

UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL DE CÓRDOBA

Trabajo Práctico De Laboratorio N°11

Lamas, Matías 65536
Navarro, Facundo 63809

Curso: 6r4
Grupo N°5

**Visión por computadora
Alineación de imágenes usando SIFT**

Docentes:
Ing. Araguás, Gastón
Ing. Redolfi, Javier

25 de junio de 2020

Índice

1. Consigna	2
2. Desarrollo	2

1. Consigna

Considerando los pasos detallados a continuación, realizar una alineación entre imágenes utilizando el algoritmo de SIFT.

- Capturar dos imágenes con diferentes vistas del mismo objeto.
- Computar puntos de interés y descriptores en ambas imágenes.
- Establecer matches entre ambos conjuntos de descriptores.
- Eliminar matches usando criterio de *Lowe*.
- Computar una homografía entre un conjunto de puntos y el otro.
- Aplicar la homografía sobre una de las imágenes y guardarla en otra (mezclarla con un alpha de 50 %).

2. Desarrollo

Importamos las librerías a usar, opencv y numpy.

```
import numpy as np
import cv2
```

Cargamos el par de imágenes a utilizar, y guardamos unas copias de las mismas para dibujar sobre ellas los puntos claves (*keypoints o kp*).

```
img1 = cv2.imread('img1.png')
img2 = cv2.imread('img2.png')
kp_img1 = img1.copy()
kp_img2 = img2.copy()
```

Para utilizar una instancia del detector de puntos claves *SIFT*, hay que llamar a la función *cv2.xfeatures2d.SIFT.create()*. Todos los argumentos de la función tienen valores predeterminados, los cuales son:

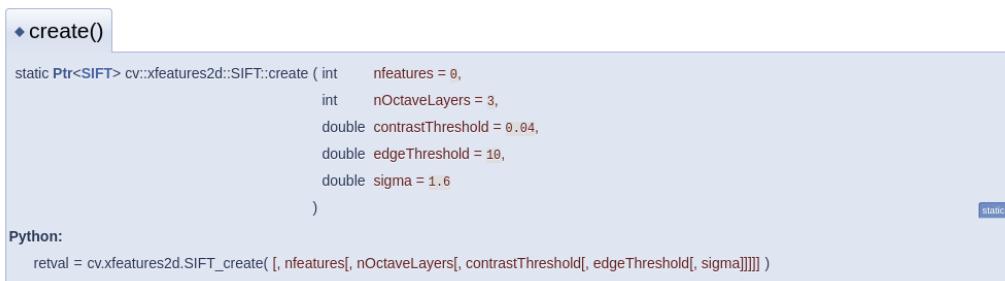


Figura 1: Función create() del descriptor SIFT.

En ese orden indican, el número de puntos claves a encontrar y retornar, número de niveles en la escala piramidal a usar, dos límites para ajustar la sensibilidad del algoritmo, la variación sigma para pre filtrar la imagen. Sin restarle importancia al resto de los argumentos, probablemente el más importante sea el primero para determinar la cantidad de puntos claves a buscar seguido por el sigma. Este último controla el tamaño máximo de los objetos que no son de interés, y es útil a la hora de remover el ruido o detalles innecesarios de la imagen.

```
dscr = cv2.xfeatures2d.SIFT_create(100)
kp1, des1 = dscr.detectAndCompute(img1, None)
kp2, des2 = dscr.detectAndCompute(img2, None)
```

Luego dibujamos los puntos encontrados en cada imagen, el resultado se puede ver en la imagen 2.

```
cv2.drawKeypoints(img1, kp1, kp_img1,(0, 255, 0), flags=cv2.
DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
cv2.drawKeypoints(img2, kp2, kp_img2,(0, 0,255), flags=cv2.
DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
```



Figura 2: Imágenes con puntos claves encontrados por SIFT.

Una vez encontrados los puntos claves por los descriptores, procedemos a encontrar las correspondencias entre ellos, opencv provee una gran variedad de herramientas para esto, el método más obvio y simple es el de comparar todos los posibles pares y elegir los mejores. Hay que destacar que este método es extremadamente lento.

Usamos la función `cv2.BFMatcher.create()`, el cual toma una bandera que configura una distancia para la comparación entre descriptores y habilita la bandera de chequeo cruzado.

```
matcher = cv2.BFMatcher(cv2.NORM_L2)
```

Una vez encontrada las correlaciones, se filtran los resultados de dos maneras distintas. Primero a través del método del vecino mas cercano (*knn*), al cual se le aplica una comprobación del radio, comprueba dos radios entre el primer y segundo elemento, si es mayor a 0.8 (criterio de *Lowe* = 0.7) se descarta, esto permite eliminar cerca del 90 % de falsas correspondencias, mientras que descarta solo el 5 % de correspondencias correctas.

```
matches_01 = matcher.knnMatch(des1, des2, k=2)
matches_10 = matcher.knnMatch(des2, des1, k=2)

# Guardamos los buenos matches usando el test de razon de Lowe
def ratio_test(matches, ratio_thr):
    good_matches = []
    for m in matches:
        ratio = m[0].distance / m[1].distance
        if ratio < ratio_thr:
            good_matches.append(m[0])
    return good_matches

RATIO_THR = 0.7 #0.7 LOWE
good_matches01 = ratio_test(matches_01, RATIO_THR)
good_matches10 = ratio_test(matches_10, RATIO_THR)
```

El segundo filtrado se hace comprobando si las correspondencias encontradas en *img2* para los puntos claves en *img1* son los mismo encontrados en *img1* para los puntos claves en *img2*, es decir que solo conservamos las correspondencias encontradas en ambas direcciones.

```
good_matches10_ = {(m.trainIdx, m.queryIdx) for m in good_matches10}
final_matches= [m for m in good_matches01 if (m.queryIdx, m.trainIdx) in good_matches10_]
```



Figura 3: Correspondencias entre las imágenes sin filtrar.



Figura 4: Correspondencias entre las imágenes luego de filtrado

El siguiente paso es calcular la matriz homográfica entre las dos imágenes utilizando el algoritmo de *RANSAC* (*RANdom SAmple Consensus*), a través de la función **cv2.findHomography()**, una vez que conseguimos la matriz, aplicamos la transformación perspectiva sobre una de las imágenes.

```
if(len(final_matches) > MIN_MATCH_COUNT):
    src_pts = np.float32([kp1[m.queryIdx].pt for m in final_matches]).reshape(-1, 1, 2)
    dst_pts = np.float32([kp2[m.trainIdx].pt for m in final_matches]).reshape(-1, 1, 2)

    H, mask = cv2.findHomography(dst_pts, src_pts, cv2.RANSAC, 3.0)

    wimg2 = cv2.warpPerspective(img2, H, img2.shape[:2][::-1])
```

Finalmente, ponderamos la imagen a la cual le aplicamos la transformación a un valor “alphaz la otra imagen con su complemento y las unimos, dando como resultado la imagen que se muestra en [5](#)



Figura 5: Resultado final de aplicar el descriptor SIFT y aplicar la transformación homográfica.