puntos resultantes en el apartado anterior y refinar su posición espacial a nivel sub-píxel usando la función OpenCV `cornerSubPix()` con la imagen del nivel de pirámide correspondiente. Actualizar los datos (cx , cy , escala) de cada uno de los puntos encontrados.

Mejoramos la posición espacial a nivel sub-píxel de los puntos Harris seleccionados en cada uno de los niveles de la pirámide correspondiente donde se encuentren cada uno de ellos, actualizando dichos valores.

```

def refinar(filename, levels=5, tam_ventana=(3,3), sigmaI=1.5,
           sigmaD=1, dibujar=True):
    ptos, gauss = ptosHarris(filename, levels, tam_ventana,
                             sigmaI, sigmaD, False)

    # Refinamos la posición de los puntos
    refinamiento = []
    criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 100, 0.001)

```

```

for i in range(len(ptos)):
    corners = cv2.cornerSubPix(gauss[i], np.float32(ptos[i]), tam_ventana,
                               (-1,-1), criteria)
    refinamiento.append(corners)

#Actualizamos los puntos
ptos = refinamiento

# Dibujamos los círculos en los nuevos puntos refinados
if dibujar:
    circulos(filename, ptos)

return ptos, gauss

```

Obteniendo como resultado:

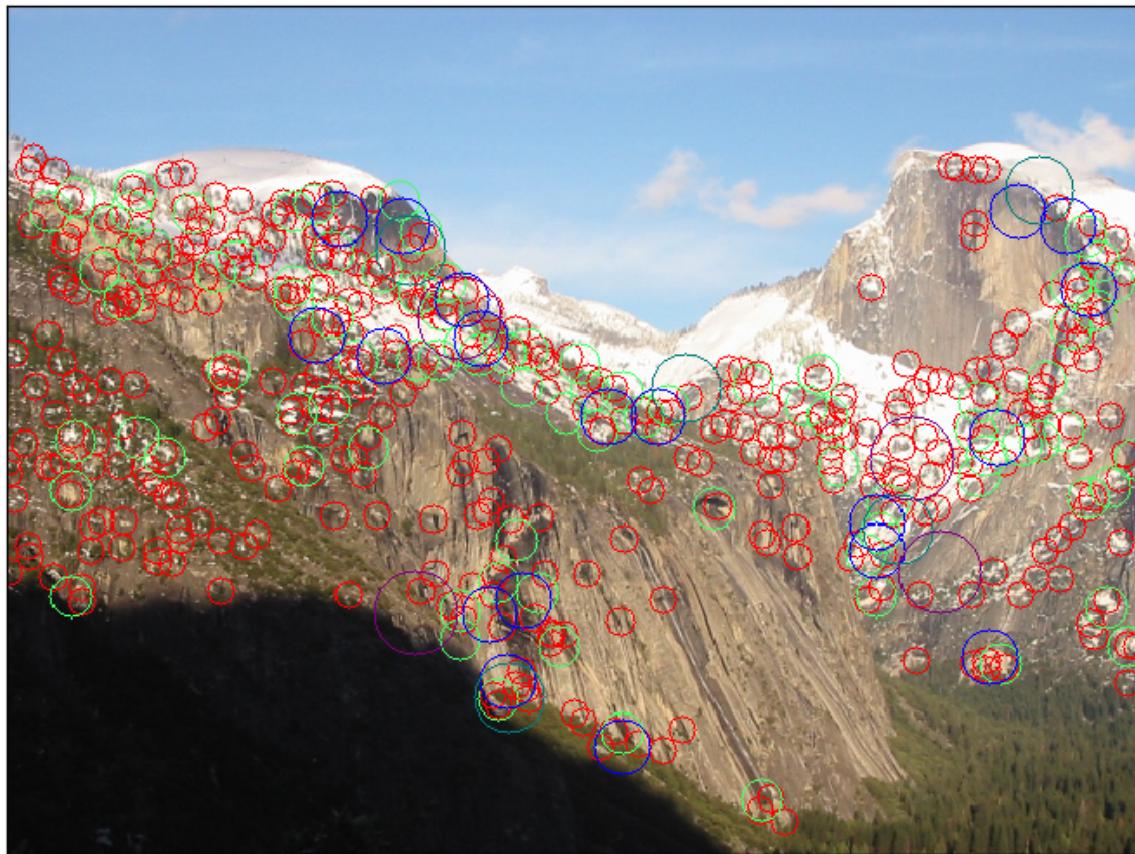


Figura 1.2: Refinamiento de la posición espacial de los puntos Harris anteriores.

- c) Calcular la orientación relevante de cada punto Harris usando el arco tangente del gradiente en cada punto. Previo a su cálculo se deben de aplicar un alisamiento fuerte a las imágenes derivada-x y derivada-y, en la escala correspondiente, como propone el paper *MOPS* de Brown & Szeliski & Winder. (Apartado 2.5) Añadir la información del ángulo al vector de información del punto. Pintar sobre la imagen original círculos con un segmento indicando la orientación estimada en cada punto.

Para calcular la orientación de un punto debemos calcular las derivadas en x y en y de las imágenes. Para ello, aplicamos un kernel gaussiano y las derivadas parciales con $\sigma_0 = 4,5$ a la imagen. Una vez tenemos las derivadas, aplicamos un alisamiento gaussiano con σ_A y calculamos la orientación θ en cada uno de los puntos Harris calculados sabiendo que la derivada en x coincide con $\cos(\theta)$ y la derivada en y con $\sin(\theta)$ basta calcular la arcotangente a partir de dichos valores de la siguiente forma:

$$\theta = \arctan\left(\frac{\partial H/\partial y}{\partial H/\partial x}\right)$$

Siendo el vector correspondiente a la orientación θ , $[\cos(\theta), \sin(\theta)]$.

```
Orientaciones de los puntos Harris
def orientacion(filename, sigma0=4.5, sigmaA=5, levels=5, tam_ventana=(3,3),
                sigmaI=1.5, sigmaD=1, dibujar=True, border_type=cv2.BORDER_REFLECT):
    ptos, gauss = refinar(filename, levels, tam_ventana, sigmaI, sigmaD, False)

    orientaciones = [] # Lista con los ángulos correspondientes a cada escala (nivel)
    for i in range(len(ptos)):
        # Calculamos las imágenes derivada-x y derivada-y
        derX, derY = convol_1derivada(gauss[i], sigma0, border_type=border_type,
                                        visualize=False, read=False)

        # Aplicamos un alisamiento a dichas imágenes
        derX = convol_separable(derX, border_type=border_type, sigma=sigmaA,
                               visualize=False, read=False)
        derY = convol_separable(derY, border_type=border_type, sigma=sigmaA,
                               visualize=False, read=False)

        # Seleccionamos los puntos Harris
        ptos_int = ptos[i].astype(int)
        derHX = []
        derHY = []
        for pto in ptos_int:
            derHX.append(derX[pto[0], pto[1]])
            derHY.append(derY[pto[0], pto[1]])
```

```

# Calculamos la orientación de cada punto Harris usando el arco
# tangente del gradiente en cada punto
orientaciones.append(np.arctan2(derHY, derHX))

# Representamos la imagen con las distintas orientaciones
if dibujar:
    circulos(filename, ptos, orientaciones=orientaciones)

return ptos, gauss, orientaciones

```

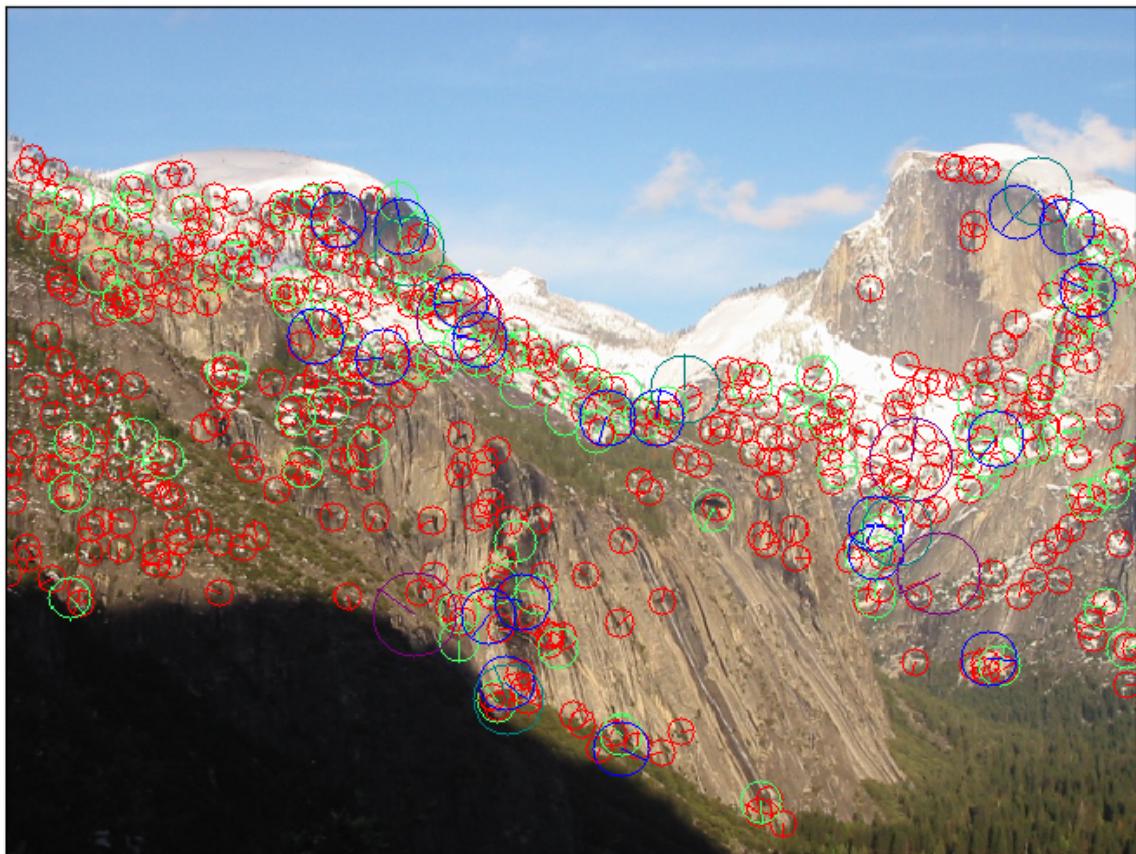


Figura 1.3: Orientaciones de los puntos Harris seleccionados.

- d) Usar el vector de keyPoint extraidos para calcular los descriptores SIFT asociados a cada punto

Guardamos las coordenadas, la escala y el ángulo de cada uno de los puntos Harris

hallados en la estructura KeyPoint, que viene definida en OpenCV, y calculamos los descriptores SIFT asociados a cada punto.

```
def myDetectAndCompute(filename, levels=5, tam_ventana=(3,3), sigmaI=1.5,
                      sigmaD=1, mask=None, dibujar=True):
    ptos, gauss, orient = orientacion(filename, levels=levels,
                                        tam_ventana=tam_ventana, sigmaI=sigmaI,
                                        sigmaD=sigmaD, dibujar=False)

    img = cv2.imread(filename)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Ptos a keyPoints
    keypoints = []
    for nivel, ptos_nivel in enumerate(ptos):
        keyptos = list(map(lambda i: cv2.KeyPoint((2**nivel)*ptos_nivel[i][1],
                                                    (2**nivel)*ptos_nivel[i][0],
                                                    2*tam_ventana[0]+1,
                                                    orient[nivel][i],
                                                    _octave=nivel),
                           range(len(ptos_nivel)))))

        keypoints.append(keyptos)

    keypoints = np.concatenate(keypoints)

    # Aplicamos SIFT con los keypoints calculados
    sift = cv2.xfeatures2d.SIFT_create()
    kpts, dcts = sift.compute(gray, keypoints)

    if dibujar:
        # Dibujamos la imagen con los keypoints
        img_kpts = cv2.drawKeypoints(gray, keypoints, img)
        visualization([img_kpts], [','], 1, 1, color = True)

return kpts, dcts
```

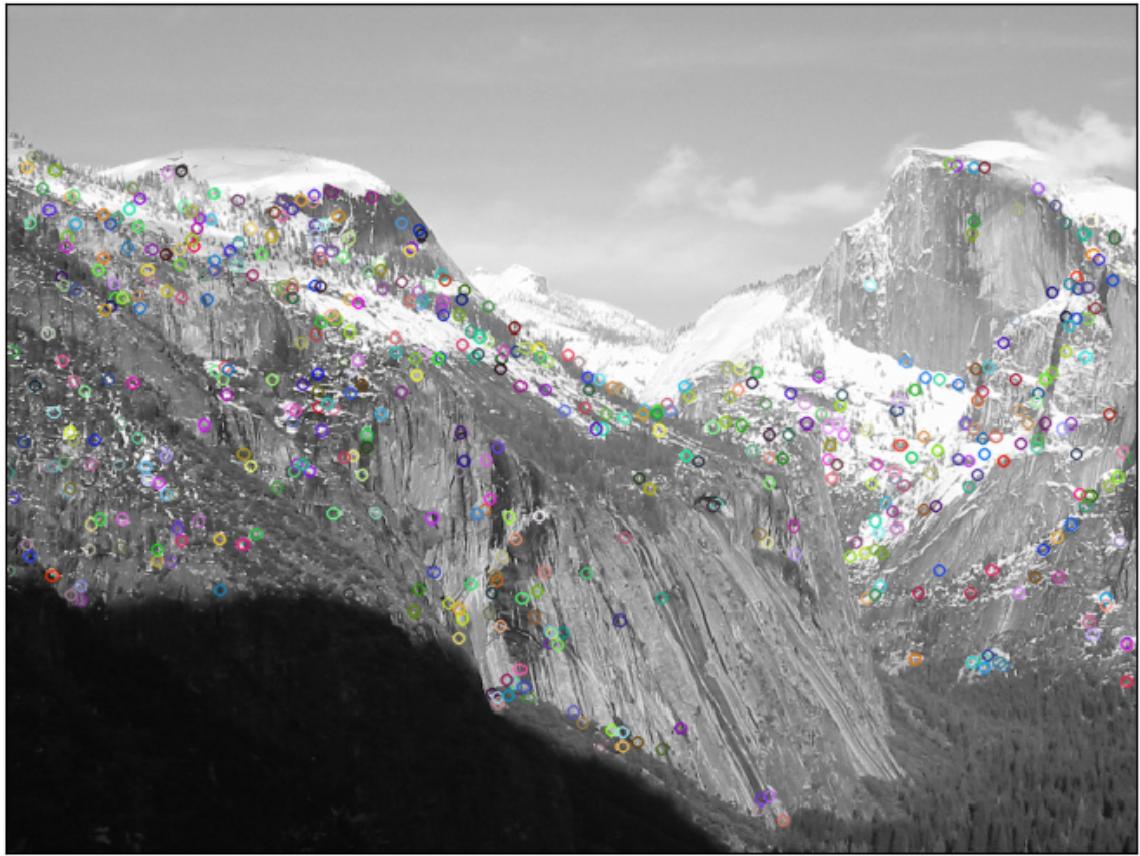


Figura 1.4: KeyPoints.

2. Ejercicio 2

Extraer sus listas de `keyPoints` y descriptores asociados. Establecer las correspondencias existentes entre ellos usando el objeto `BFMatcher` de OpenCV. Valorar la calidad de los resultados obtenidos en términos de correspondencias válidas usando los criterios de correspondencias `BruteForce + crossCheck` y `Lowe-Average-2NN`

Para determinar los emparejamientos existentes entre los distintos puntos Harris de dos imágenes dadas, se emplea el detector SIFT mediante la función implementada en el ejercicio anterior (`ej1=True`) o directamente con las funciones de OpenCV (`ej1=False`), con las cuales extraemos una lista de `KeyPoints` y descriptores. Y se establece las correspondencias `BruteForce` con ayuda de la función `BFMatcher` empleando en los dos caso la

distancia euclídea (

$$|x| = \sqrt{\sum_{k=1}^n |x_k|^2}$$

) con `normType=NORM_L2`, que es el valor que viene por defecto. Para "BruteForce+crossCheck", activamos el parámetro `crossCheck`, calculamos los mejores emparejamientos posibles con `match` y los ordenamos según su distancia para imprimir los N mejores (si $N > 0$, en caso contrario, se dibujan todos). En el segundo caso, utilizamos la función `knnMatch` con $k=2$, y como propone el paper MOPS aceptamos aquellos puntos que cumplan que $e < f \cdot e_{outliers}$.

```
Correspondencias
def emparejarSift(filenames, dibujar=True, leer=True, N=-1, M=-1, f=0.65,
                  mask=None, ej1=False):
    # Leemos y pasamos la imagenes a blanco y negro
    if leer:
        img1 = cv2.imread(filenames[0])
        img2 = cv2.imread(filenames[1])
    else:
        img1 = filenames[0]
        img2 = filenames[1]

    gray1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
    gray2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

    if ej1: # Con ayuda del 1d
        # Detectamos los keypoint y extraemos los descriptores de la foto
        kpts1, dcts1 = myDetectAndComputer(filenames[0], dibujar=False)
        kpts2, dcts2 = myDetectAndComputer(filenames[1], dibujar=False)
    else:
        # Inicializamos SIFT
        sift = cv2.xfeatures2d.SIFT_create()

        # Detectamos los keypoint y extraemos los descriptores de la foto
        kpts1, dcts1 = sift.detectAndCompute(gray1, mask)
        kpts2, dcts2 = sift.detectAndCompute(gray2, mask)

    #BruteForce+crossCheck
    bf = cv2.BFMatcher(crossCheck=True)
    matches1 = bf.match(dcts1, dcts2) # Descriptores de emparejamiento

    # Los ordenamos segun su distancia
    matches1 = sorted(matches1, key = lambda x:x.distance)
```

```

if dibujar:
    if N>=len(matches1) or N<0:
        N=len(matches1)-1

    img_match = cv2.drawMatches(img1,kpts1, img2,kpts2, matches1[:N], None, flags=2)
    visualization([img_match], [','], 1, 1, color = True)

# Lowe-Average-2NN
bf = cv2.BFMatcher()
matches2 = bf.knnMatch(dcts1,dcts2, k=2)

aptos = []
matches = []
for m,n in matches2:
    if m.distance < f*n.distance:
        aptos.append([m])
        matches.append(m)

matches2 = aptos

if dibujar:
    if M>=len(matches2) or M<0:
        M=len(matches2)-1

    img_knnmatch = cv2.drawMatchesKnn(img1,kpts1, img2,kpts2, matches2[:M], None)
    visualization([img_knnmatch], [','], 1, 1, color = True)

# Devolvemos los keyPoints y las correspondencias con Lowe-Average-2NN
return kpts1, kpts2, matches

```



Figura 2.1: *BruteForce+crossCheck* con el detector SIFT de OpenCV.

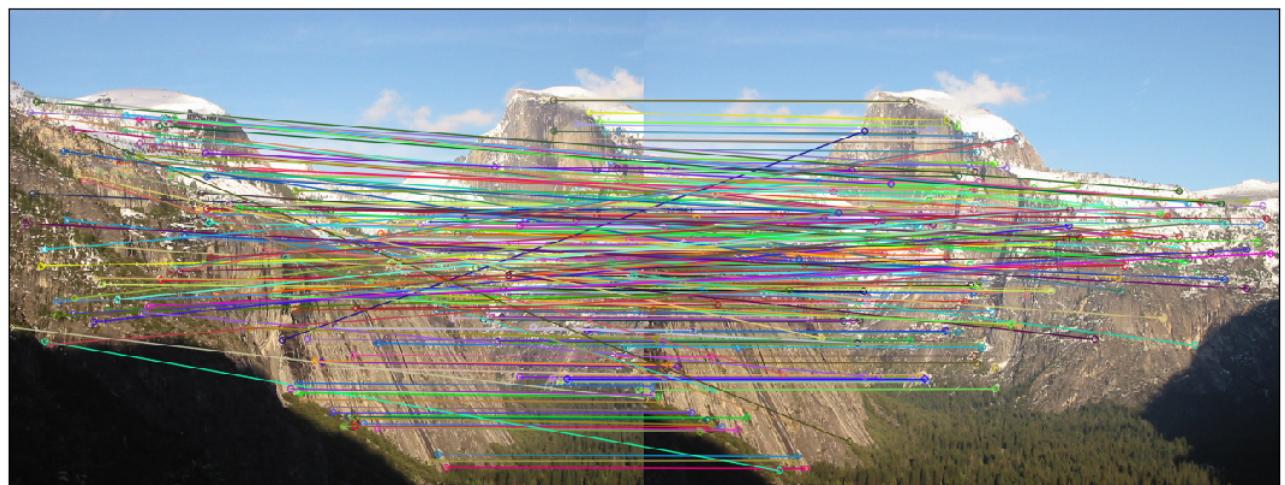


Figura 2.2: *BruteForce+crossCheck* con el detector SIFT implementado en el ejercicio 1.



Figura 2.3: *Lowe-Average-2NN* con el detector SIFT de OpenCV.

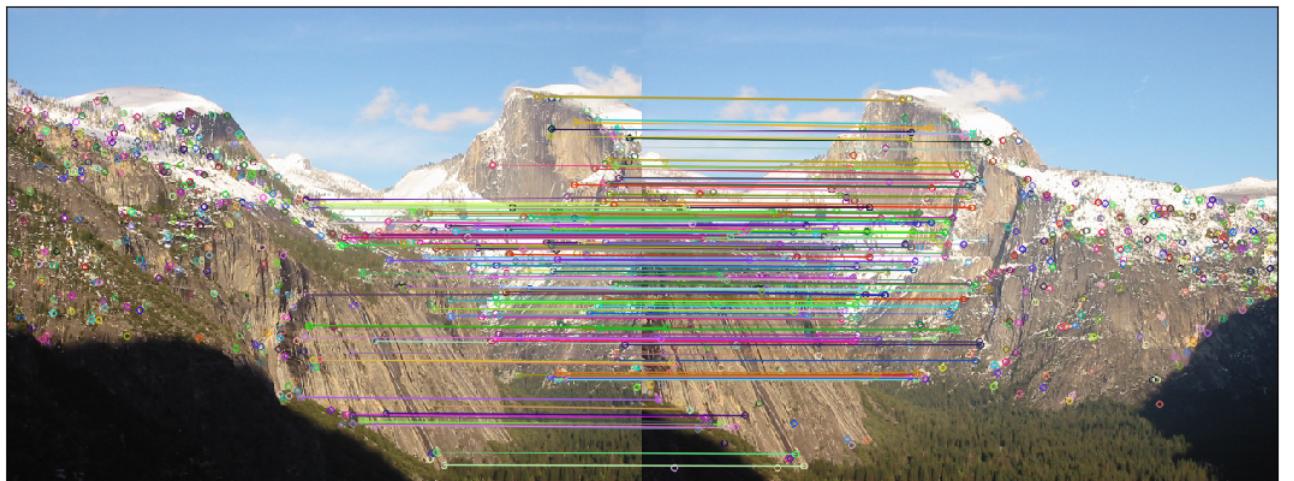


Figura 2.4: *Lowe-Average-2NN* con el detector SIFT implementado en el ejercicio 1.

Se puede observar que *BruteForce + crossCheck* teniendo en cuenta las mejores asociaciones posibles casi no se puede apreciar las imágenes en ambos casos de tantos *outliers* generados, mientras que con *Lowe-Average-2NN* la mayoría de emparejamientos son correctos. Por tanto, sólo se devuelve estas correspondencias.

3. Ejercicio 3

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 crear un mosaico a partir de N=3 imágenes primero definiremos una imagen de dimensión, la suma total de las dimensiones de las tres imágenes. Centraremos la imagen base (que será la que se encuentra en la segunda posición) mediante una traslación, y calcularemos las homografías de la imagen primera y última a la central, y se multiplicará la traslación anterior con dichas homografías para obtener la posición de colocación de cada una de ellas en el mosaico, aplicando dicho movimiento con ayuda de cv2.warpPerspective(). Y, por último, recortaremos los laterales sobrantes.

Construcción del mosaico con 3 imágenes

```
# Calcular la homografía
def homography(filenames, ej1=False):
    # Obtenemos los keypoints y descriptores mediante el detecto-descriptor SIFT
    kpts1, kpts2, matches = emparejarSift(filenames, False, leer=ej1)

    # Estimamos la homografía
    dst_pts = np.float32([kpts1[m.queryIdx].pt for m in matches]).reshape(-1, 1, 2)
    src_pts = np.float32([kpts2[m.trainIdx].pt for m in matches]).reshape(-1, 1, 2)
    homografia, mascara = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 1)

    # Devolvemos dicha homografía
    return homografia

# La imagen central en la lista filename será la que usaremos como base para
# pintar el mosaico
def mosaico3(filenames, dibujar=True, leer=True, ej1=False):
    # Comprobamos que sólo nos pasan 3 imágenes
    assert len(filenames)==3

    # Leemos las imágenes
    image = [cv2.imread(filenames[0]), cv2.imread(filenames[1]),
              cv2.imread(filenames[2])]
    dim = [image[0].shape, image[1].shape, image[2].shape]
    rows = [dim[0][0], dim[1][0], dim[2][0]]
    cols = [dim[0][1], dim[1][1], dim[2][1]]

    # Dimensiones iniciales del mosaico
    nrows = sum(rows)
```

```

ncols = sum(cols)

# Centramos la imagen base mediante una translación
mosaico = np.empty((nrows, ncols, 3))

icentral = 1
H = np.matrix([[1.0,0.0,0.0],[0.0,1.0,0.0], [0.0,0.0,1.0]])
H[1,2] = nrows/2 - rows[icentral]/2
H[0,2] = ncols/2 - cols[icentral]/2

mosaico = cv2.warpPerspective(image[icentral], H, (ncols,nrows))

# Calculamos las homografías de cada imagen con respecto de la central
if ej1:
    Hizq = H*homography([filenames[1], filenames[0]], ej1)
else:
    Hizq = H*homography([image[1], image[0]])
mosaico = cv2.warpPerspective(image[0], Hizq, (ncols,nrows), mosaico,
                               borderMode=cv2.BORDER_TRANSPARENT)

if ej1:
    Hdcha = H*homography([filenames[1], filenames[2]], ej1)
else:
    Hdcha = H*homography([image[1], image[2]])
mosaico = cv2.warpPerspective(image[2], Hdcha, (ncols,nrows), mosaico,
                               borderMode=cv2.BORDER_TRANSPARENT)

mosaico = recortar(mosaico)
visualization([mosaico], [','], 1, 1, color = True)

```

Para recortar eliminamos las filas y columnas que son totalmente nulas, es decir, que todos los elementos de estas son cero en las tres matrices de colores.

```

Recortado de una imagen
# True si el vector es nulo, False en caso contrario
def nula(vector):
    for elemento in vector:
        if elemento!=0:
            return False
    return True

# Devolvemos el primer índice que cumpla que la fila o columna es nula
def nulaInfF(matrix):
    for i in range(matrix.shape[0]):

```

```

    if nula(matrix[i,:]):
        return i

def nulaSupF(matrix):
    n = range(matrix.shape[0])
    for i in n[::-1]:
        if nula(matrix[i,:]):
            return i

def nulaIzqC(matrix):
    m = range(matrix.shape[1])
    for j in m[::-1]:
        if nula(matrix[:,j]):
            return j

def nulaDchaC(matrix):
    for j in range(matrix.shape[1]):
        if nula(matrix[:,j]):
            return j

# Recortamos el mosaico eliminando las filas y columnas nulas de la imagen resultante
def recortar(mosaico):
    # Obtenemos las tres matrices de colores
    b,g,r = cv2.split(mosaico)

    # Dimensiones de las tres matrices
    dim = b.shape

    # Punto central
    fcentral = int(dim[0]/2)
    ccentral = int(dim[1]/2)

    # Obtenemos las filas y las columnas nulas de cada color
    filas_nulasB_sup = nulaSupF(b[:fcentral, :])
    filas_nulasB_inf = fcentral+nulaInfF(b[fcentral:, :])

    filas_nulasG_sup = nulaSupF(g[:fcentral, :])
    filas_nulasG_inf = fcentral+nulaInfF(g[fcentral:, :])

    filas_nulasR_sup = nulaSupF(r[:fcentral, :])
    filas_nulasR_inf = fcentral+nulaInfF(r[fcentral:, :])

    columnas_nulasB_izq = nulaIzqC(b[:,ccentral])

```

```

columnas_nulasB_dcha = ccentral+nulaDchaC(b[:,ccentral:])

columnas_nulasG_izq = nulaIzqC(g[:,ccentral])
columnas_nulasG_dcha = ccentral+nulaDchaC(g[:,ccentral:])

columnas_nulasR_izq = nulaIzqC(r[:,ccentral])
columnas_nulasR_dcha = ccentral+nulaDchaC(r[:,ccentral:])

# Eliminamos las filas y las columnas totalmente nulas
Fsup = min(filas_nulasB_sup, filas_nulasG_sup, filas_nulasR_sup)
Finf = max(filas_nulasB_inf, filas_nulasG_inf, filas_nulasR_inf)

Cizq = min(columnas_nulasB_izq, columnas_nulasG_izq, columnas_nulasR_izq)
Cdcha = max(columnas_nulasB_dcha, columnas_nulasG_dcha, columnas_nulasR_dcha)

mosaico = np.delete(arr=mosaico, obj=range(Finf, dim[0]), axis=0)
mosaico = np.delete(arr=mosaico, obj=range(Fsup), axis=0)

mosaico = np.delete(arr=mosaico, obj=range(Cdcha, dim[1]), axis=1)
mosaico = np.delete(arr=mosaico, obj=range(Cizq), axis=1)

# Devolvemos el mosaico recortado
return mosaico

```

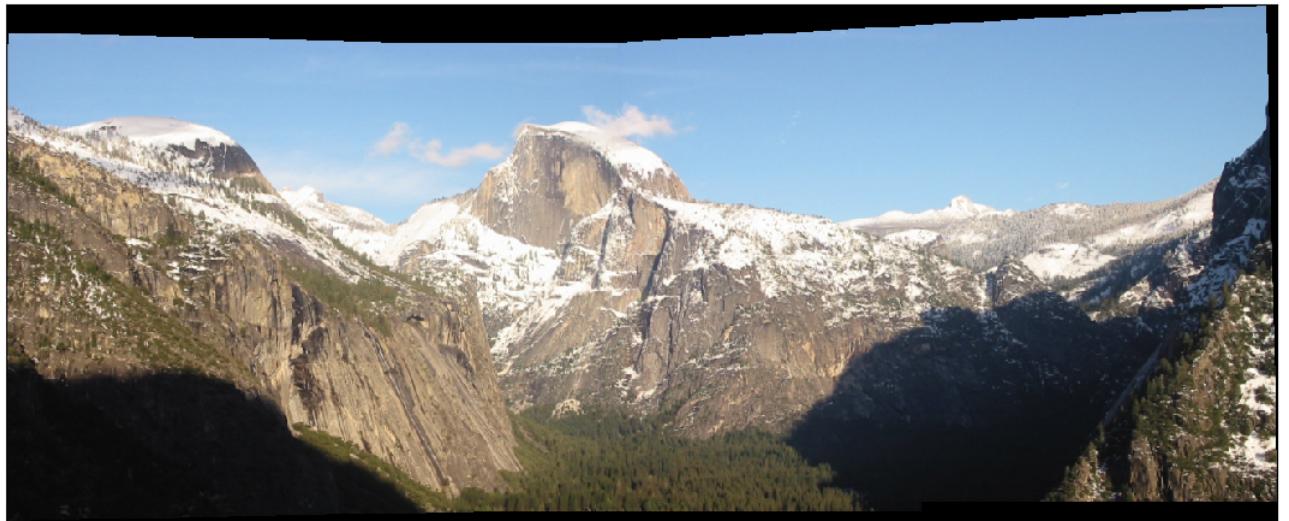


Figura 3.1: Mosaico con el detector SIFT de OpenCV.

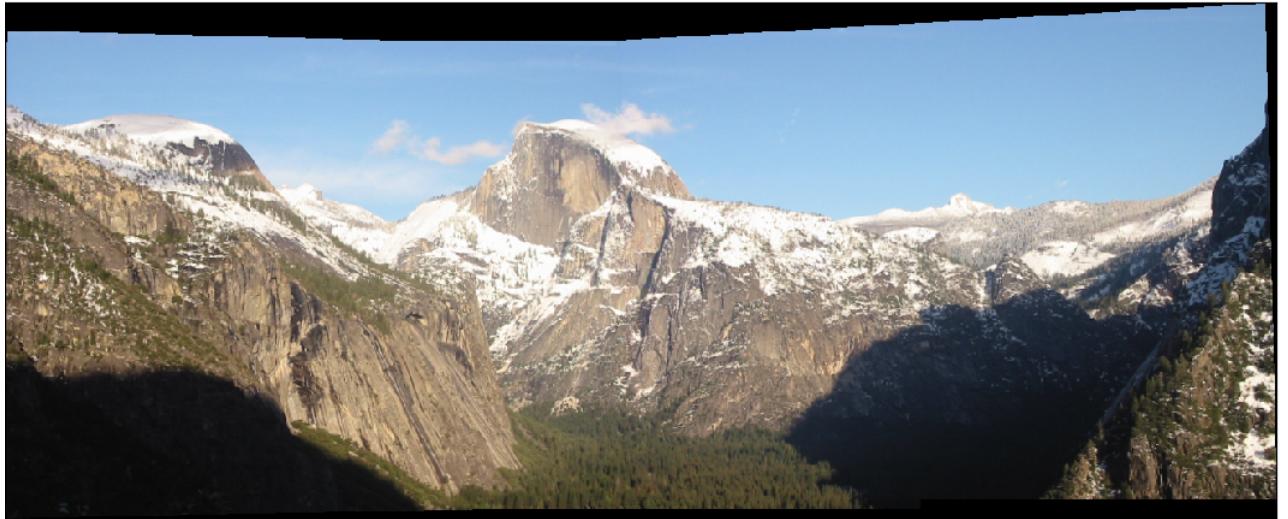


Figura 3.2: Mosaico con el detector SIFT implementado en el ejercicio 1.

4. Ejercicio 4

Lo mismo que en el punto anterior pero para $N > 5$.

Análogo al ejercicio anterior, pero teniendo en cuenta que la imagen base será la que se encuentra en la posición central de la lista que contiene dichas imágenes, y calculando las homografías entre sus dos vecinas más cercanas, es decir, entre las imágenes que tiene a derecha e izquierda en la lista respectivamente.

```
Mosaico con N>3
def mosaicoN(filenames, dibujar=True, leer=True):
    # Comprobamos que nos pasan más de 3 imágenes
    assert len(filenames)>3

    # Leemos las imágenes
    image = []
    rows = []
    cols = []
    for img in filenames:
        image.append(cv2.imread(img))
        dim = image[-1].shape
        rows.append(dim[0])
        cols.append(dim[1])

    # Dimensiones iniciales del mosaico
```

```

nrows = sum(rows)
ncols = sum(cols)

# Creamos la matriz para la base del mosaico
mosaico = np.empty((nrows, ncols, 3))

# Centramos la imagen base mediante una translación
icentral = ceil(len(image)/2)

H = np.matrix([[1.0,0.0,0.0],[0.0,1.0,0.0], [0.0,0.0,1.0]])
H[1,2] = nrows/2 - rows[icentral]/2
H[0,2] = ncols/2 - cols[icentral]/2

mosaico = cv2.warpPerspective(image[icentral], H, (ncols,nrows))

# Calculamos las homografías de cada imagen con cada una de sus vecinas
# y las colocamos en el lugar correspondiente.
Hizq = H
Hdcha = H
for i in range(icentral):
    if (icentral-i)>0:
        Hizq = Hizq*homography([image[icentral-i], image[icentral-i-1]])
        mosaico = cv2.warpPerspective(image[icentral-i-1], Hizq,
                                       (ncols,nrows), mosaico,
                                       borderMode=cv2.BORDER_TRANSPARENT)
    if (icentral+i+1)<len(image):
        Hdcha = Hdcha*homography([image[icentral+i], image[icentral+i+1]])
        mosaico = cv2.warpPerspective(image[icentral+i+1], Hdcha,
                                       (ncols,nrows), mosaico,
                                       borderMode=cv2.BORDER_TRANSPARENT)

# Recortamos la parte sobrante del mosaico
mosaico = recortar(mosaico)

# Visualizamos el resultado final
visualization([mosaico], [','], 1, 1, color = True)

```

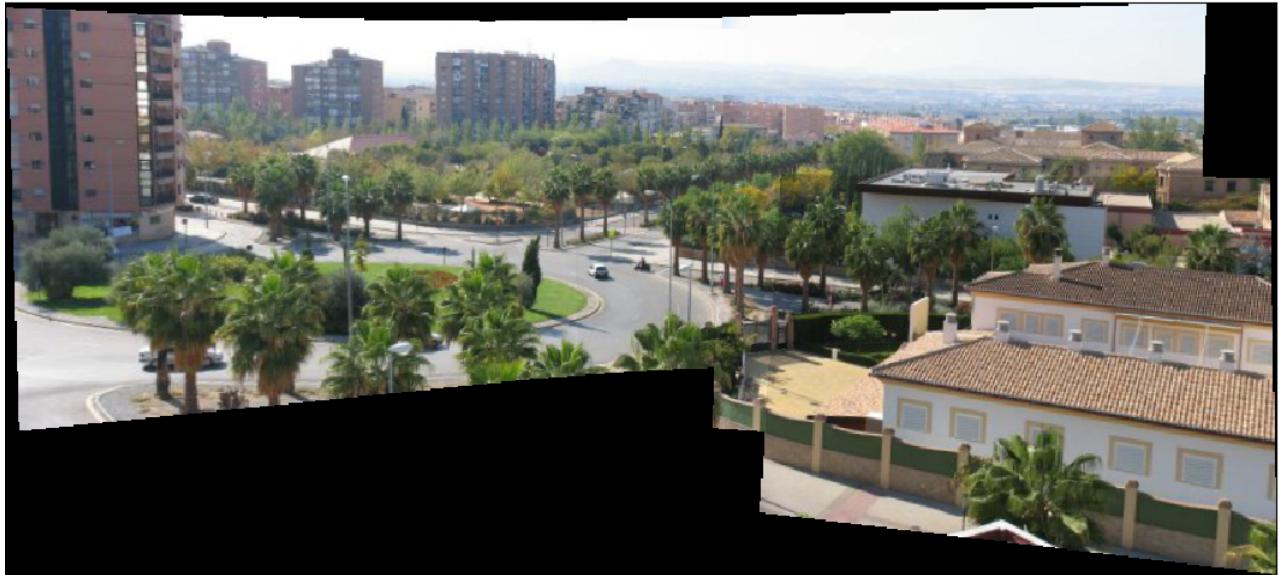


Figura 4.1: Mosaico con el detector SIFT de OpenCV.

Como podemos observar las distintas imágenes se han colocado de manera correcta pero hay pequeños saltos de color y pequeñas distorsiones debido al solapamiento de una imágenes sobre otras. Esto se podría corregir con un filtro de alisamiento pero podríamos perder detalles ya que se eliminarían frecuencias altas.