```
from google.colab import files
uploaded = files.upload()
```

Elegir archivos 8 archivos

- **left1.png**(image/png) - 277119 bytes, last modified: 12/10/2018 - 100% done
- **left2.jpg**(image/jpeg) - 460055 bytes, last modified: 12/10/2018 - 100% done
- **left3.jpg**(image/jpeg) - 573641 bytes, last modified: 17/10/2018 - 100% done
- **left4.jpg**(image/jpeg) - 109215 bytes, last modified: 17/10/2018 - 100% done
- **right1.png**(image/png) - 265042 bytes, last modified: 12/10/2018 - 100% done
- **right2.jpg**(image/jpeg) - 478172 bytes, last modified: 12/10/2018 - 100% done
- **right3.jpg**(image/jpeg) - 547728 bytes, last modified: 17/10/2018 - 100% done
- **right4.jpg**(image/jpeg) - 61019 bytes, last modified: 17/10/2018 - 100% done

```
Saving left1.png to left1.png
Saving left2.jpg to left2.jpg
Saving left3.jpg to left3.jpg
Saving left4.jpg to left4.jpg
Saving right1.png to right1.png
Saving right2.jpg to right2.jpg
Saving right3.jpg to right3.jpg
Saving right4.jpg to right4.jpg
time: 50.7 s
```

```
!ls
```

```
'poligonos (1).png'   sample_data        uch006a.jpg        uch101b.jpg
 poligonos.png       'uch006a (1).jpg'  'uch101b (1).jpg'
time: 126 ms
```

```
!pip install ipython-autotime
```

```
Requirement already satisfied: ipython-autotime in /usr/local/lib/python3.6/dist-pack
Requirement already satisfied: ipython in /usr/local/lib/python3.6/dist-packages (fro
Requirement already satisfied: pickleshare in /usr/local/lib/python3.6/dist-packages
Requirement already satisfied: pexpect; sys_platform != "win32" in /usr/local/lib/pyt
Requirement already satisfied: decorator in /usr/local/lib/python3.6/dist-packages (f
Requirement already satisfied: prompt-toolkit<2.0.0,>=1.0.4 in /usr/local/lib/python3
Requirement already satisfied: pygments in /usr/local/lib/python3.6/dist-packages (fr
Requirement already satisfied: traitlets>=4.2 in /usr/local/lib/python3.6/dist-packag
Requirement already satisfied: simplegeneric>0.8 in /usr/local/lib/python3.6/dist-pac
Requirement already satisfied: setuptools>=18.5 in /usr/local/lib/python3.6/dist-pack
Requirement already satisfied: ptyprocess>=0.5 in /usr/local/lib/python3.6/dist-packa
Requirement already satisfied: six>=1.9.0 in /usr/local/lib/python3.6/dist-packages (
Requirement already satisfied: wcwidth in /usr/local/lib/python3.6/dist-packages (fro
Requirement already satisfied: ipython-genutils in /usr/local/lib/python3.6/dist-pack
time: 2.64 s
```

```
%load_ext autotime
```

```
The autotime extension is already loaded. To reload it, use:
  %reload_ext autotime
time: 4.55 ms
```

```
!pip install -U opencv-python
```

```
Requirement already up-to-date: opencv-python in /usr/local/lib/python3.6/dist-packag
Requirement already satisfied, skipping upgrade: numpy>=1.13.3 in /usr/local/lib/pyth
time: 2.93 s
```

Introducción

En el presente documento se presenta tanto el desarrollo, como los resultados obtenidos durante el desarrollo de la segunda tarea del curso EL7008. En esta, se implementan y analizan los detectores de puntos de interés de Harris y Shi-tomasi, además de la correspondencia entre imágenes utilizando descriptores para finalmente alinear imágenes.

A continuación, se listan las secciones principales del informe:

- Marco Teórico: en esta sección se da una breve explicación de los algoritmos principales, tales como los detectores de Harris y Shi-tomasi, la correspondencia usando descriptores y la transformación de homografía.
- Cálculo de puntos de interés
- Alineamiento de imágenes

Marco teórico

Detector de Harris

El detector de Harris, sirve para detectar puntos de interés, los cuales en este caso corresponden a esquinas. Estos puntos pueden servir para describir la imágen en cierto modo, lo cual tiene aplicaciones en el área de visión computacional.

Detector de Shi-tomasi

Correspondencias usando descriptores

Transformación de homografía

Cálculo de puntos de interés

```
import cython
import cv2

```

```
time: 2.43 ms
```

```
%load_ext Cython
```

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

En primer lugar, se programan las funciones para el calculo de los gradientes en ambos ejes (x e y). Estas se programan en cython para obtener un menor tiempo de computo. En ambos casos se calcula la derivada hacia adelante, es decir, para el punto (x,y), la derivada en el eje x se calcula como:

$$\frac{d}{dx}(x,y) = (x+1,y) - (x,y)$$

de la misma forma, la derivada en el eje y queda:

$$\frac{d}{dy}(x,y) = (x,y+1) - (x,y)$$

Esto se calcula para todos los puntos exceptuando los bordes en los que no es posible tomar el punto hacia adelante (borde vertical derecho y borde horizontal inferior).

```
%%cython
cimport numpy as np
import cython
import numpy as np


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 int rows, cols
  rows, cols = input.shape[0], input.shape[1]

  for j in range(cols):
    for i in range(rows - 1):
      output[i,j] = input[i + 1,j] - input[i,j]
    output[rows - 1, j] = input[rows - 1, j]
  return output
```

```
time: 11.7 ms
```

```
%%cython
cimport numpy as np
import cython
import numpy as np


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 int rows, cols
  rows, cols = input.shape[0], input.shape[1]

  for i in range(rows):
    for j in range(cols - 1):
      output[i,j] = input[i,j+1] - input[i,j]
    output[i, cols - 1] = input[i, cols - 1]
```

```
    output[i, cols - 1] = input[i, cols - 1]
  return output
```

```
time: 4.68 ms
```

Por otro lado, se utilizan las funciones auxiliares subtract y add, para la resta y suma de arrays.

```
def subtract(input1, input2):
  # Por hacer: calcular la resta entre input1 e input2, pixel a pixel
  assert(input1.shape == input2.shape)
  output = np.zeros_like(input1)
  for i in range(input1.shape[0]):
    for j in range(input2.shape[1]):
      output[i,j] = input1[i,j] - input2[i,j]
  return output
```

```
time: 4.36 ms
```

```
def add(input1, input2):
  # Por hacer: calcular la resta entre input1 e input2, pixel a pixel
  assert(input1.shape == input2.shape)
  output = np.zeros_like(input1)
  for i in range(input1.shape[0]):
    for j in range(input2.shape[1]):
      output[i,j] = input1[i,j] + input2[i,j]
  return output
```

```
time: 8.52 ms
```

A continuación, se muestra la función *product*, que se encarga de multiplicar pixel a pixel, dos matrices de entrada. Así, se recorren ambos inputs para hacer la multiplicación.

```
%%cython
cimport numpy as np
import cython
import numpy as np


cpdef float[:, :] product(float [:, :] input1, float [:, :] input2):
  # POR HACER: generar una matriz que contenga el producto entre input1 e input2, pixel
  cdef np.ndarray output=np.zeros([input2.shape[0], input2.shape[1]], dtype = np.float3
  cdef int rows, cols
  rows, cols = input2.shape[0], input2.shape[1]
  for i in range(rows):
    for j in range(cols):
      output[i,j] = input1[i,j] * input2[i,j]
  return output
```

```
time: 3.99 ms
```

La función *harris*, es la implementación del filtro de Harris a partir de **mxx**, **mxy** y **myy**.

En específico, aplica la regla mencionada en el marco teórico:

$$cornerness_{harris} = det - 0.04 * tr^2$$

donde

$$det = mxx \cdot myy - mxy^2$$
$$tr = mxx + myy$$

```
%%cython
cimport numpy as np
import cython
import numpy as np
from __main__ import product, subtract, add

cpdef float[:, :] harris(float [:, :] mxx, float [:, :] mxy, float [:, :] myy):
  # POR HACER: Calcular el filtro de Harris a partir de (mxx, mxy, myy)
  cdef np.ndarray output, det, tr

  output=np.zeros([mxx.shape[0], mxx.shape[1]], dtype = np.float32)
  det = subtract(product(mxx, myy),product(mxy, mxy))
  tr = add(mxx, myy)
  output = subtract(det,np.multiply(0.04,product(tr,tr)))
  return output
```

```
time: 4.1 ms
```

La función *shiTomasi*, es la implementación del filtro de Harris a partir de **mxx**, **mxy** y **myy**.

En específico, aplica la regla mencionada en el marco teórico:

$$\frac{tr}{2} - \frac{\sqrt{tr^2 - 4 \cdot det}}{2}$$

con las mismas definiciones para la traza (tr) y determinante (det) que en el caso anterior.

```
%%cython
import cython
import numpy as np
cimport numpy as np
from libc.math cimport sqrt
from __main__ import product, subtract, add

cpdef float[:, :] shiTomasi(float [:, :] mxx, float [:, :] mxy, float [:, :] myy):
  # POR HACER: Calcular el filtro de Shi-Tomasi a partir de (mxx, mxy, myy)
  cdef np.ndarray output, det, tr

  output = np.zeros([mxx.shape[0], mxx.shape[1]], dtype = np.float32)
  det = subtract(product(mxx, myy),product(mxy, mxy))
  tr = add(mxx, myy)
  output = subtract(tr/2, np.sqrt(subtract(product(tr,tr), 4*det))/2)
  return output
```

```
time: 4.06 ms
```

Se programa la función auxiliar *get_vecinos*, la cual recibe los índices de un punto en una matriz y entrega los índices de los 8 vecinos más cercanos de este punto. Así, si se reciben como indices (idx = 5, idy = 5), entonces se debe retornar una lista con las siguientes coordenadas:

vecinos = [4,4] [4,5] [4,6] [5,4] [5,6] [6,4] [6,5] [6,6]

```
def get_vecinos(idx,idy):
  vecinos = []
  for a in range(-1,2,1):
    for b in range(-1,2,1):
      if a != 0 or b != 0:
        vecinos.append((idx + a, idx + b))
  return vecinos
```

```
time: 4.92 ms
```

Por otro lado, se tiene la función *getMaxima*, la cual recibe una matriz y un threshold, y se encarga de evaluar para cada pixel, si este corresponde a un máximo local respecto a sus 8 vecinos, adicionalmente, cada máximo local debe ser mayor que el threshold. El output de la función es una máscara de las mismas dimensiones que la matriz de entrada, que contiene un 1 en las posiciones en las que se encontró un máximo local.

```
%%cython
import cython
import numpy as np
cimport numpy as np
from __main__ import get_vecinos, product

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):
      #Se asume que el punto es un maximo
      esMaximo = True
      #primero se descartan los pixeles menores o iguales al valor de threshold
      if h[i,j] > val:
        vecinos = get_vecinos(i,j) # se obtienen los indices vecinos
        for vecino in vecinos:
          #los casos de borde se manejan con try-except
          try:
            #si al menos un vecino es mayor o igual que el pixel, entonces no es maximo
            if h[vecino[0],vecino[1]] >= h[i,j]:
```

```
                esMaximo = False
          except:
            pass
      else:
        esMaximo = False

      if esMaximo:
        output[i,j] = 1

  return output
```

```
time: 4.29 ms
```

La función auxiliar *getKeyPoints*, genera una lista con objetos cv2.KeyPoint, a partir de una matriz de entrada. En este caso, los *key points* son los índices en los cuales el valor del input es mayor a cero.

```
# 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: 7.61 ms
```

Las funciones *harrisDetector* y *shiTomasiDetector*, implementan el procedimiento para obtener los puntos de interés con ambos métodos descritos en el marco teórico.

El cálculo en ambos casos es similar, por lo que a continuación se describen los pasos que se siguen en las dos funciones:

- Se suaviza la imagen con una gaussiana.
- Se calculan los momentos **imxx**, **imyy** e **imxy** a partir de los gradientes en los ejes x e y.
- Se suavizan los momentos utilizando un sigma mayor que en el caso anterior para la gaussiana.
- A partir de los momentos suavizados, se calculan los puntos de interés de los filtros de Harris/Shi-Tomasi con la función *harris* o *shiTomasi* según corresponda.
- Se obtienen los máximos locales del filtro utilizando la función *getMaxima*.

- Finalmente a partir de los máximos encontrados, se generan los **keypoints**.

```
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( )
  blur = cv2.GaussianBlur(input, (7,7), 1)
  # 2) Calcular gradientes imx e imy (usando funciones de cython definidas arriba)
  imx = np.float32(gradx(blur))
  imy = np.float32(grady(blur))
  # 3) Calcular momentos usando la funcion product( )
  #    imxx = imx*imx (pixel a pixel)
  imxx = product(imx, imx)
  #    imxy = imx*imy (pixel a pixel)
  imxy = product(imx,imy)
  #    imyy = imy*imy (pixel a pixel)
  imyy = product(imy,imy)


  # 4) Suavizar momentos imxx, imxy, imyy con cv.GaussianBlur( )
  #    => Puede requerir transformar las imagenes de Cython a Python usando np.float32(
  imxx = cv2.GaussianBlur(np.float32(imxx), (7,7), 3)
  imxy = cv2.GaussianBlur(np.float32(imxy), (7,7), 3)
  imyy = cv2.GaussianBlur(np.float32(imyy), (7,7), 3)
  # 5) Aplicar el filtro de Harris (usando funcion de Cython definida arriba)
  filtrada = harris(imxx,imxy,imyy)
  # 6) Encontrar puntos maximos usando getMaxima( )
  maximos = getMaxima(filtrada, val)
  # 7) Generar el listado de puntos usando getKeyPoints( )
  key_points = getKeyPoints(maximos)
  # 8) Devolver los puntos e interes y la imagen filtrada

  return key_points, np.float32(filtrada)
```

```
time: 16.6 ms
```

```
def shiTomasiDetector(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

  # 1) Suavizar la imagen de entrada con cv.GaussianBlur( )
  blur = cv2.GaussianBlur(input, (7,7), 1)
  # 2) Calcular gradientes imx e imy (usando funciones de cython definidas arriba)
  imx = gradx(blur)
  imy = grady(blur)
  # 3) Calcular momentos usando la funcion product( )
  #    imxx = imx*imx (pixel a pixel)
  imxx = product(imx,imx)
  #    imxy = imx*imy (pixel a pixel)
  imxy = product(imx,imy)
  #    imyy = imy*imy (pixel a pixel)
  imyy = product(imy,imy)
```

```


  # 4) Suavizar momentos imxx, imxy, imyy con cv.GaussianBlur( )
  #    => Puede requerir transformar las imagenes de Cython a Python usando np.float32(
  imxx = np.float32(cv2.GaussianBlur(np.float32(imxx), (7,7), 3))
  imxy = np.float32(cv2.GaussianBlur(np.float32(imxy), (7,7), 3))
  imyy = np.float32(cv2.GaussianBlur(np.float32(imyy), (7,7), 3))
  # 5) Aplicar el filtro de Shi-tomasi (usando funcion de Cython definida arriba)
  filtrada = shiTomasi(imxx,imxy,imyy)
  # 6) Encontrar puntos maximos usando getMaxima( )
  maximos = getMaxima(filtrada, val)
  # 7) Generar el listado de puntos usando getKeyPoints( )
  key_points = getKeyPoints(maximos)
  # 8) Devolver los puntos e interes y la imagen filtrada

  return key_points, np.float32(filtrada)
```

```
time: 17 ms
```

Función auxiliar que rota una imagen de entrada en un cierto ángulo de inclinación.

```
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: 12.1 ms
```

Finalmente se aplican los detectores implementados sobre 3 imágenes de prueba, utilizando la función auxiliar *generate_results*, la cual se encarga de aplicar los detectores y mostrar los resultados.

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

def generate_results(path_to_img, angle, threshold, method = 'harris'):
  img1 = cv2.imread(path_to_img,0)
  img2 = do_rotate(img1, angle)

  if method == 'harris':
    kp1, h1 = harrisDetector(img1, threshold)
```

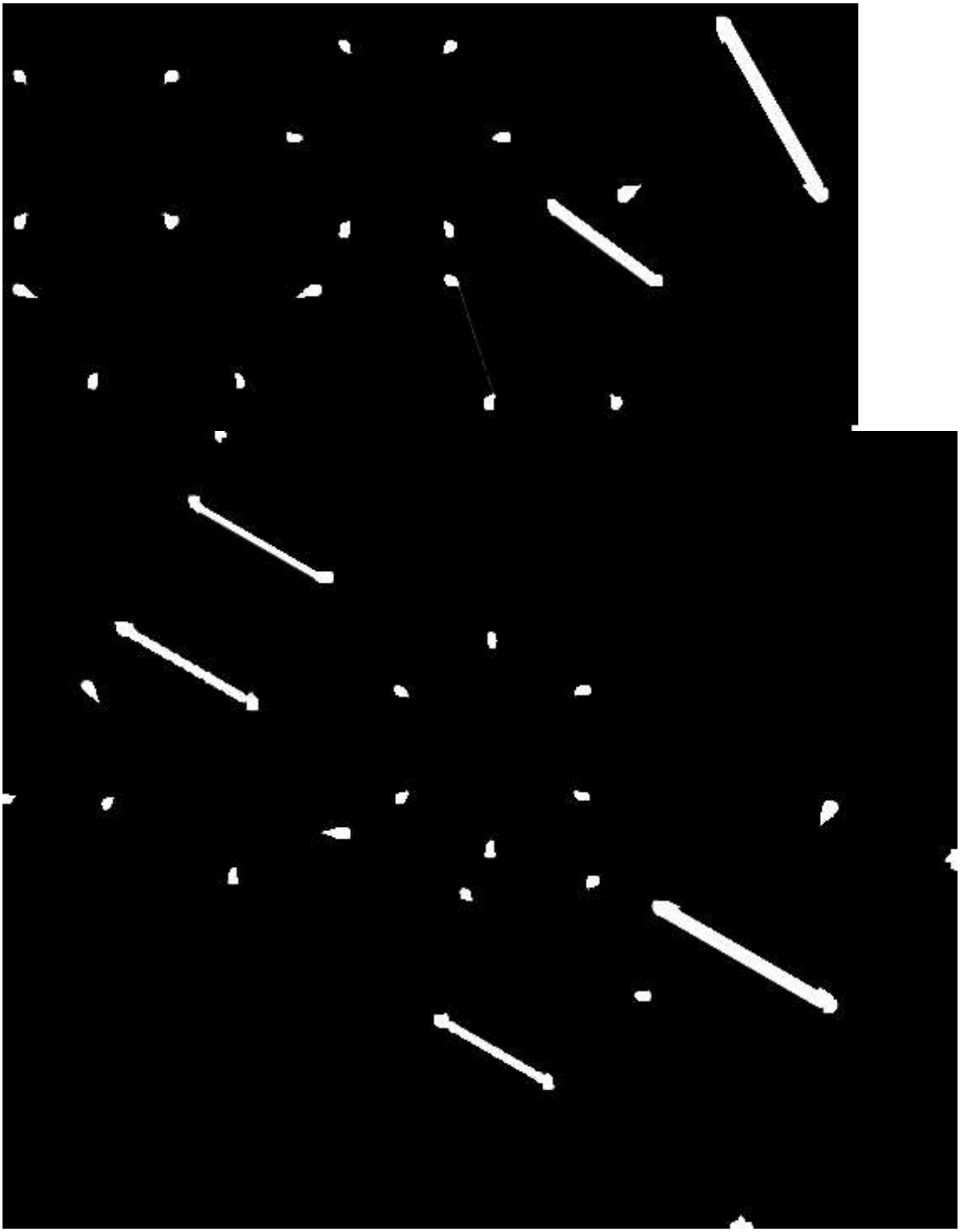
```
    kp2, h2 = harrisDetector(img2, threshold)
    print1 = 'Resultados de filtro de Harris'
    print2 = 'Puntos de interes filtro de Harris'
  if method == 'shitomasi':
    kp1, h1 = shiTomasiDetector(img1, threshold)
    kp2, h2 = shiTomasiDetector(img2, threshold)
    print1 = 'Resultados de filtro de Shi-tomasi'
    print2 = 'Puntos de interes filtro de Shi-tomasi'
  res1 = cv2.drawKeypoints(img1, kp1, img1, None, cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYP
  res2 = cv2.drawKeypoints(img2, kp2, img2, None, cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYP


  print(print1)
  cv2_imshow(h1)
  cv2_imshow(h2)
  print(print2)
  cv2_imshow(res1)
  cv2_imshow(res2)

```

```
time: 26.7 ms
```

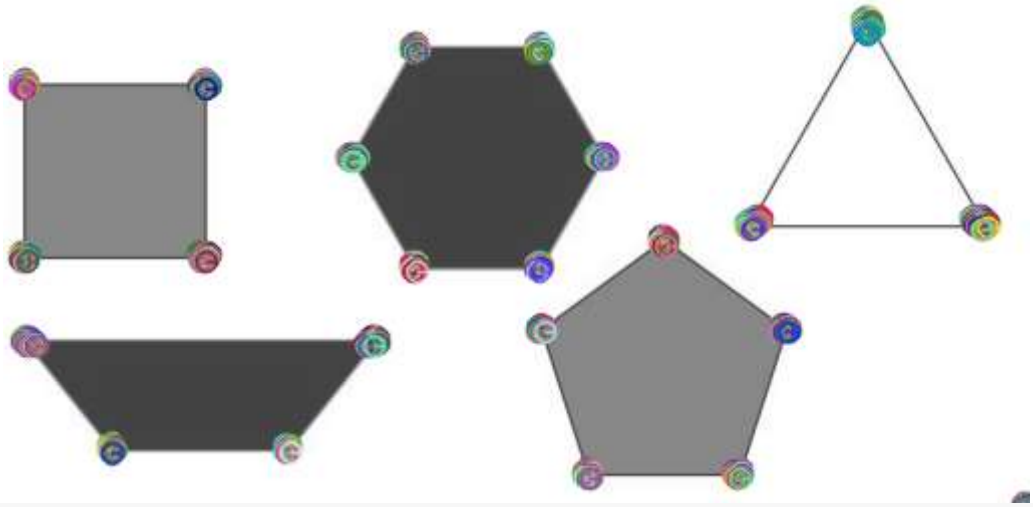
Pruebas con el detector de Harris

```
generate_results('poligonos.png', 30, 3e4,method='harris')
```

Resultados de filtro de Harris



Puntos de interes



```
generate_results('uch006a.jpg', 80, 3e5, method='harris')
```

```
generate_results( uch006a.jpg , 80, 5e5,method= harris )
```

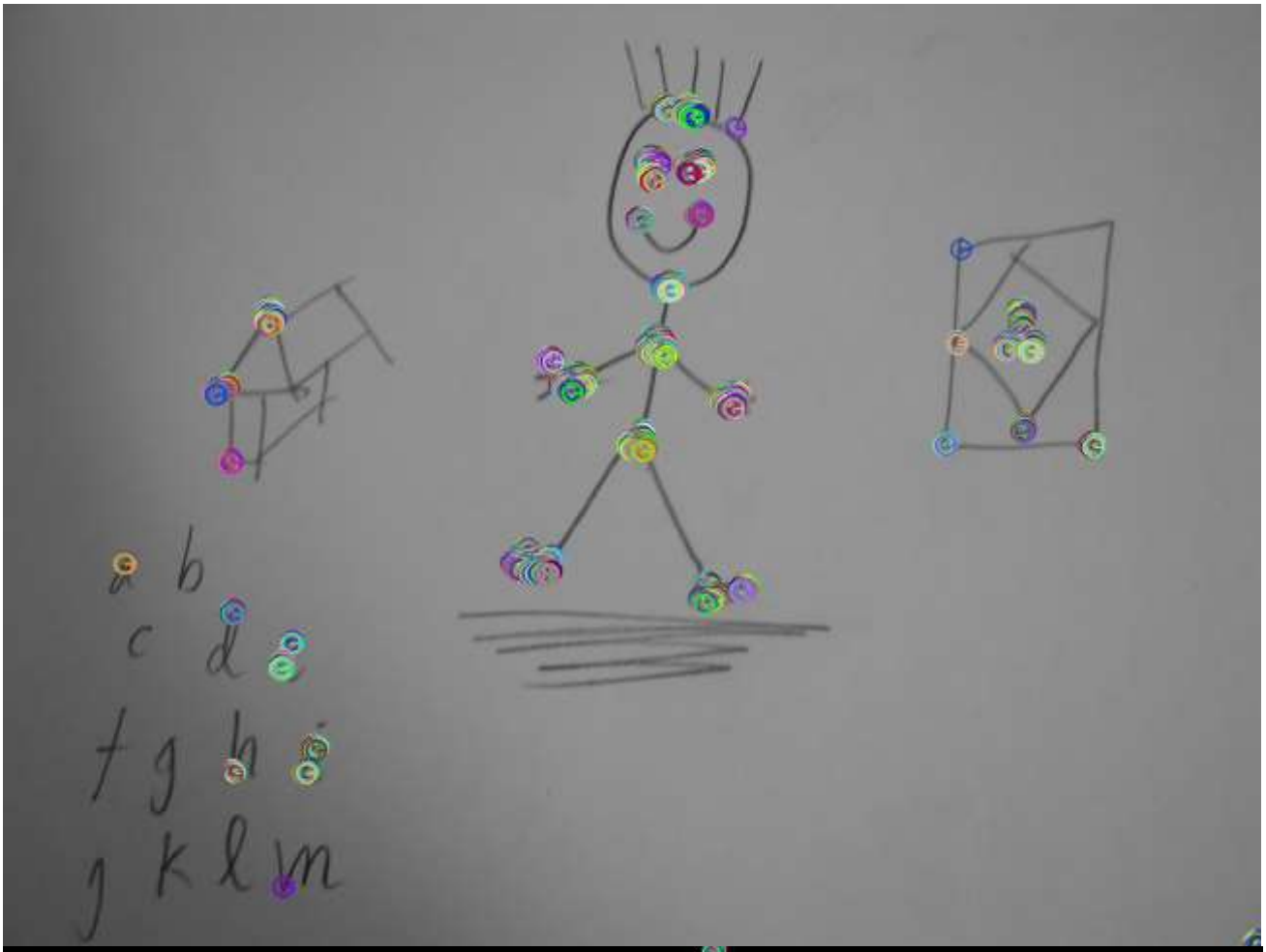
Resultados de filtro de Harris

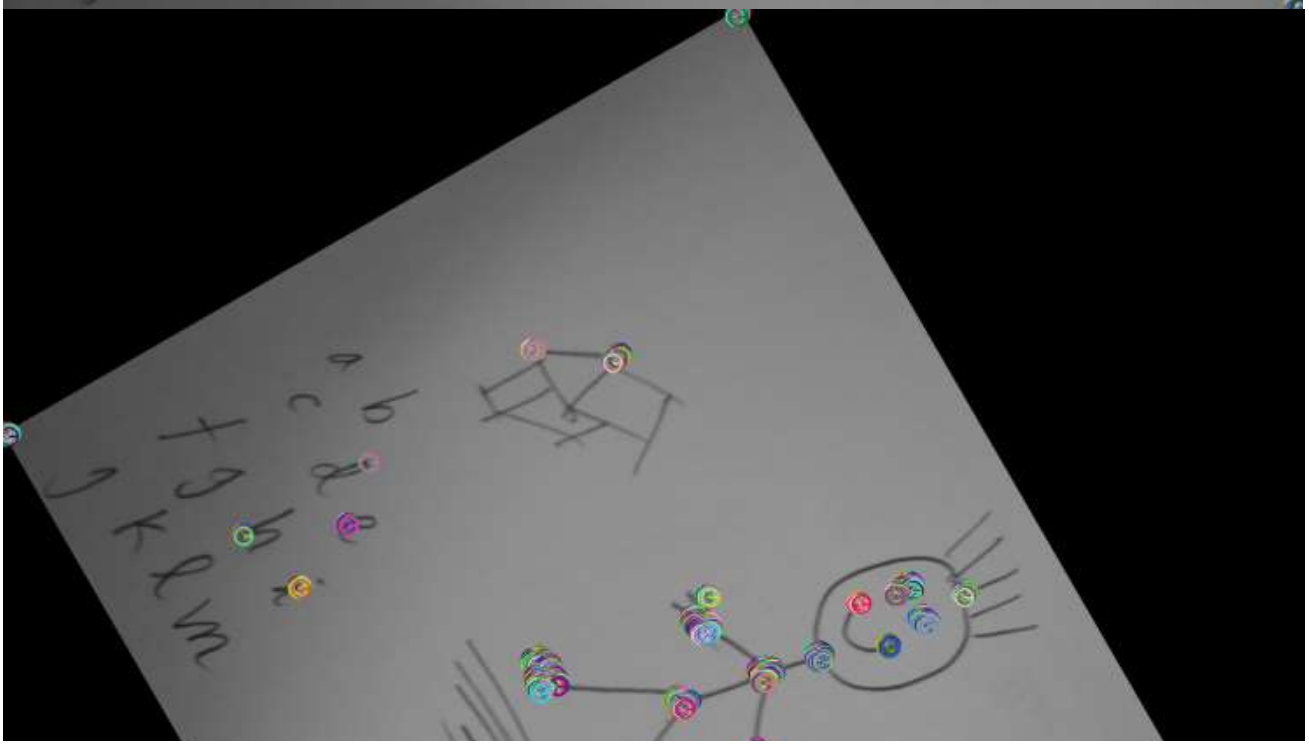
Puntos de interes

```
generate_results('uch101b.jpg', 60, 3e3,method='harris')
```

Resultados de filtro de Harris

Puntos de interes





De los resultados obtenidos utilizando el detector de Harris, se puede notar que es bastante robusto para la detección de esquinas, en particular, los resultados son prácticamente invariantes ante rotaciones en las imágenes, ya que se probó con diferentes ángulos en cada caso (30, 60 y 80) y se obtenian detecciones similares.

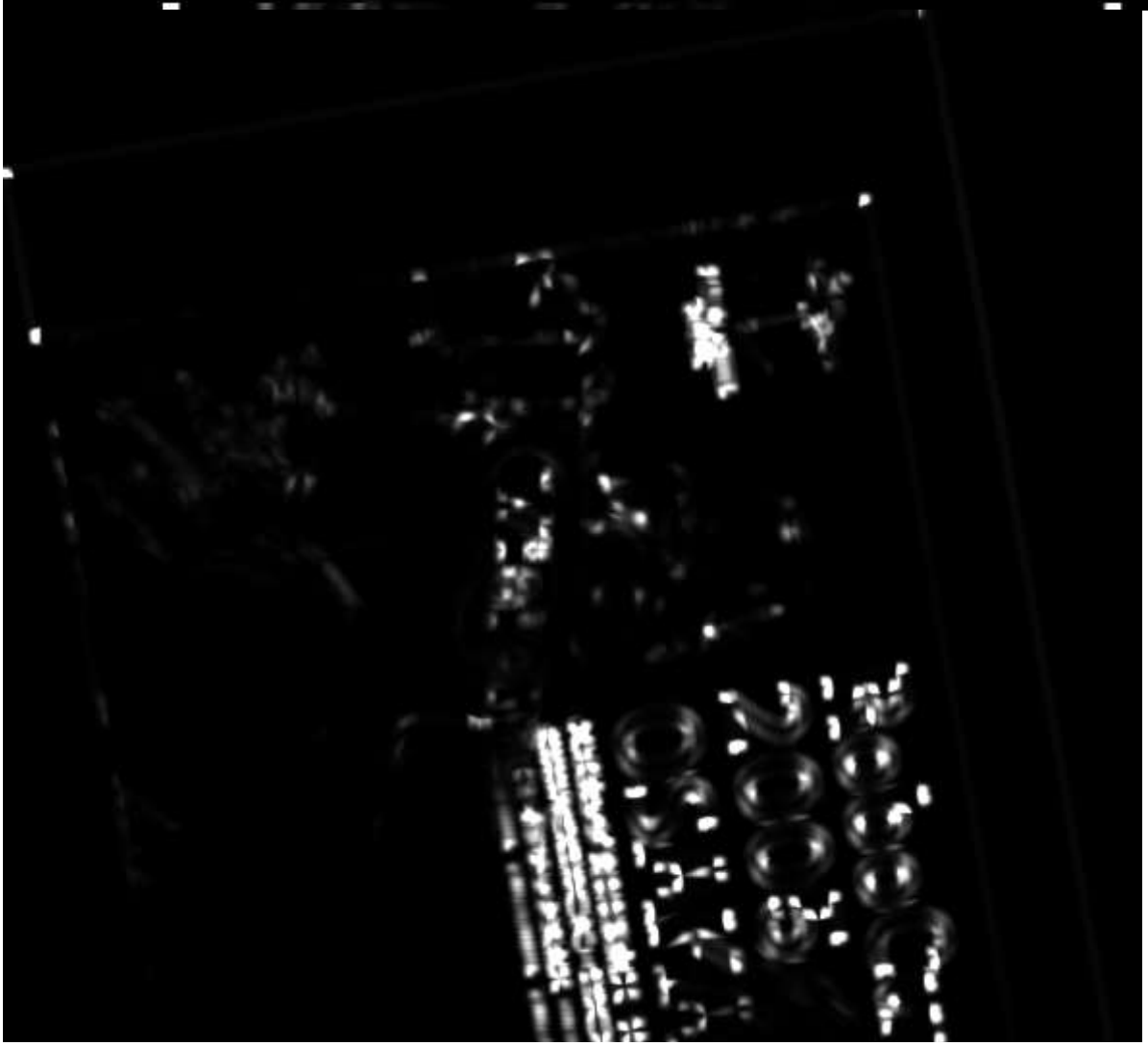
Pruebas con el detector de Shi-tomasi

```
generate_results('poligonos.png', 30, 1.5e2,method='shitomasi')
```

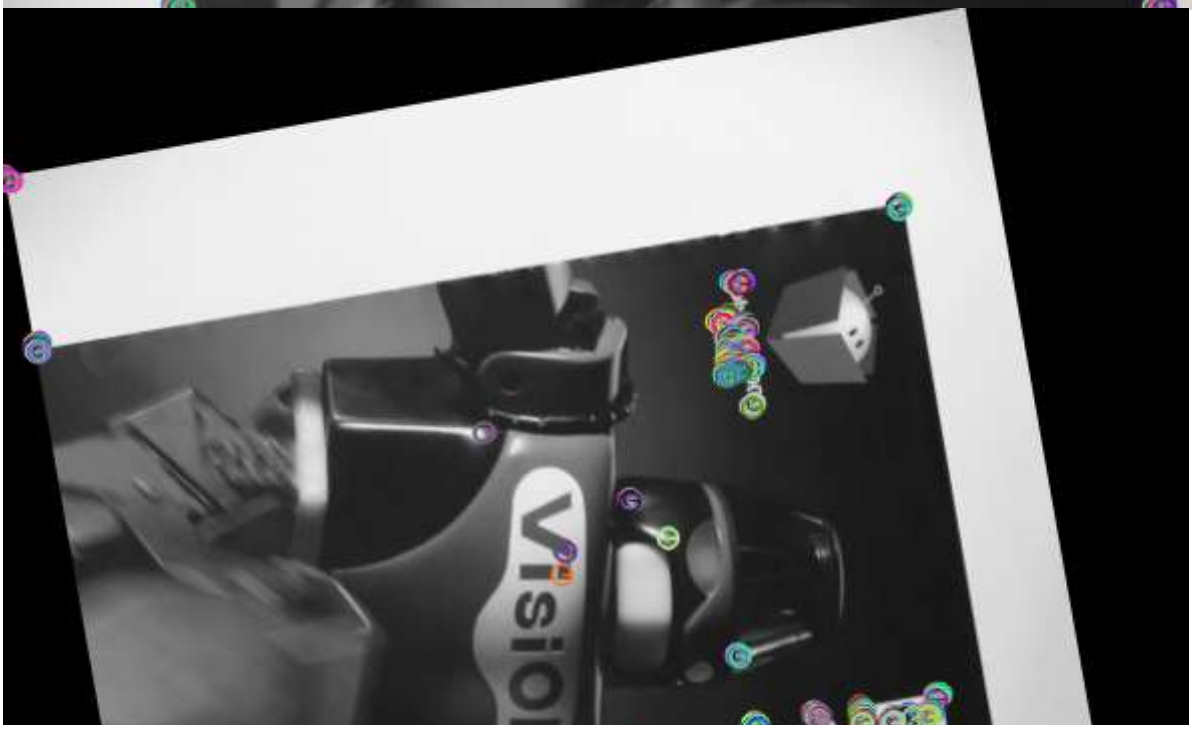
Resultados de filtro de Shi-tomasi

```
generate_results('uch006a.jpg', 80, 3e2,method='shitomasi')
```

Resultados de filtro de Shi-tomasi



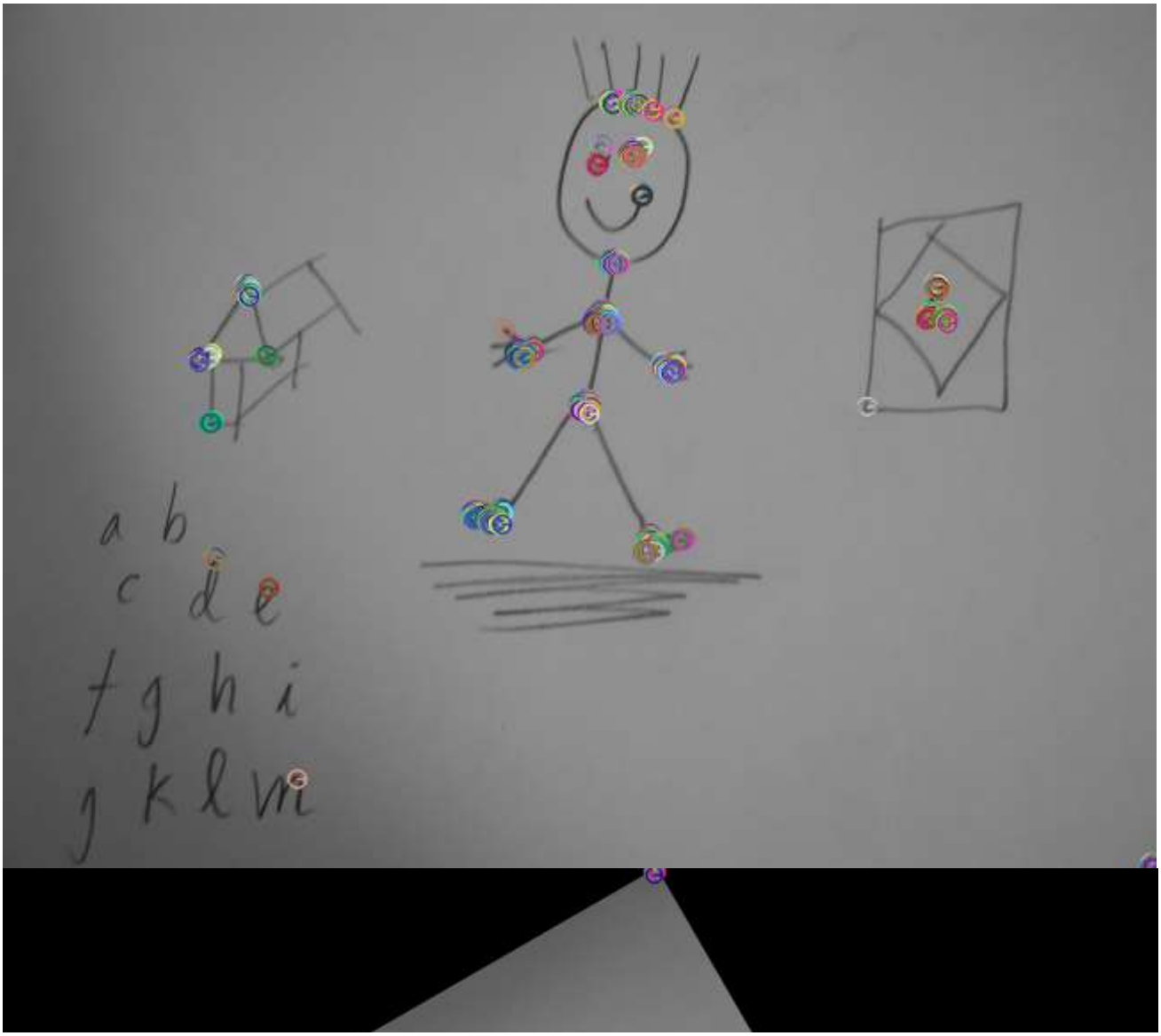
Puntos de interes filtro de Shi-tomasi



```
generate_results('uch101b.jpg', 60, 5e1,method='shitomasi')
```

Resultados de filtro de Shi-tomasi

Puntos de interes filtro de Shi-tomasi



De los resultados obtenidos utilizando el detector de Shi-tomasi, nuevamente se puede notar que es bastante robusto ante rotaciones en las imágenes, ya que se probó con diferentes ángulos en cada caso (30, 60 y 80) y se obtuvieron detecciones similares. Sin embargo, se notó que las detecciones en este caso son más debiles que en el detector de Harris, por lo que se necesitan thresholds más pequeños.

Una posible debilidad que se detecta en ambos métodos, es la dependencia del threshold para filtrar puntos en los resultados, puesto que para cada imagen fue necesario ocupar uno diferente. Esto finalmente puede ser un limitante para alguna aplicación de este sistema, ya que con un valor de threshold fijo, solo funcionaría en escenarios puntuales.

Alineamiento de imágenes

```
# Esta funcion ya esta lista, no debe ser modificada
def filterMatches(matches, kp1, kp2):
  # 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
```

Se crea la función *alineamiento*, la cual toma 2 imágenes (parte izquierda y parte derecha), y aplica el algoritmo de alineamiento.

Para esto, en primer lugar se generan los puntos de interés de ambas imágenes mediante el detector cv2.SIFT_create.

Luego se generan los matchings entre los puntos de interés encontrados utilizando el BFMatcher de opencv, el cual compara todos los primera imagen con todos los puntos de la segunda para encontrar el más cercano.

Luego, se filtran los matchings mediante la función *filterMatches* entregada por el cuerpo docente, y con esto se obtienen los puntos para hacer el alineamiento.

Con los puntos encontrados en el paso anterior, se calcula la homografía mediante la función *findHomography* de opencv, la cual consiste en una matriz de transformación.

A partir de esta homografía, se transforma la perspectiva de la segunda imagen mediante la función *warpPerspective* de opencv.

Finalmente se fuciona la imagen original izquierda, con la imagen derecha previamente transformada.

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

def alineamiento(imgleft, imgright):
  img1 = cv2.imread(imgleft,0)
  img2 = cv2.imread(imgright,0)
  print('Imagen Izquierda')
  cv2_imshow(img1)
  print('Imagen Derecha')
  cv2_imshow(img2)
  detector = cv2.SIFT_create()
```

```
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, kp1, kp2)

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

print('Puntos de matching entre las dos imágenes')
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
M, mask = cv2.findHomography(points2, points1, cv2.RANSAC)
# 2) Por hacer: transformar la imagen img2 usando la funcion warpPerspective( ) y la
#   La imagen de salida debe tener un tamano suficiente
#   Se recomienda que tenga el doble del ancho de la imagen de entrada
transform = cv2.warpPerspective(img2, M,(img2.shape[1]*2,img2.shape[0]))
# 3) Por hacer: copiar la imagen img1 a la imagen resultante del paso anterior (se ob
transform[:,:img2.shape[1]] = img1[:img2.shape[0],:img2.shape[1]]
# 4) Por hacer: mostrar la imagen fusionada
print('Imagen fusionada')
cv2_imshow(transform)
```

```
alineamiento('left1.png', 'right1.png')
```

Imagen Izquierda

Imagen Derecha

Puntos de matching entre las dos imágenes

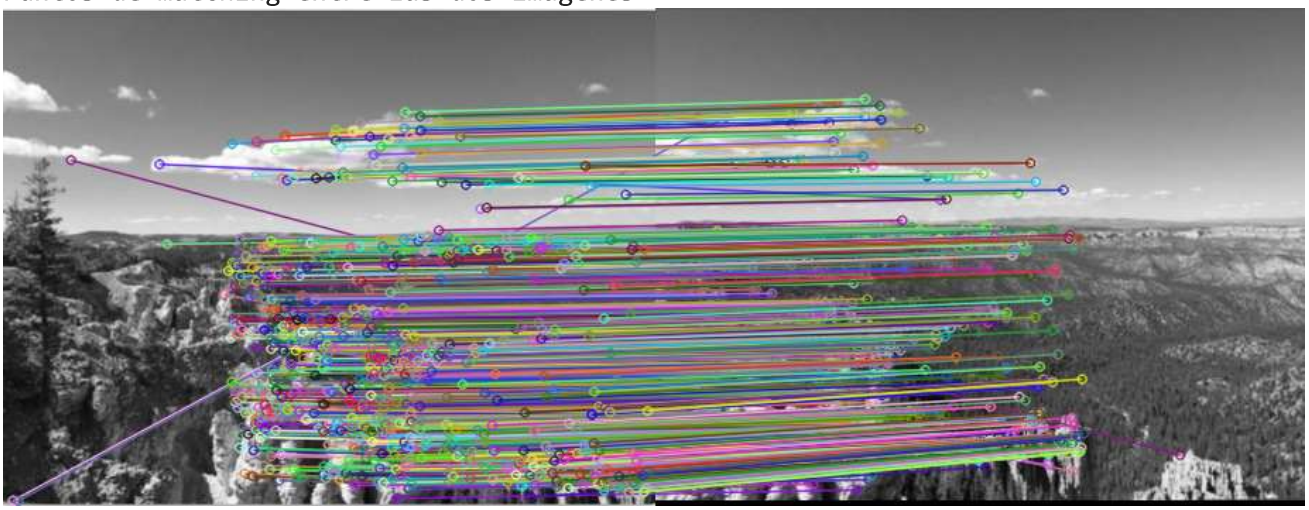


Imagen fusionada

```
alineamiento('left2.jpg', 'right2.jpg')
```

Imagen Izquierda

Imagen Derecha

Puntos de matching entre las dos imágenes

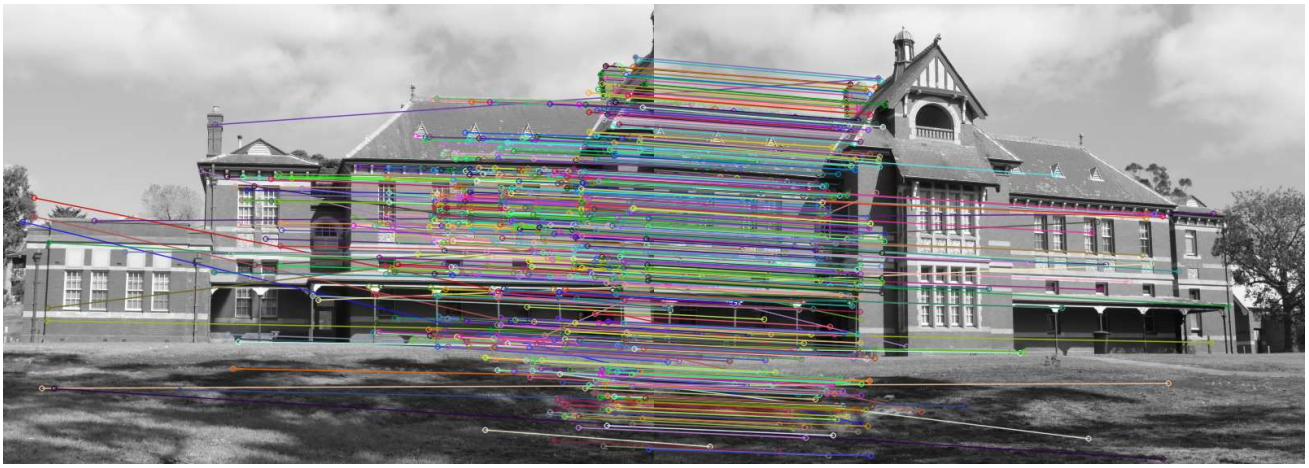


Imagen fusionada

```
alineamiento('left3.jpg', 'right3.jpg')
```

Imagen Izquierda

Imagen Derecha

Puntos de matching entre las dos imágenes

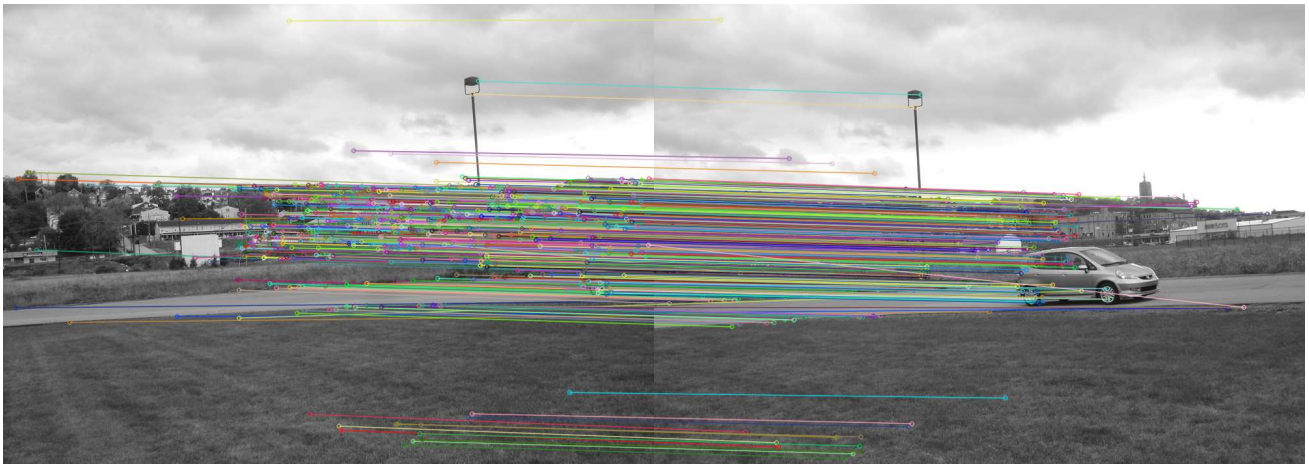


Imagen fusionada

```
alineamiento('left4.jpg', 'right4.jpg')
```

Imagen Izquierda

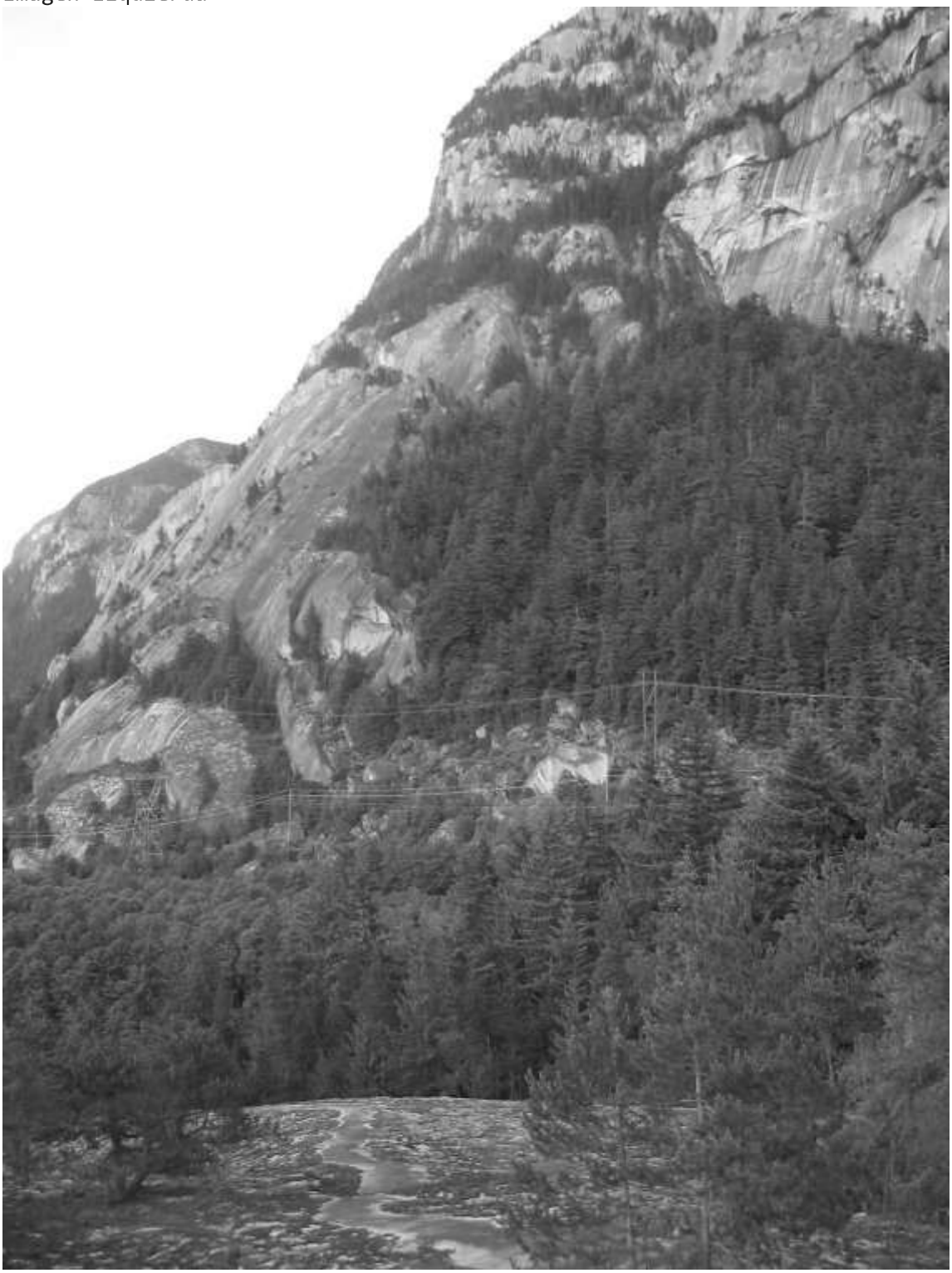


Imagen Derecha

Puntos de matching entre las dos imágenes

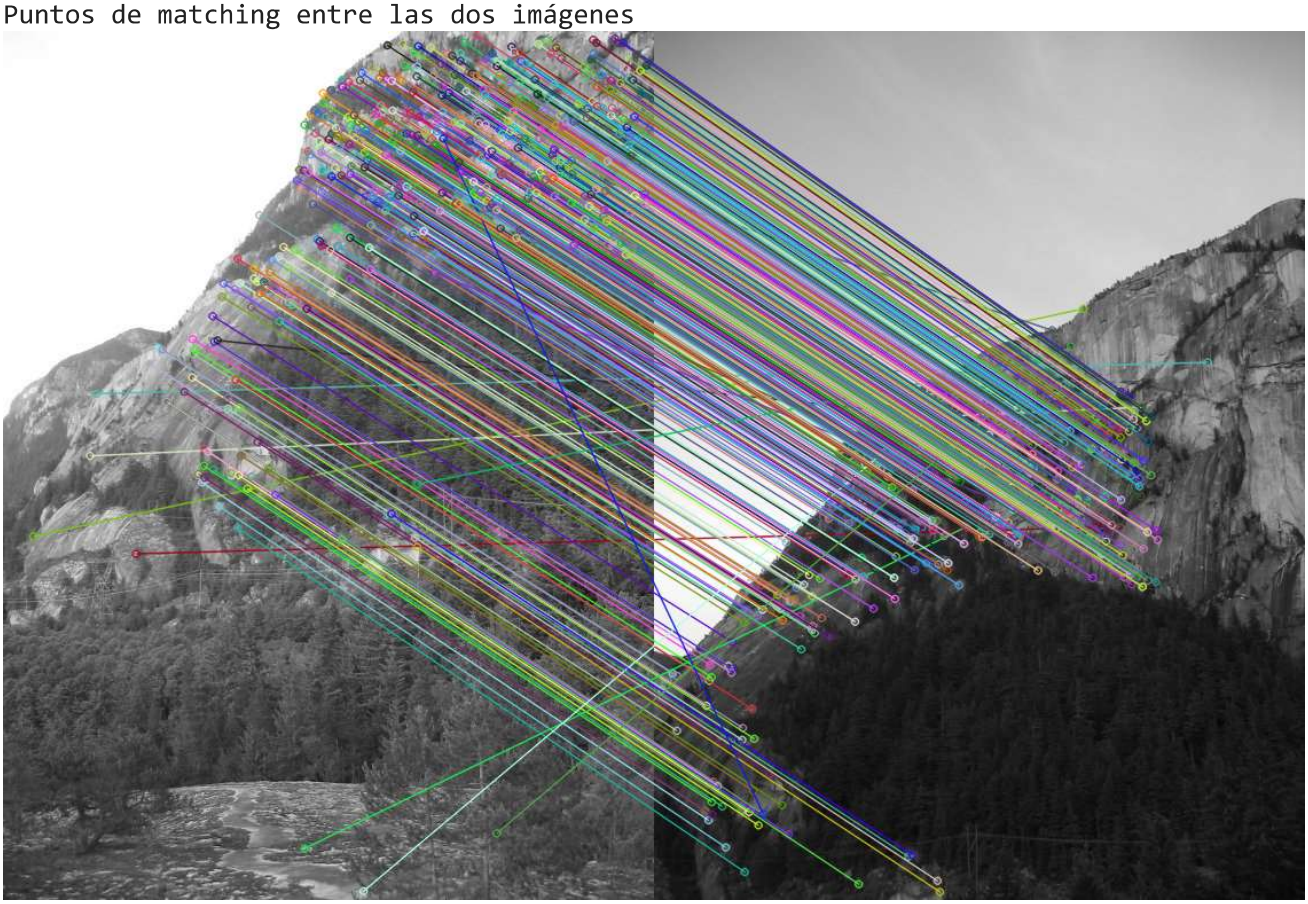


Imagen fusionada

En todos los casos de prueba de alineamiento de imágenes, se pueden observar buenos resultados, puesto que el sistema fue capaz de identificar la región de unión en las imágenes de entrada. De lo anterior, se puede concluir que el sistema es bastante robusto, considerando que en el útlimo caso, las imágenes *izquierda* y *derecha* no estaban tan bien definidas, ya que más bien una imagen enfoca una parte más baja de la montaña. Aún así, es capaz de extraer la parte de la montaña faltante de la imagen derecha y agregarla a la imagen izquierda.

Conclusiones

De los resultados obtenidos se puede concluir que se cumplió con los objetivos propuestos, ya que se implementaron tanto los detectores de puntos de interés, como el sistema de alineamiento de imágenes, ambos con resultados esperables.

En cuanto a los detectores, se puede decir que ambos son bastante robustos ante rotaciones, pero estan bastante limitados por el threshold, ya que diferentes imágenes necesitan diferente threshold.

```
1
```