

tarea2_diego_irarrazaval

October 16, 2020

1 Tarea 2: Calculo de puntos de interes.

2 Diego Irarrazaval

2.1 Introduccion:

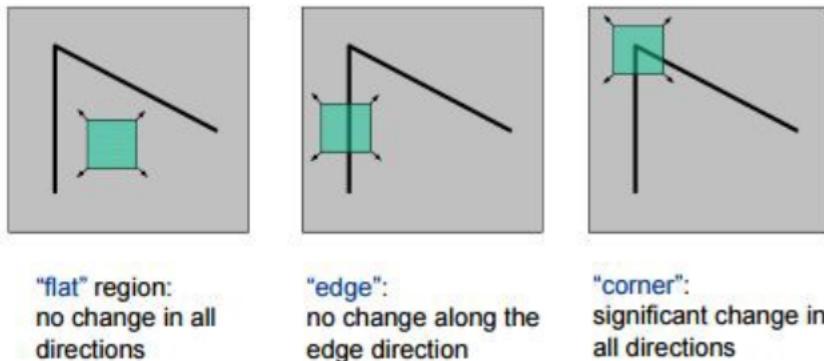
En la siguiente tarea, se estudia el calculo de puntos de interes y la alineacion de imagenes. Los detectores de esquinas o puntos de interes, son muy utilizados en *Computer Vision* para extraer caracteristicas.

En el presente informe, se hace un breve recorrido teorico de los detectores de *Harris* y *ShiTomasi*, luego sobre la generacion de *matches* (o correspondencias) usando descriptores y una breve descripcion de la trasformacion homografia. La siguiente seccion consiste en la implementacion de los detectores y la alineacion de imagenes. Finalmente, se procede a la conclusion.

2.2 Marco Teorico:

2.2.1 Detectores:

El principio de los detectores es basicamente, tomando una ventana de la imagen, observar la diferencia al trasladar esta en alguna direccion. Esto fue propuesto por Moravec (aproximadamente en 1980):



EN la imagen, se observa la ventana de color verde que se mueve por las distintas posiciones de la imagen para detectar las diferencias en el eje x o en el eje y. Esto se traduce en la siguiente ecuacion:

$$E(x, y; \Delta x, \Delta y) = \sum_{x,y} w(x, y) (I(x + \Delta x, y + \Delta y) - I(x, y))^2 \quad (1)$$

Donde se observa en la ecuacion anterior que si la ventana observa lo mismo que al desplazarse, ese punto no sera identificado como punto de interes.

De la ecuacion anterior, se obtienen los siguientes “momentos”:

$$I_1(x, y) = E(x, y, 1, 0) \quad (2)$$

$$I_2(x, y) = E(x, y, 0, 1) \quad (3)$$

$$I_3(x, y) = E(x, y, -1, 0) \quad (4)$$

$$I_4(x, y) = E(x, y, 0, -1) \quad (5)$$

Como nos interesa obtener esquinas, estas se detectaran con:

$$I_{min}(x, y) = \min\{I_1, I_2, I_3, I_4\}, \quad (6)$$

buscando los maximos puntos de interes I_{min} .

Detector de Harris: El detector de Harris corresponde a una mejora del detector de Moravec: Para hacerlo menos susceptible al ruido, la ventana utilizada es *gaussiana* y las diferencias en el eje x e y se transforman en derivadas:

$$w(x, y; \sigma_I) = \mathcal{N}(x, y, \sigma_I) \quad (7)$$

$$L(x, y, \sigma_D) = I(x, y) * \mathcal{N}(x, y, \sigma_D). \quad (8)$$

En la ecuacion anterior, $L(x, y, \sigma_D)$ corresponde a la imagen suavizada.

Luego, se debe expresar $E(x, y; \Delta x, \Delta y)$ con las derivadas parciales de la imagen:

$$E(x_0, y_0; \Delta x, \Delta y) = (\Delta x, \Delta y) * \mu_{x_0, y_0} * \left(\begin{matrix} \Delta x \\ \Delta y \end{matrix} \right) \quad (9)$$

$$\mu_{x_0, y_0} = \mathcal{N}(x - x_0, y - y_0, \sigma_I) * \begin{pmatrix} L_{xx} & L_{xy} \\ L_{yx} & L_{yy} \end{pmatrix}, \text{ con :} \quad (10)$$

$$L_{xx} = \frac{\partial L}{\partial x} \frac{\partial L}{\partial x}, \quad L_{yy} = \frac{\partial L}{\partial y} \frac{\partial L}{\partial y}, \quad L_{xy} = \frac{\partial L}{\partial x} \frac{\partial L}{\partial y}, \quad L_{yx} = \frac{\partial L}{\partial y} \frac{\partial L}{\partial x} \quad (11)$$

De esta forma, se desea elegir los puntos donde la energia $E(x_0, y_0)$ sea grande en cualquier direccion de desplazamiento. Es importante destacar que la matriz μ_{x_0, y_0} es diagonalizable. Como solo nos interesa el contenido de la matriz, los puntos de interes se encontraran con los valores propios de esta:

$$cornerness(x, y) = \min\{\lambda_1(x, y), \lambda_2(x, y)\} \quad (12)$$

$$\text{Punto de interes} = \max\{cornerness(x, y)\} \quad (13)$$

Para el caso particular de la tarea, se utilizara lo siguiente:

$$cornerness(x, y) = \lambda_1 * \lambda_2 - 0.04 * (\lambda_1 + \lambda_2)^2, \quad (14)$$

$$cornerness(x, y) = det - 0.04 * tr^2, \quad \text{donde} \quad (15)$$

$$det = m_{xx} * m_{yy} - m_{xy}^2 \quad y \quad tr = m_{xx} + m_{yy} \quad (16)$$

Detector de ShiTomasi: El detector de *ShiTomas* es igual en el sentido que la imagen de entrada se suaviza con un filtro *gaussiano* y se utilizan las derivadas en cada direccion para encontrar puntos de interes. La diferencia esta en el calculo de *cornerness*:

$$cornerness(x, y) = \lambda_1, \quad (17)$$

$$cornerness(x, y) = \frac{tr}{2} - \frac{\sqrt{tr^2 - 4 * det}}{2} \quad (18)$$

2.2.2 Correspondencia utilizando descriptores:

En primer lugar, los descriptores corresponden a informacion relevante que “rodea” un punto de interes. En una imagen de montanas, pueden ser las cumbres de estas, etc... Los mejores descriptores, son invariantes a la posicion. De esta forma, independiente de la posicion o rotacion de la imagen, se puede identificar el mismo punto de interes en dos imagenes distintas.

```
[ ]: import os.path
try:
    import google.colab as colab
    IN_COLAB = True
except:
    IN_COLAB = False

# Cargamos los datos solo la primera vez que nos conectamos a la "runtime":
# La descarga se hace automatica desde un repo en github
if IN_COLAB:
    if os.path.exists('/content/imagenes_tarea_2.zip'):
        print("Datos ya descargados")
    else:
        !wget https://github.com/Diego-II/Procesamiento-Avanzado-de-Imagenes/raw/
        ↪master/Tarea2/imagenes_tarea_2.zip
        !unzip /content/imagenes_tarea_2
```

```
[ ]: !ls
```

```
imagenes_parte_2  imagenes_parte_3  imagenes_tarea_2.zip  sample_data
```

```
[ ]: !pip install ipython-autotime
```

```
[ ]: %load_ext autotime
```

The autotime extension is already loaded. To reload it, use:

```
%reload_ext autotime
```

time: 2.03 ms

```
[ ]: !pip install -U opencv-python
```

Requirement already up-to-date: opencv-python in /usr/local/lib/python3.6/dist-packages (4.4.0.44)

Requirement already satisfied, skipping upgrade: numpy>=1.13.3 in /usr/local/lib/python3.6/dist-packages (from opencv-python) (1.18.5)
time: 2.23 s

Parte 2

```
[ ]: %load_ext Cython
```

The Cython extension is already loaded. To reload it, use:

```
%reload_ext Cython
```

time: 1.09 ms

Para el calculo de los gradientes, se utilizara el siguiente metodo:

$$G_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A} \quad (19)$$

```
[ ]: %%cython
```

```
import cython
import numpy as np
cimport numpy as np
from scipy import signal

cpdef float[:, :] gradx(float[:, :] input):
    # POR HACER: calcular el gradiente en x
    cdef np.ndarray output=np.zeros([input.shape[0], input.shape[1]], dtype = np.float32)
    cdef np.ndarray kernel = np.array(([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]]), dtype="int")

    output = np.float32(signal.convolve2d(input, kernel, mode = 'same', boundary='fill', fillvalue = 0))

    return output
```

```
time: 3.72 ms
```

```
[ ]: %%cython
import cython
import numpy as np
cimport numpy as np
from scipy import signal

cpdef float[:, :] grady(float[:, :] input):
    # POR HACER: Calcular el gradiente en y
    cdef np.ndarray output=np.zeros([input.shape[0], input.shape[1]], dtype = np.
→float32)
    cdef np.ndarray kernel = np.array(([[-1, -2, -1],[0, 0, 0],[1, 2, 1]]), →
→dtype="int")
    output = np.float32(signal.convolve2d(input, kernel, mode = 'same', boundary= →
→= 'fill', fillvalue = 0))
    return output
```

```
time: 4.15 ms
```

```
[ ]: %%cython
import cython
import numpy as np
cimport numpy as np

cpdef float[:, :] product(float[:, :] input1, float[:, :] input2):
    # POR HACER: generar una matriz que contenga el producto entre input1 e
→input2, pixel a pixel
    cdef int rows, cols,
    cdef np.ndarray output=np.zeros([input1.shape[0], input1.shape[1]], dtype = →
→np.float32)
    rows, cols = input1.shape[0], input1.shape[1]

    for i in range(rows):
        for j in range(cols):
            output[i,j] = input1[i,j] * input2[i,j]

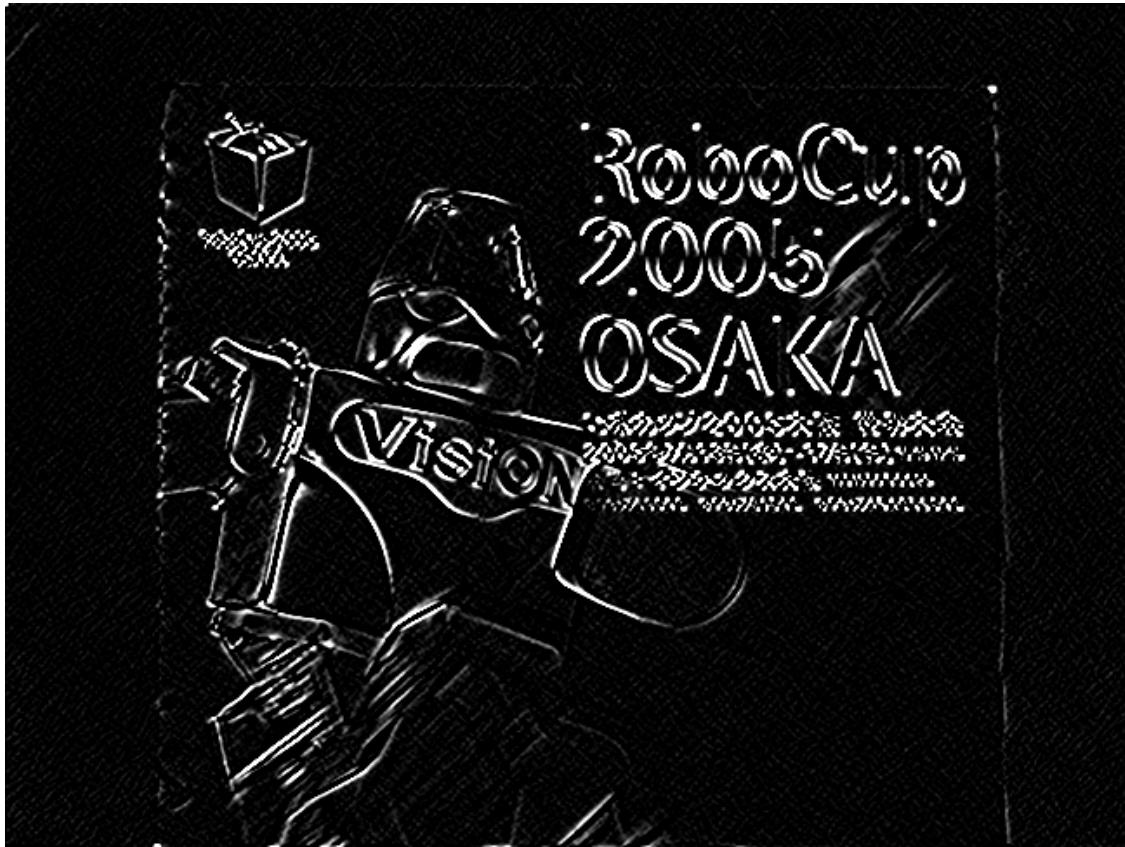
    return np.float32(output)
```

```
time: 3.83 ms
```

```
[ ]: import cv2
from google.colab.patches import cv2_imshow
#img1 = cv2.imread('imagenes_parte_2/uch006a.jpg',0)
originalRGB = cv2.imread('imagenes_parte_2/uch006a.jpg') #Leer imagen
```

```
if len(originalRGB.shape) == 3:  
    original = cv2.cvtColor(originalRGB, cv2.COLOR_BGR2GRAY)  
else:  
    original = originalRGB  
  
input = np.float32( original )  
cv2_imshow(np.float32(input))  
  
out = grady(gradx(input))  
cv2_imshow(np.float32(out))
```





time: 162 ms

```
[ ]: %%cython
import cython
import numpy as np
cimport numpy as np
from __main__ import product

cpdef float[:, :] harris(float[:, :] mxx, float[:, :] mxy, float[:, :] myy):
    ...
    Args:
        mxx: momento xx de la imagen
        myy: momento yy de la imagen
        mxy: momento xy de la imagen
    ...
    # POR HACER: Calcular el filtro de Harris a partir de (mxx, mxy, myy)
    cdef float det, tr,
    cdef int rows, cols,
    cdef np.ndarray output = np.zeros([mxx.shape[0], mxx.shape[1]], dtype = np.
    ↪float32)
    rows, cols = mxx.shape[0], mxx.shape[1]
```

```

for i in range(rows):
    for j in range(cols):
        det = mxx[i,j] * myy[i,j] - mxy[i,j] * mxy[i,j]
        tr = mxx [i,j] + myy[i,j]
        output[i,j] = det - 0.04 * tr*tr

return output

```

time: 3.75 ms

```
[ ]: %%cython
import cython
import numpy as np
cimport numpy as np
from libc.math cimport sqrt
from __main__ import product

cpdef float[:, :] shiTomas(float [:, :] mxx, float [:, :] mxy, float [:, :] ↴myy):
    # POR HACER: Calcular el filtro de Shi-Tomasi a partir de (mxx, mxy, myy)
    cdef float det, tr,
    cdef int rows, cols,
    cdef np.ndarray output = np.zeros([mxx.shape[0], mxx.shape[1]], dtype = np. ↴float32)
    rows, cols = mxx.shape[0], mxx.shape[1]

    for i in range(rows):
        for j in range(cols):
            det = mxx[i,j] * myy[i,j] - mxy[i,j] * mxy[i,j]
            tr = mxx [i,j] + myy[i,j]
            output[i,j] = tr/2 - sqrt(tr*tr - 4*det)/2

    return output

```

time: 2.31 ms

```
[ ]: %%cython
import cython
import numpy as np
cimport numpy as np

# Funcion auxiliar para obtener el maximo de los vecinos.
def get_h(h, i, j, rows, cols):
    output = h[max(i-1, 0):min(i+2, rows), max(j-1, 0):min(j+2, cols)]
    return np.float32(np.amax(output))

```

```

cpdef float[:, :] getMaxima(float [:, :] h, float val):
    # POR HACER: Crear una imagen, inicialmente llena con ceros.
    # Luego, hacer 1 los valores en los cuales:
    #   h[r,c] sea un maximo local respecto a sus 8 vecinos
    #   h[r,c] supera el valor val
    cdef int rows, cols,
    cdef np.ndarray output=np.zeros([h.shape[0], h.shape[1]], dtype = np.float32)
    rows, cols = h.shape[0], h.shape[1]

    for i in range(rows):
        for j in range(cols):
            max_vecino = np.float32(get_h(h, i, j, rows, cols))
            if (max_vecino >= val):
                output[i,j] = 1

    return np.float32(output)

```

time: 2.47 ms

```

[ ]: # Esta funcion genera keypoints a partir de una imagen de entrada
# La imagen de entrada puede ser la salida del filtro de Harris o el de
# Shi-Tomasi
# No es necesario modificar esta funcion
def getKeyPoints(input):
    output = []
    input_rows = input.shape[0]
    input_cols = input.shape[1]
    for r in range(input_rows):
        for c in range(input_cols):
            if input[r,c] > 0:
                kp = cv2.KeyPoint()
                kp.pt = (c,r)
                kp.size = 10
                kp.angle = 0
                output.append(kp)
    return output

```

time: 5.16 ms

```

[ ]: from cv2 import GaussianBlur
from __main__ import product
def harrisDetector(input, val):
    input = np.float32( input )
    # Por hacer: calcular el filtro de Harris
    # Hay que realizar los siguientes pasos:
    # 1) Suavizar la imagen de entrada con cv.GaussianBlur( )

```

```

I_soft = GaussianBlur(input, (3,3), 0)
# 2) Calcular gradientes imx e imy (usando funciones de cython definidas arriba)
Ix , Iy = gradx(I_soft), grady(I_soft)
# 3) Calcular momentos usando la funcion product( )
#     imxx = imx*imx (pixel a pixel)
Ixx = GaussianBlur(np.float32(product(Ix, Ix)), (3,3), 0)
Iyy = GaussianBlur(np.float32(product(Iy, Iy)), (3,3), 0)
Ixy = GaussianBlur(np.float32(product(Ix, Iy)), (3,3), 0)
#     imxy = imx*imy (pixel a pixel)
#     imyy = imy*imy (pixel a pixel)
# 4) Suavizar momentos imxx, imxy, imyy con cv.GaussianBlur( )
#     => Puede requerir transformar las imagenes de Cython a Python usando np.
#         float32( )
# 5) Aplicar el filtro de Harris (usando funcion de Cython definida arriba)
harris_f = harris(np.float32(Ixx), np.float32(Ixy), np.float32(Iyy))
# 6) Encontrar puntos maximos usando getMaxima( )
maximos = getMaxima(harris_f, val)
# Harris -> 2e8, shitomasi -> 7e3
# 7) Generar el listado de puntos usando getKeyPoints( )
# 8) Devolver los puntos e interes y la imagen filtrada
points = getKeyPoints(maximos)

return points, np.float32(harris_f)

```

time: 18.9 ms

```

[ ]: def shiTomasoDetector(input, val):
    input = np.float32( input )
    # Por hacer: calcular el filtro de Shi-Tomasi
    # El procedimiento es similar al de la funcion de arriba
    # Sin embargo, se debe llamar al filtro de Shi-Tomasi en vez del de Harris
    I_soft = GaussianBlur(input, (3,3), 0)
    # 2) Calcular gradientes imx e imy (usando funciones de cython definidas arriba)
    Ix , Iy = gradx(I_soft), grady(I_soft)
    # 3) Calcular momentos usando la funcion product( )
    #     imxx = imx*imx (pixel a pixel)
    Ixx = GaussianBlur(np.float32(product(Ix, Ix)), (3,3), 0)
    Iyy = GaussianBlur(np.float32(product(Iy, Iy)), (3,3), 0)
    Ixy = GaussianBlur(np.float32(product(Ix, Iy)), (3,3), 0)
    #     imxy = imx*imy (pixel a pixel)
    #     imyy = imy*imy (pixel a pixel)
    # 4) Suavizar momentos imxx, imxy, imyy con cv.GaussianBlur( )
    #     => Puede requerir transformar las imagenes de Cython a Python usando np.
    #         float32( )
    # 5) Aplicar el filtro de Harris (usando funcion de Cython definida arriba)

```

```

shitomasi_f = shiTomasi(np.float32(Ixx), np.float32(Ixy), np.float32(Iyy))
# 6) Encontrar puntos maximos usando getMaxima( )
maximos = getMaxima(shitomasi_f, val)
# Harris -> 2e8, shitomasi -> 7e3
# 7) Generar el listado de puntos usando getKeyPoints( )
# 8) Devolver los puntos e interes y la imagen filtrada
points = getKeyPoints(maximos)

return points, np.float32(shitomasi_f)

```

time: 20.1 ms

[]:

```

[ ]: def do_rotate(img, angle):
    h = img.shape[0]
    w = img.shape[1]
    cx = w // 2
    cy = h // 2
    m = cv2.getRotationMatrix2D((cx, cy), -angle, 1.0)
    cosa = np.cos(angle * np.pi / 180.0)
    sina = np.sin(angle * np.pi / 180.0)
    nw = int((h * sina) + (w * cosa))
    nh = int((h * cosa) + (w * sina))
    m[0,2] += (nw / 2) - cx
    m[1,2] += (nh / 2) - cy
    return cv2.warpAffine(img, m, (nw, nh))

```

time: 8.11 ms

```

[ ]: import numpy as np
import cv2
from google.colab.patches import cv2_imshow

img1 = cv2.imread('imagenes_parte_2/uch006a.jpg',0)
img2 = do_rotate(img1, 30)

kp1, h1 = harrisDetector(img1, 2e8)
kp2, h2 = harrisDetector(img2, 2e8)

res1 = cv2.drawKeypoints(img1, kp1, img1, None, cv2.
    DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
res2 = cv2.drawKeypoints(img2, kp2, img2, None, cv2.
    DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

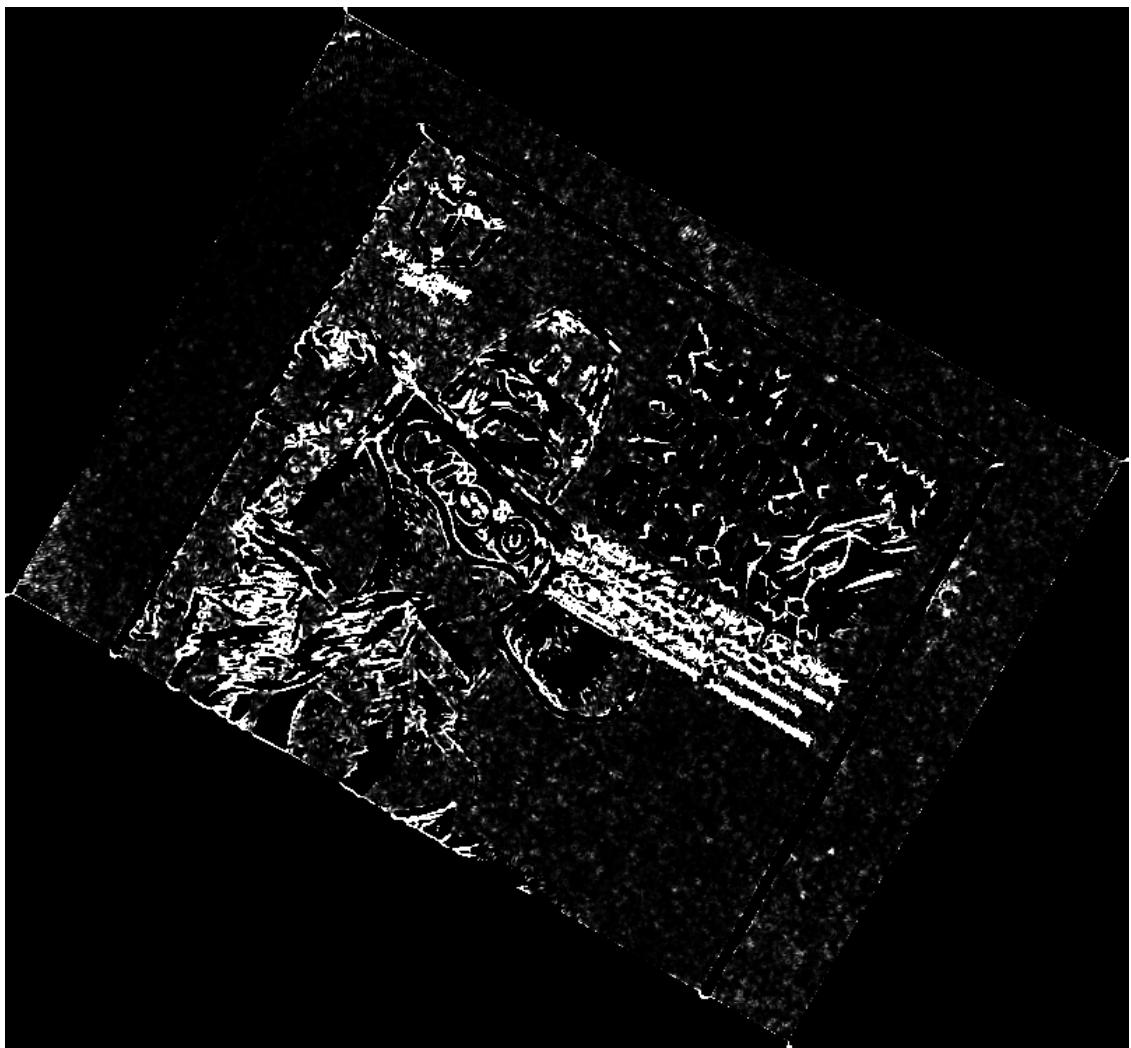
print('Resultados de filtro de Harris')
cv2_imshow(h1)

```

```
cv2_imshow(h2)
#cv2_imshow(h1 * 100 / np.max(np.max(h1)))
#cv2_imshow(h2 * 100 / np.max(np.max(h2)))
print('Puntos de interes')
cv2_imshow(res1)
cv2_imshow(res2)
```

Resultados de filtro de Harris





Puntos de interes





time: 11.5 s

```
[ ]: # Visualizando ShiTomasi
img1 = cv2.imread('imagenes_parte_2/uch006a.jpg',0)
img2 = do_rotate(img1, 30)

kp1, h1 = shiTomasiDetector(img1, 3e3)
kp2, h2 = shiTomasiDetector(img2, 3e3)

res1 = cv2.drawKeypoints(img1, kp1, img1, None, cv2.
    ↪DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
res2 = cv2.drawKeypoints(img2, kp2, img2, None, cv2.
    ↪DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

print('Resultados de filtro de Harris')
```

```
cv2_imshow(h1)
cv2_imshow(h2)
#cv2_imshow(h1 * 100 / np.max(np.max(h1)))
#cv2_imshow(h2 * 100 / np.max(np.max(h2)))
print('Puntos de interes')
cv2_imshow(res1)
cv2_imshow(res2)
```

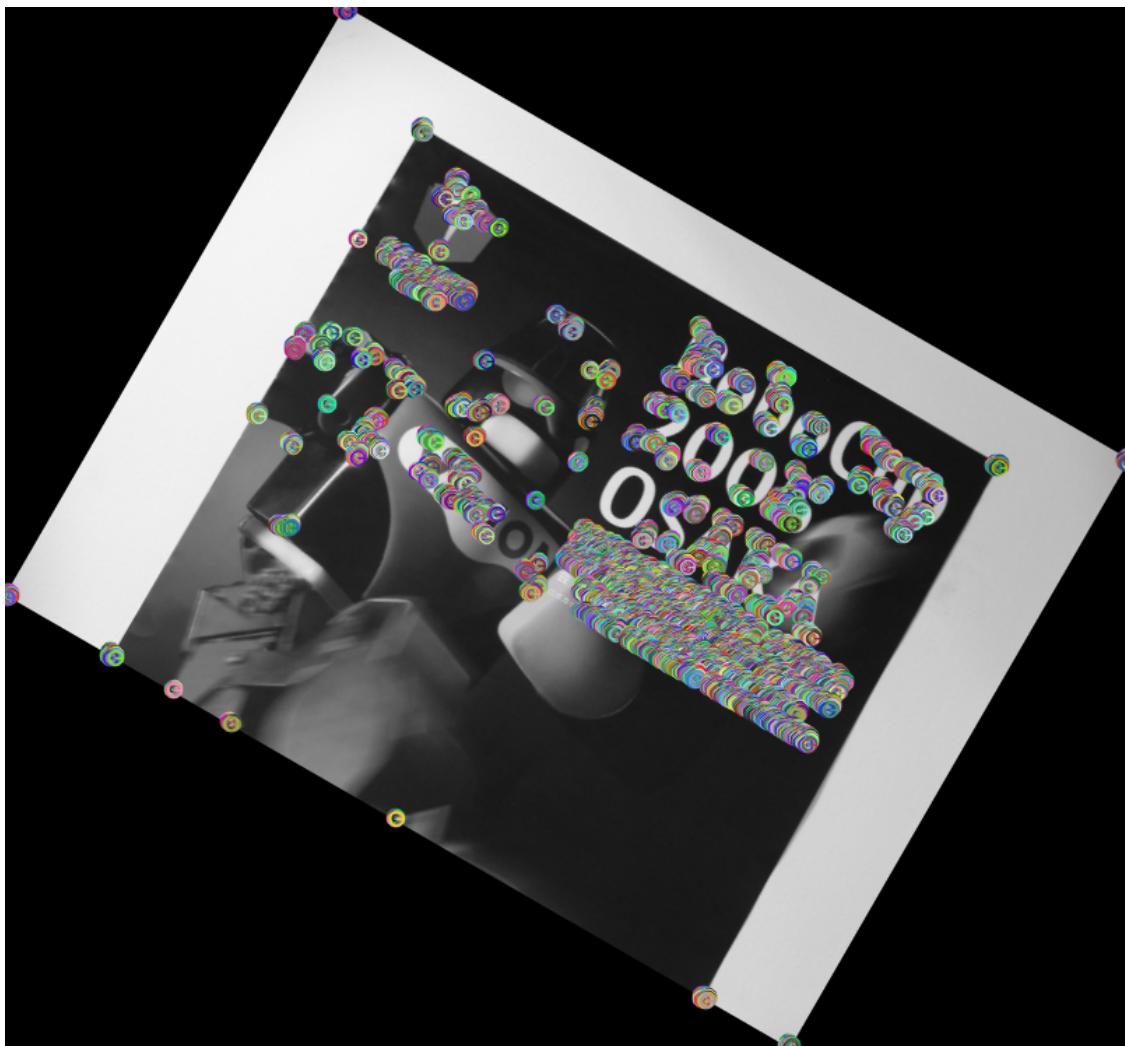
Resultados de filtro de Harris





Puntos de interes





time: 11.6 s

[]:

Parte 3

```
[ ]: # Esta funcion ya esta lista, no debe ser modificada
def filterMatches(matches):
    # Apply ratio test
    points1 = []
    points2 = []
    good = []

    for m,n in matches:
        if m.distance < 0.75*n.distance: # 0.75
            good.append([m])
```

```

    points1.append(kp1[m.queryIdx].pt)
    points2.append(kp2[m.trainIdx].pt)
    return np.array(points1), np.array(points2), good

```

time: 3.39 ms

```

[ ]: import numpy as np
import cv2
from google.colab.patches import cv2_imshow

img1 = cv2.imread('imagenes_parte_3/left2.jpg',0) #/content/imagenes_parte_3/

img2 = cv2.imread('imagenes_parte_3/right2.jpg',0)

detector = cv2.SIFT_create()

(kp1, des1) = detector.detectAndCompute(img1, None)
(kp2, des2) = detector.detectAndCompute(img2, None)

# BFMatcher with default params
bf = cv2.BFMatcher()
matches = bf.knnMatch(des1,des2, k=2)

points1, points2, good = filterMatches(matches)

img_match = cv2.drawMatchesKnn(img1, kp1, img2, kp2,
                               good, None, flags=cv2.

cv2_imshow(img_match)

# El objetivo del codigo que sigue es alinear dos imagenes

# 1) Por hacer: calcular la homografia usando cv2.findHomography( ) con

M, mask = cv2.findHomography(points2, points1, cv2.RANSAC)

# 2) Por hacer: transformar la imagen img2 usando la funcion warpPerspective( )

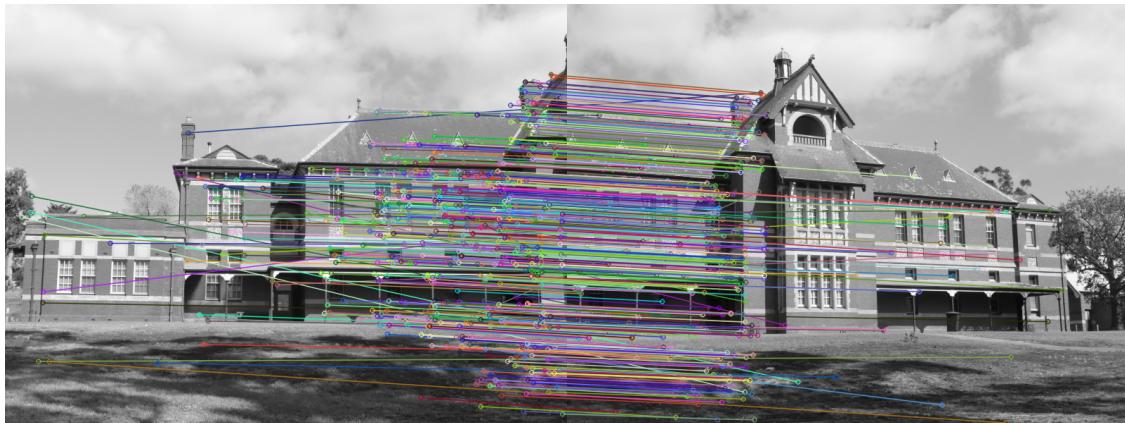
im_size = ((img1.shape[0] + img2.shape[0])*2, img1.shape[1])

im_wraped = cv2.warpPerspective( img2, M, im_size )

# La imagen de salida debe tener un tamano suficiente
# Se recomienda que tenga el doble del ancho de la imagen de entrada
im_wraped[:img1.shape[0],:img1.shape[1]] = img1[:img1.shape[0], :img1.shape[1]]

```

```
# 3) Por hacer: copiar la imagen img1 a la imagen resultante del paso anterior ↳
    ↳( se obtiene la imagen fusionada )
cv2_imshow( im_wraped )
# 4) Por hacer: mostrar la imagen fusionada
```



time: 1.1 s

```
[ ]: import numpy as np
import cv2
from google.colab.patches import cv2_imshow

img1 = cv2.imread('imagenes_parte_3/left3.jpg',0)
img2 = cv2.imread('imagenes_parte_3/right3.jpg',0)

detector = cv2.SIFT_create()

(kp1, des1) = detector.detectAndCompute(img1, None)
(kp2, des2) = detector.detectAndCompute(img2, None)
```

```

# BFMatcher with default params
bf = cv2.BFMatcher()
matches = bf.knnMatch(des1,des2, k=2)

points1, points2, good = filterMatches(matches)

img_match = cv2.drawMatchesKnn(img1, kp1, img2, kp2,
                               good, None, flags=cv2.
                               →DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
cv2_imshow(img_match)

# El objetivo del codigo que sigue es alinear dos imagenes

# 1) Por hacer: calcular la homografia usando cv2.findHomography( ) con
# →parametro cv2.RANSAC
M, mask = cv2.findHomography(points2, points1, cv2.RANSAC)

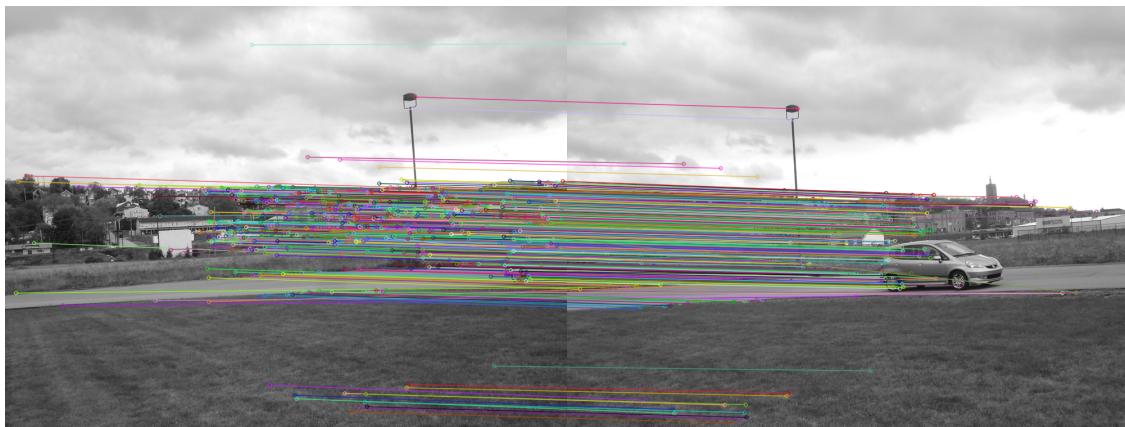
# 2) Por hacer: transformar la imagen img2 usando la funcion warpPerspective( ) y
# → la homografia

im_size = ((img1.shape[0] + img2.shape[0])*2, img1.shape[1])

im_wraped = cv2.warpPerspective( img2, M, im_size )

# La imagen de salida debe tener un tamano suficiente
# Se recomienda que tenga el doble del ancho de la imagen de entrada
im_wraped[:img1.shape[0],:img1.shape[1]] = img1[:img1.shape[0], :img1.shape[1]]
# 3) Por hacer: copiar la imagen img1 a la imagen resultante del paso anterior
# →( se obtiene la imagen fusionada )
cv2_imshow( im_wraped )
# 4) Por hacer: mostrar la imagen fusionada

```





time: 1.1 s

```
[ ]: import numpy as np
import cv2
from google.colab.patches import cv2_imshow

img1 = cv2.imread('imagenes_parte_3/left1.png',0)
img2 = cv2.imread('imagenes_parte_3/right1.png',0)

detector = cv2.SIFT_create()

(kp1, des1) = detector.detectAndCompute(img1, None)
(kp2, des2) = detector.detectAndCompute(img2, None)

# BFMatcher with default params
bf = cv2.BFMatcher()
matches = bf.knnMatch(des1,des2, k=2)

points1, points2, good = filterMatches(matches)

img_match = cv2.drawMatchesKnn(img1, kp1, img2, kp2,
                               good, None, flags=cv2.
                               DRAW_MATCHES_FLAGS_NOT_DRAW_SINGLE_POINTS)
cv2_imshow(img_match)

# El objetivo del codigo que sigue es alinear dos imagenes

# 1) Por hacer: calcular la homografia usando cv2.findHomography( ) con
parametro cv2.RANSAC
M, mask = cv2.findHomography(points2, points1, cv2.RANSAC)

# 2) Por hacer: transformar la imagen img2 usando la funcion warpPerspective( )
y la homografia
```

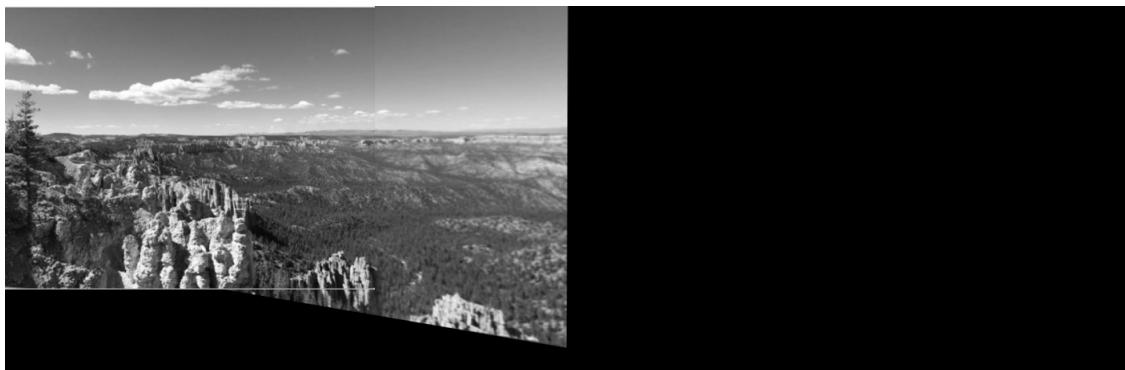
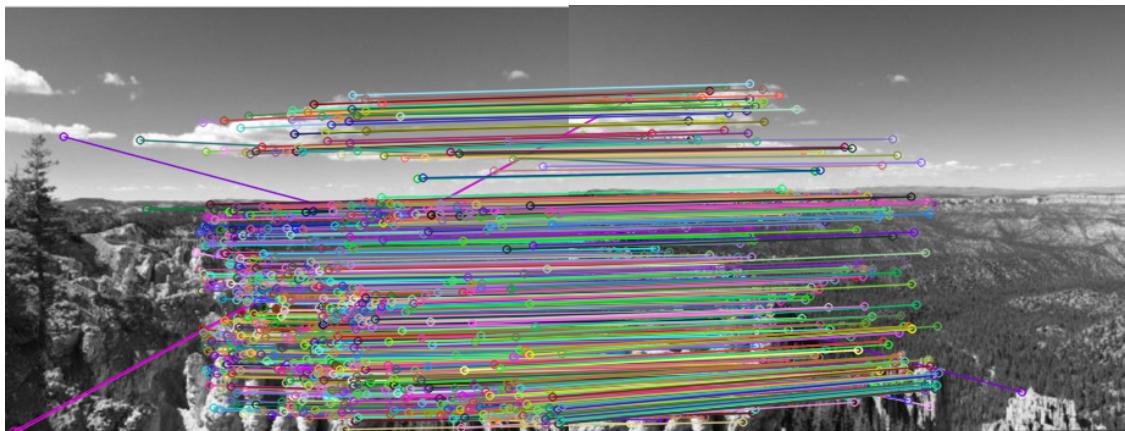
```

im_size = ((img1.shape[0] + img2.shape[0])*2, img1.shape[1])

im_wraped = cv2.warpPerspective( img2, M, im_size )

# La imagen de salida debe tener un tamano suficiente
# Se recomienda que tenga el doble del ancho de la imagen de entrada
im_wraped[:img1.shape[0],:img1.shape[1]] = img1[:img1.shape[0], :img1.shape[1]]
# 3) Por hacer: copiar la imagen img1 a la imagen resultante del paso anterior
# →( se obtiene la imagen fusionada )
cv2_imshow( im_wraped )
# 4) Por hacer: mostrar la imagen fusionada

```



time: 428 ms

```

[ ]: import numpy as np
import cv2
from google.colab.patches import cv2_imshow

img1 = cv2.imread('imagenes_parte_3/left4.jpg',0)

```

```

img2 = cv2.imread('imagenes_parte_3/right4.jpg',0)

detector = cv2.SIFT_create()

(kp1, des1) = detector.detectAndCompute(img1, None)
(kp2, des2) = detector.detectAndCompute(img2, None)

# BFMatcher with default params
bf = cv2.BFMatcher()
matches = bf.knnMatch(des1,des2, k=2)

points1, points2, good = filterMatches(matches)

img_match = cv2.drawMatchesKnn(img1, kp1, img2, kp2,
                               good, None, flags=cv2.
                               →DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
cv2_imshow(img_match)

# El objetivo del codigo que sigue es alinear dos imagenes

# 1) Por hacer: calcular la homografia usando cv2.findHomography( ) con
→parametro cv2.RANSAC
M, mask = cv2.findHomography(points2, points1, cv2.RANSAC)

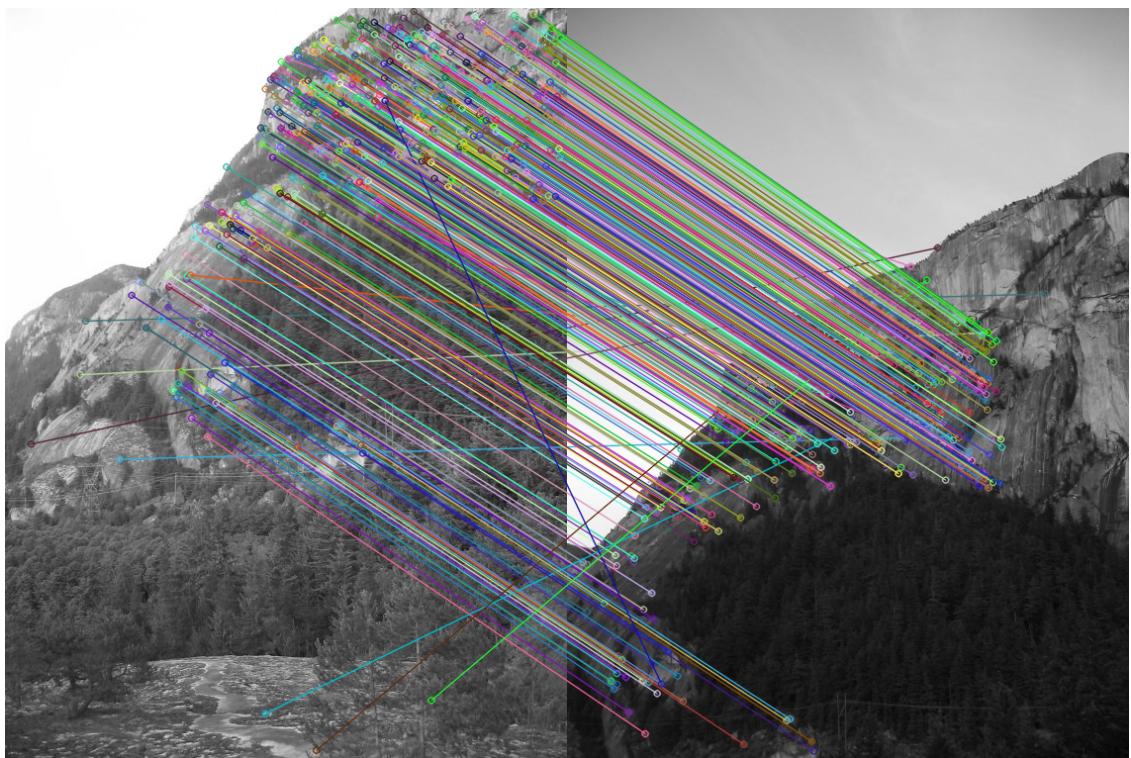
# 2) Por hacer: transformar la imagen img2 usando la funcion warpPerspective( )
→y la homografia

im_size = ((img1.shape[0] + img2.shape[0]), img1.shape[1]*2)

im_wraped = cv2.warpPerspective( img2, M, im_size )

# La imagen de salida debe tener un tamano suficiente
# Se recomienda que tenga el doble del ancho de la imagen de entrada
im_wraped[:img1.shape[0],:img1.shape[1]] = img1[:img1.shape[0], :img1.shape[1]]
# 3) Por hacer: copiar la imagen img1 a la imagen resultante del paso anterior
→( se obtiene la imagen fusionada )
cv2_imshow( im_wraped )
# 4) Por hacer: mostrar la imagen fusionada

```



time: 707 ms

3 Conclusion

En el presente trabajo, se aprendio sobre la utilidad de los detectores de esquinas como generadores de caracteristicas de una imagen. Estos pueden ser utiles para hacer *match* entre imagenes u otras utilidades como identificacion de objetos con distinta posicion relativa o en distinta orientacion.

[]: