Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Computer Vision

Práctica 4. Features y Panorama

Autores:
Víctor Gallardo Sánchez (801159)
Nerea Gallego Sánchez (801950)

4 de mayo de 2023

Índice

1. Introducción

Se presenta la memoria del trabajo realizado para la práctica L4 y L5. En esta se explicarán todos los apartados que se han ido realizando en ambas prácticas, es decir, todo lo relacionado con la extracción de características y el emparejamiento de imágenes(L4) y todo lo relacionado con el cálculo de la homografía y la construcción de un panorama.

2. Implementación de un método de extracción de características y emparejamiento de imágenes

En esta sección se probó los extractores que proponía openCV de features: HARRIS, ORB, SIFT y AKAZE y posteriormente se desarrollaron dos métodos para el emparejamiento de imágenes(fuerza bruta y flann Matching).

Para la extracción de características como se ha comentado se probaron los 4 métodos proporcionados por openCV. En la imagen 1 se pueden observar las características obtenidas con el extractor de features SIFT.
En la imagen 2 se pueden observar las características obtenidas con el extractor de features HARRIS
En la imagen 3 se pueden observar las características obtenidas con el extractor de features ORB.
En la imagen 4 se pueden observar las características obtenidas con el extractor de features AKAZE.

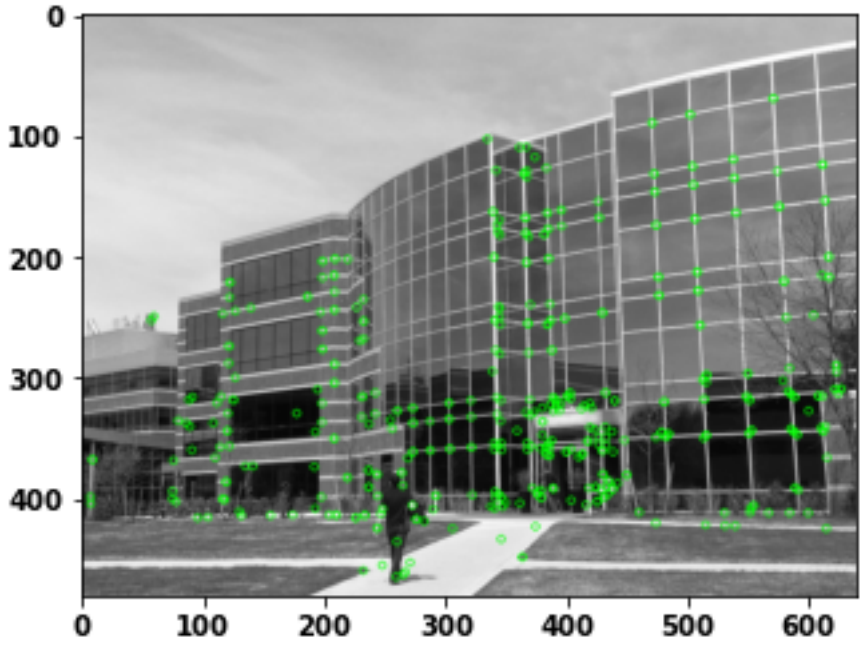

Figura 1: Features obtenidas con SIFT

A continuación, se pide realizar la búsqueda de emparejamientos por fuerza bruta, buscando el vecino más próximo y comprobando el ratio al segundo vecino. Como se comentó en clase, el detector de features HARRIS es un poco especial, por lo tanto, se ha realizado este método para los detectores de features SIFT, ORB y AKAZE.
Para realizar los emparejamientos por fuerza bruta, primero ha sido necesario obtener las características de dos imágenes. Una vez se han obtenido los descriptores de dos imágenes se ha creado un *DescriptorMatcher* (función de opencv) con la opción *BRUTE_FORCE_L1*. Es decir, que realice los emparejamientos por fuerza bruta con la norma L1. Se pide comprobar el ratio al segundo vecino. Para ello, se he la indicado al *DescriptorMatcher* con la función *knnMatch* pasándole como argumentos los descriptores de las características de ambas imágenes e indicandole que se quire comprobar el segundo vecino. Este *DescriptorMatcher* devuelve los emparejamientos de features. Por último queda comprobar que el ratio al segundo vecino es el adecuado. Para ello, se eligen como matches buenos aquellos que están a una distancia menor que 0.8 que la distancia que hay al segundo vecino.

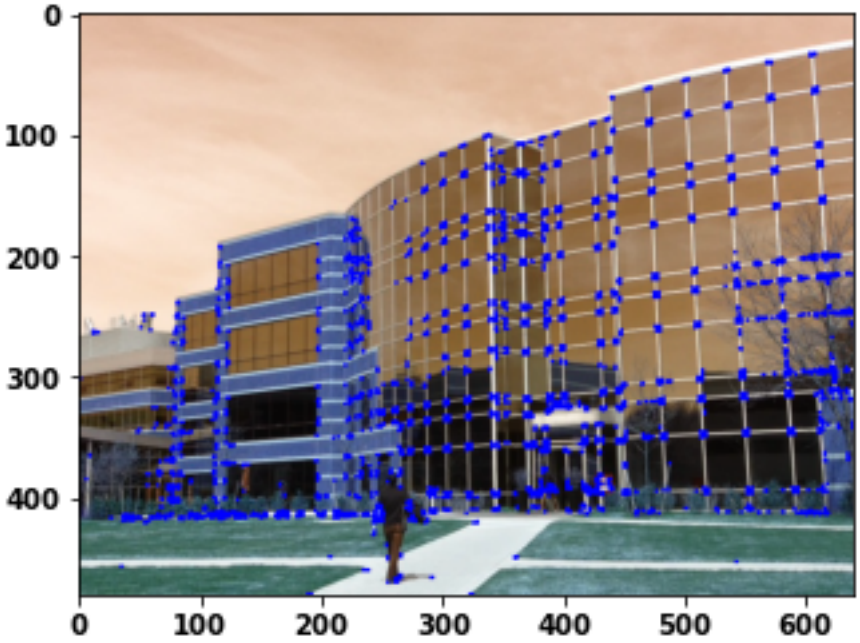

Figura 2: Features obtenidas con HARRIS

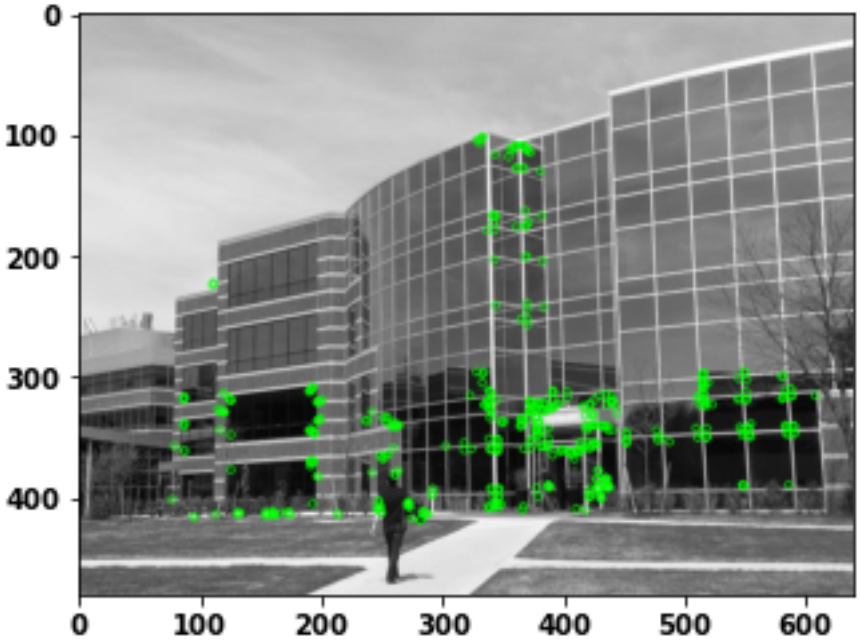

Figura 3: Features obtenidas con ORB

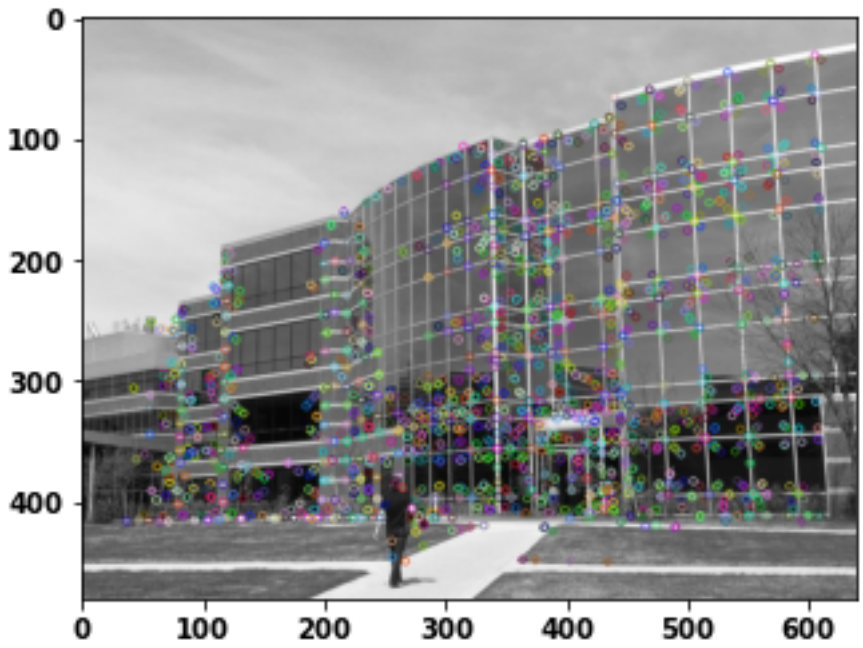

Figura 4: Features obtenidas con AKAZE

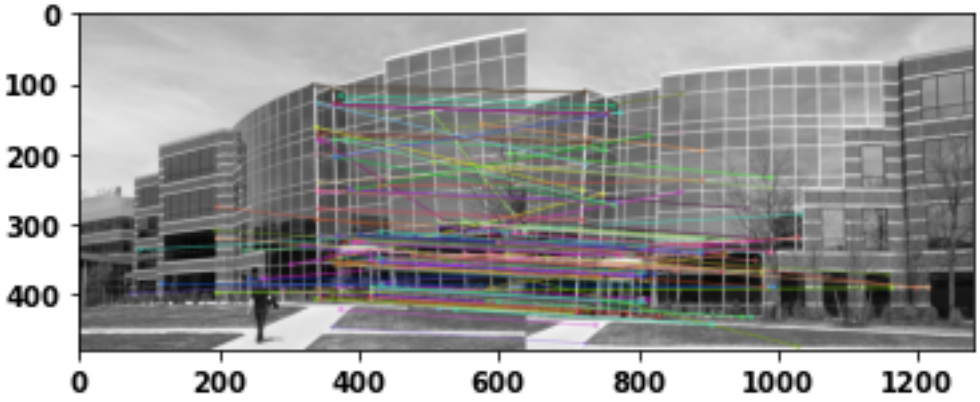

Figura 5: Matches con el método de fuerza bruta para SIFT

En la imagen 5 se muestran los matches obtenidos de dos imágenes con el método de fuerza bruta.
En la imagen 6 se muestran los matches obtenidos de dos imágenes con el método de fuerza bruta.
En la imagen 7 se muestran los matches obtenidos de dos imágenes con el método de fuerza bruta.

2.1. Opcional. Flann Matching

En esta sección se va a comentar como se ha implementado el opcional del Flann matcher. Para ello se ha realizado una función que recibe como parámetros las dos imágenes que se quieren emparejar, sus keypoints y descriptores.

Se utiliza el emparejador de flann que trae openCV, para ello hay que configurarlo y utilizar el método de knnMatch para poder tener en cuenta la distancia al segundo vecino.
Finalmente de todos los matches que obtiene el emparejador, únicamente se quieren devolver los matches buenos. Por lo que se selecciona un ratio(en este caso 0.7 de acuerdo al ratio propuesto por Lowe) y se mira la distancia entre matches. Si la distancia entre 2 matches es buena, (- de 0.7) se decide que ese match es bueno y se añade al resultado. Después se dibuja el emparejamiento obtenido que se puede observar en las figuras 8, 9 y 10

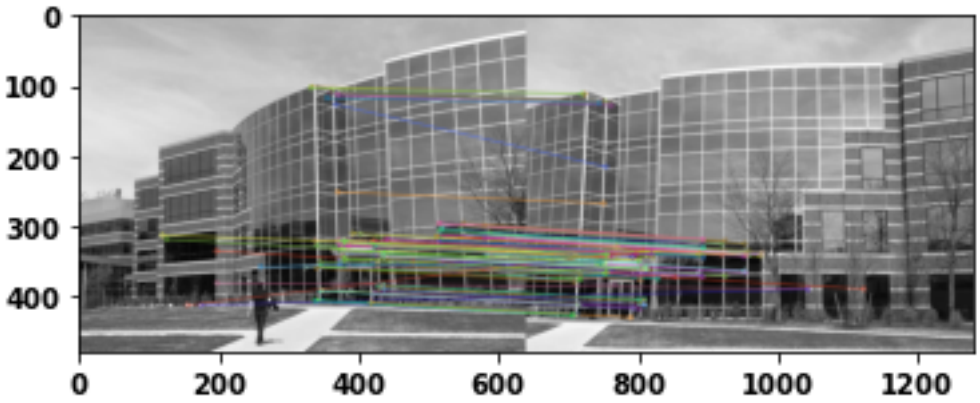

Figura 6: Matches con el método de fuerza bruta para ORB

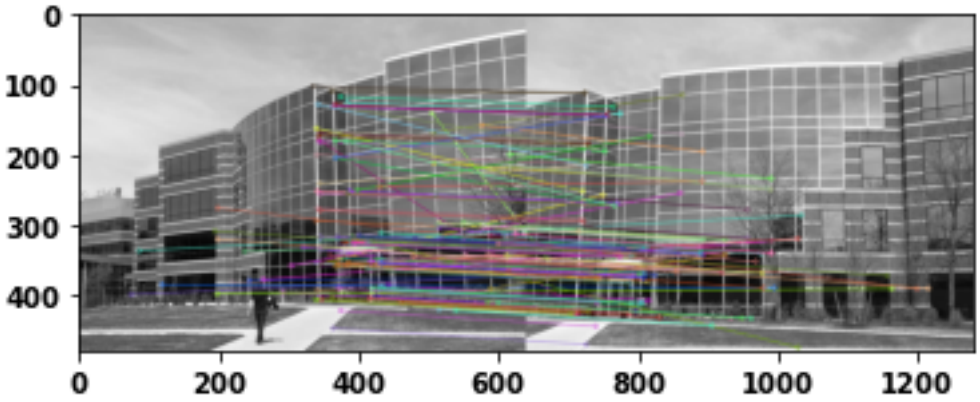

Figura 7: Matches con el método de fuerza bruta para AKAZE

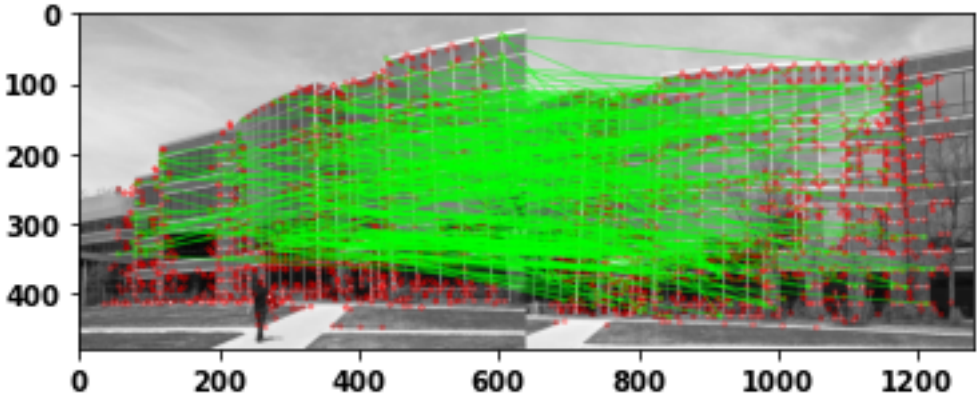

Figura 8: Matches con el método de FLANN para AKAZE

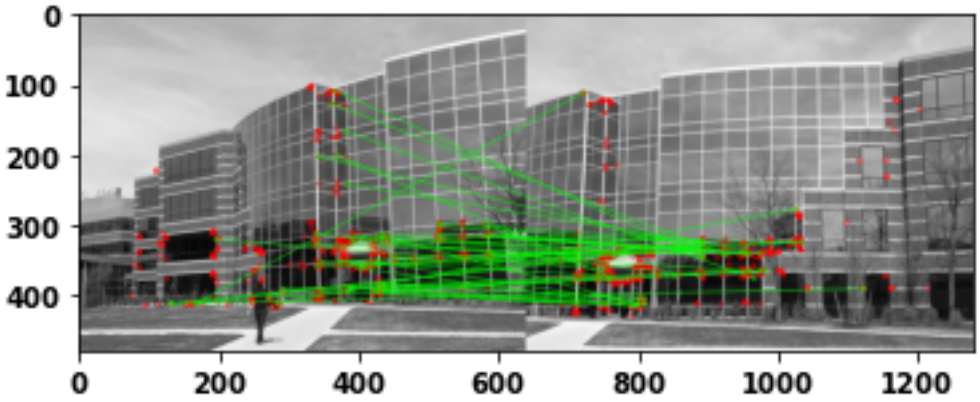

Figura 9: Matches con el método de FLANN para ORB

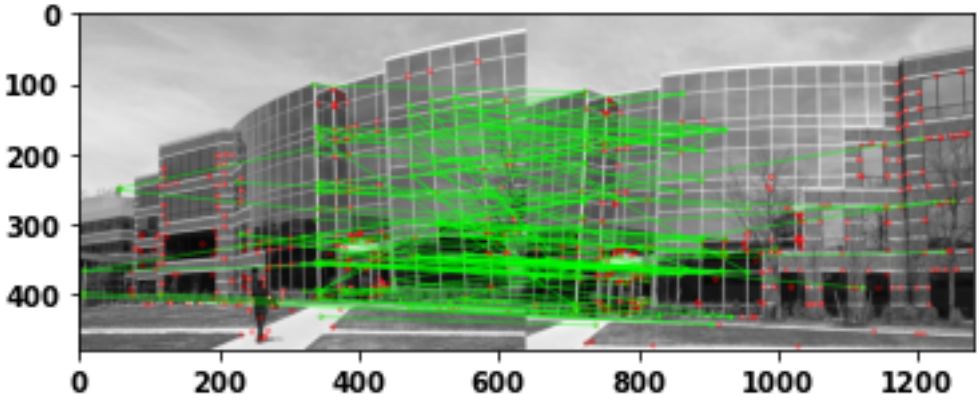

Figura 10: Matches con el método de Flann para SIFT

Asimismo, se muestra una tabla que compara los tiempos de ejecución del método de Flann vs método de fuerza bruta. Se han obtenido para 3 extractores de características: SIFT, ORB, AKAZE. De esta manera la primera fila es SIFT, la segunda ORB y la tercera AKAZE.

Flann	BruteForce
0.0638282299041748	0.03689980506896973
0.01894521713256836	0.00698089599609375
0.07679414749145508	0.16406822204589844

Como se puede observar depende el tipo de extractor los tiempos varían y es mas rápido uno u otro. Por lo tanto, es decisión del programador utilizar un método u otro según le convenga.

2.2. Estudio comparativo

En esta sección se va a realizar una comparativa de los 3 métodos de extracción de características que se han probado(SIFT,AKAZE y ORB). En ella se mostrarán tiempos de ejecución, número de características, como varían y a que conclusiones se llega.
Como se comentó anteriormente, el extractor de HARRIS es algo especial, por lo cual no se ha considerado su uso en esta práctica.

En la tabla siguiente se muestran los tiempos de ejecución en orden SIFT, ORB y AKAZE. Se han hecho para nFeatures = 500, aunque AKAZE siempre saca todos los que puede.

Tiempos
0.1421353816986084
0.022936344146728516
0.11469268798828125

Asimismo, se muestra una tabla con los matches buenos que obtiene cada método de extracción para Flann y para BF. Nuevamente la primera fila es SIFT, la segunda ORB y la tercera AKAZE.

Flann	BF
500	174
500	145
1517	608

De esta tabla se pueden sacar varias conclusiones:

- Flann siempre coge todos los matches como matches buenos. Se puede observar ya que se sabe que SIFT y ORB eran 500 features y ha obtenido 500 matches. Esto le hace tener menos precisión ya que esta cogiendo matches que realmente no lo son.
- Por otra parte, BF es mas selectivo, se observa como devuelve muchos menos matches que el Flann asegurándose de que estos sí que son buenos. Esto en imágenes que sean sencillas no tiene un efecto tan dispar, ya que ambos métodos obtienen resultados similares. Sin embargo, si se busca emparejar imágenes mas complejas, el BF lo hará mejor puesto que Flann seguro que se equivoca en una gran cantidad de matches.
- Por otra parte, se puede observar lo que se mencionó de que AKAZE siempre saca el máximo número de features que puede. Esto lo sabemos ya que flann coge todos los features como matches buenos y se ve que en AKAZE ha cogido 1517.

Si se estudian de manera cualitativa las imágenes se pueden contrastar estos puntos que se han mencionado. Por ejemplo en las figuras mostradas anteriormente(por ejemplo la 7 y 8) se observa perfectamente este fenómeno. Flann esta cogiendo **todos** las features como matches buenos, por eso se obtienen tantas líneas verdes que representan los matches. Mientras tanto, el método de BF es mucho mas selectivó como se comentó en el estudio cuantitativo por lo que el número de matches que aparecen en la figura son mucho menores.

2.3. Opcional. Estudiar los parámetros de los detectores de features

Una de las tareas opcionales es estudiar los parámetros de los métodos de detección de features.

En cuanto a la función de SIFT se han estudiado los siguientes parámetros:

```
"""_summary_ Devuelve los keypoints, los descriptores, el tiempo de detecci n y la
    cantidad de caracter sticas

Args:
    nfeatures: number of features returned
    contrstThreshold: filter out weak features in semi-uniform (low-contrast) regions.
    The larger the threshold, the less features are produced by the detector.
    sigma: The sigma of the Gaussian applied to the input image at the octave #0

"""
def SIFT_keypoints(gray, nfeatures : int, contrastThreshold=0.04, sigma=1.6):
    start = time.time()
    sift = cv2.SIFT_create(nfeatures, contrstThreshold=contrastThreshold, sigma=sigma)
    kp, desc = sift.detectAndCompute(gray,None)
    end = time.time()
    return kp, desc, end - start, len(kp)
```

Para comenzar, se puede elegir la cantidad de features que queremos que saque el detector. A continuación, se ha añadido *constrastThreshold* que es un umbral para filtrar los rasgos débiles de las regiones semiuniformes. Cuanto mayor sea el threshold, menos features se obtendran. Por último, se ha añadido el parámetro *sigma* que consiste en el parámetro de la función gaussiana que se aplica al detector.

```
    """summary Devuelve los puntos de HARRYS
        Args:

            blockSize:    Neighborhood size
            ksize:    Aperture parameter for the Sobel operator.
            k:    Harris detector free parameter.
    """
def HARRIS_keypoints(gray, blockSize=2, ksize = 3, k = 0.04):

    dst = cv2.cornerHarris(gray,blockSize, ksize , k)
    #result is dilated for marking the corners, not important
    dst = cv2.dilate(dst,None)

    return dst
```

Para la función HARRIS se han estudiado 3 parámetros. En primer lugar blockSize, este es el tamaño del vecindario donde se quiere buscar, es decir, con cuántos vecinos cercanos se queda el extractor.
En segundo lugar, kSize representa el parámetro de apertura del operador de Sobel y por último la K es un parámetro de ajuste, modificándolo se pueden conseguir obtener más o menos keypoints. El 0.04 es un valor típico que se suele utilizar bastante.

Para la función ORB se han estudiado los siguientes parámetros:

```
# https://docs.opencv.org/3.4/d1/d89/tutorial_py_orb.html
    """_summary_ Devuelve los keypoints, los descriptores, el tiempo de detecci n y la
    cantidad de caracter sticas
        Args:

            nfeatures: number of features returned
            edgeThreshold:  This is size of the border where the features are not
    detected.
    """
def ORB_keypoints(gray, nfeatures = 500, edgeThreshold = 31):

    # Initiate ORB detector
    start = time.time()
    orb = cv2.ORB_create(nfeatures, edgeThreshold = edgeThreshold)
    # find the keypoints with ORB
    kp = orb.detect(gray,None)
    # compute the descriptors with ORB
```

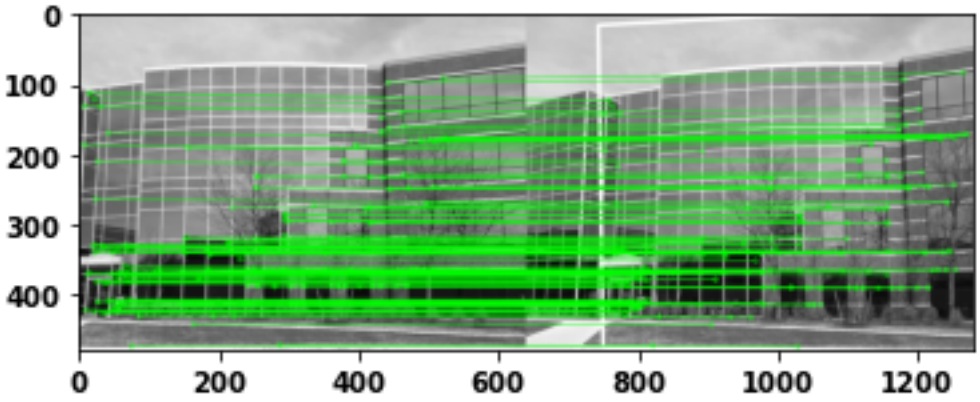

Figura 11: Matches para SIFT y FLANN en homografía de openCV

```
    kp, des = orb.compute(gray, kp)
    end = time.time()
    # draw only keypoints location,not size and orientation
    return kp, des.astype(np.float32), end - start, len(kp)
```

Como ocurría en los casos anteriores se ha añadido un parámetro para indicar la cantidad de features que se quieren obtener. Además, se ha añadido el parámetro *edgeThreshold* que es el umbral a partir del que las features ya no se detectan

Para la función AKAZE no se han encontrado parámetros interesantes que estudiar.

Todas las funciones de descriptores devuelven los keypoints, los descriptores de los mismos, el tiempo de detección de features y la cantidad de features devueltas.

3. Panoramas

En esta segunda parte del trabajo se pide calcular un panorama mediante homografias calculadas con RANSAC. En primer lugar se va a ahondar en los detalles del cálculo de la homografía, explicando como se ha hecho y que resultados se obtienen. Después, se hablará de la construcción del panorama como tal.

3.1. Cálculo de la homografía

Para una primera versión se utilizó el findHomography que viene con openCv el cual permite pasarle un parámetro cv2.RANSAC para que calcule el RANSAC de manera automática con las cuatro esquinas de la imagen a la que se le aplica la homografía.
Esta función utiliza SIFT y el emparejador de FLANN y con los matches obtenidos utiliza findHomography para obtener la matriz de homografía. Finalmente se muestra la imagen 11 con la homografía calculada.

En una segunda versión se realizó una implementación propia del algoritmo de RANSAC para calcular la homografía. Para ello, se sacan los puntos característicos con la función SIFT y se obtienen los matches con la función de FLANN. A continuación, es necesario iterar hasta que se obtenga una homografía correcta. Los pasos para obtener la homografía son:

- Se eligen 4 matches aleatorios
- Con los cuatro puntos aleatorios se calcula la homografía.
- Una vez obtenida la posible homografía, se calcula cuántos de los puntos restantes con una transformación de la homografía se encuentran a una distancia máxima de dos píxeles del punto en la segunda imagen.

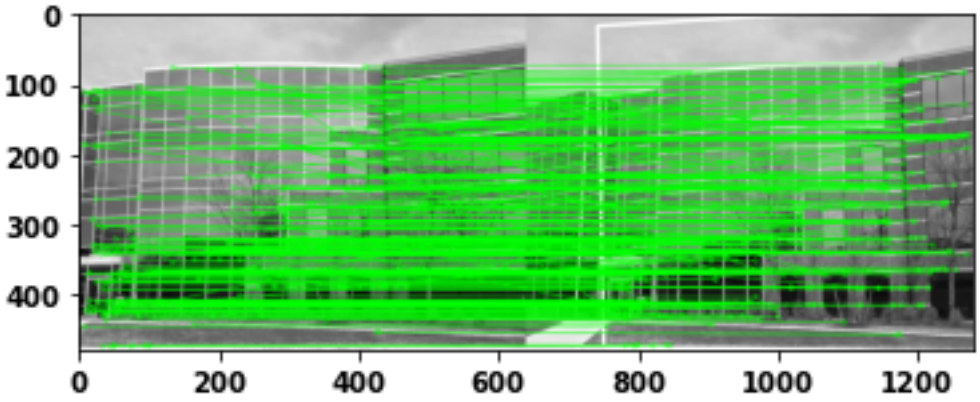

Figura 12: Matches para SIFT y FLANN en homografía propia

- Se realiza el cálculo de la homografía hasta que pasen 30 segundos, o se encuentre una homografía en la que al menos 20 matches sean buenos.

Se ha decidido limitar el tiempo a 30 segundos de iteraciones ya que puede haber ocasiones en las que no se pueda añadir la imagen al panorama porque los matches sean insuficientes.
Una vez finaliza se devuelve la matriz de homografía y si se ha podido calcular correctamente.
En la imagen 12 se puede observar el cálculo de la homografía propia.

3.2. Construcción del panorama

Una vez obtenida una matriz de homografía de una imagen que se puede añadir al panorama solo queda construir las imágenes del panorama.
Para ello, primero se cogen las esquinas de la imagen original y las esquinas de la imagen a la que se aplica la homografía con la perspectiva necesaria (calculada con la matriz de homografía obtenida). Una vez se obtienen las esquinas de ambas imágenes, se calcula la x e y máximas y mínimas. Con esta información ya se puede construir la matriz de translación.

Una vez se obtiene la matriz de translación, se aplica *warpPerspective* con la homografía y con la translación a la imagen a la que se le debe aplicar la homografía. Por último se añade la imagen no modificada en el hueco de la translación realizada.

A continuación se añaden unos cuantos resultados de los panoramas obtenidos:
En la imagen 13 se encuentra el panorama del edificio con el detector de características SIFT.
En la imagen 14 se encuentra el panorama del edificio con el detector de características ORB.
En la imagen 15 se encuentra el panorama del edificio con el detector de características AKAZE. Se puede observar que SIFT realiza un panorama más correcto que los otros dos extractores de features, por lo tanto, se ha decidido calcular el resto de panoramas con el detector de características SIFT.
En la imagen 16 se muestra la imagen de poster con el detector de características SIFT.
En la imágenes 17 y 18 se muestra la imagen del panorama de dos imágenes 3d construidas con el detector de características SIFT.

4. Opcional. Construcción de un panorama en vivo

Se ha decidido construir como opcional un panorama en vivo. Para ello se ha reutilizado todas las implementaciones explicadas anteriormente.

Para construir el panorama en vivo se ha modificado la función que devuelve la homografía y en este caso, la función indica que se puede añadir en los mismos casos que antes y además debe cumplir que se debe hacer una translación de al menos 20 píxeles y menos de 150 píxeles. De esta manera se garantiza que haya solape entre las imágenes.
Por lo tanto, la cámara va tomando imágenes constantemente y se van calculando la homografía y en caso de que haya matches suficientes y haya separación suficiente entre las imágenes, se añade al panorama

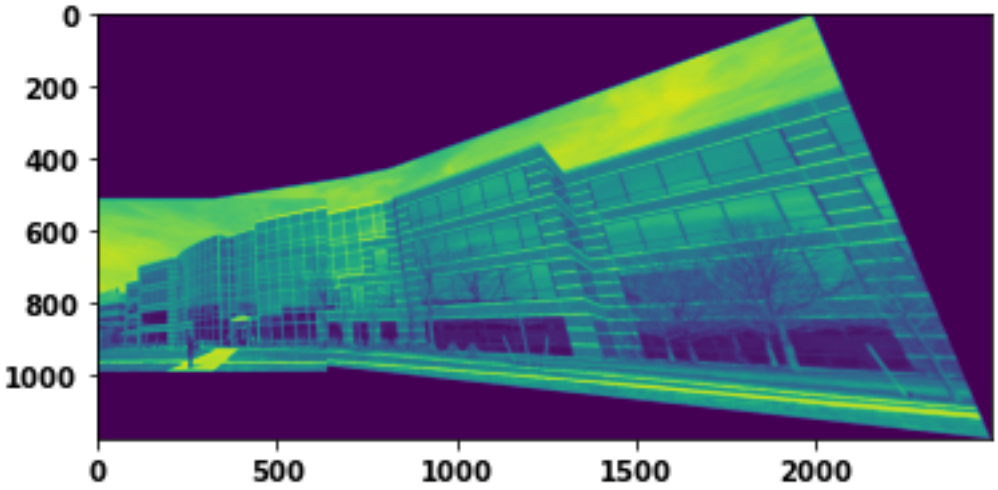

Figura 13: Panorama del edificio con SIFT

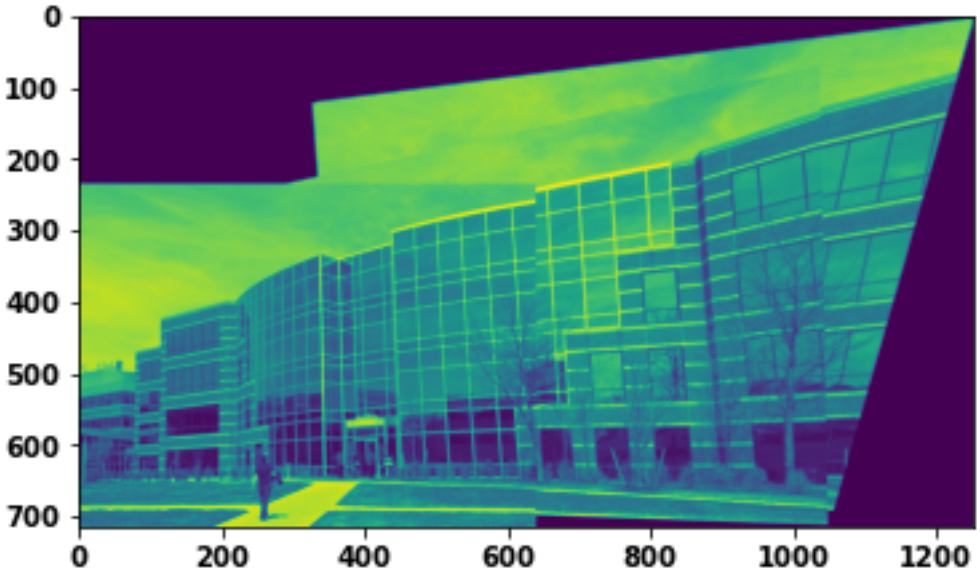

Figura 14: Panorama del edificio con ORB

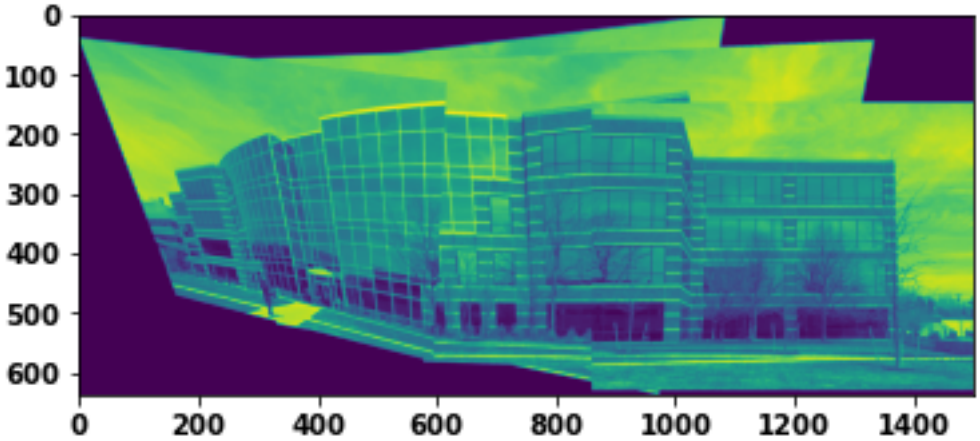

Figura 15: Panorama del edificio con AKAZE

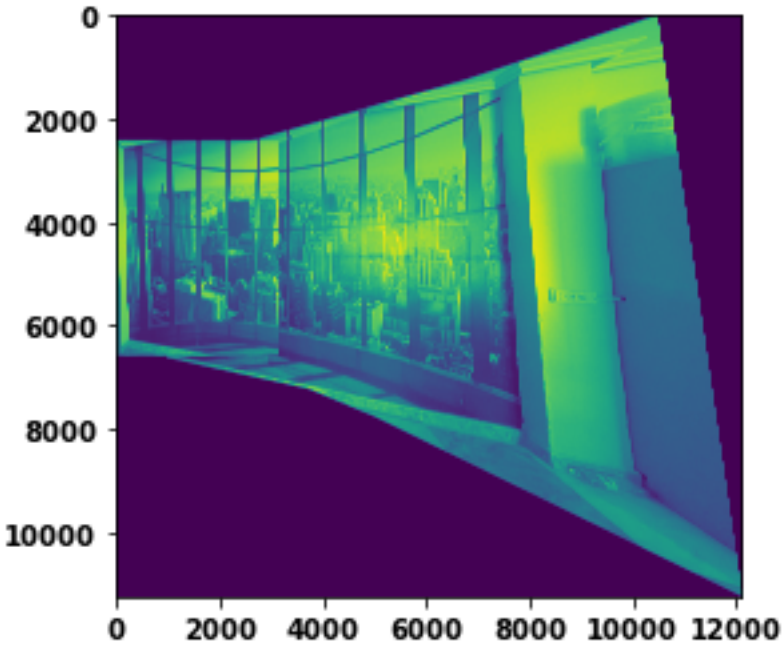

Figura 16: Panorama del poster con SIFT

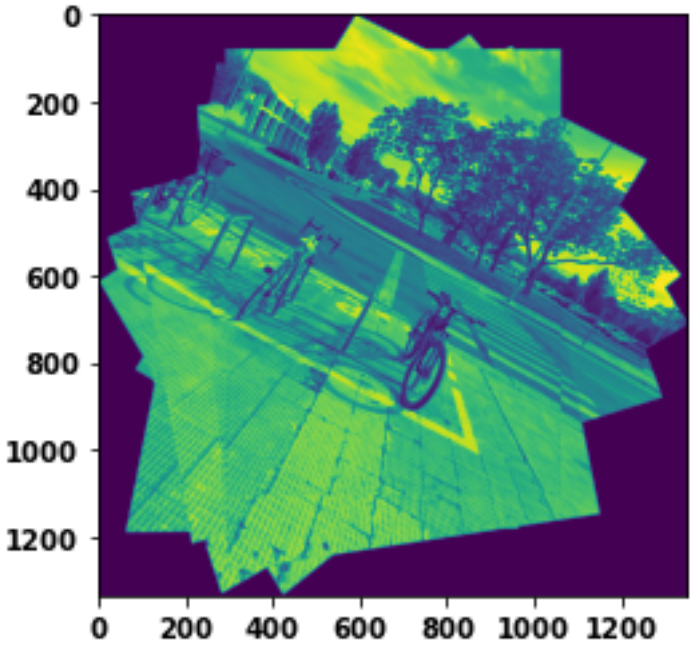

Figura 17: Panorama de un escenario 3d con SIFT

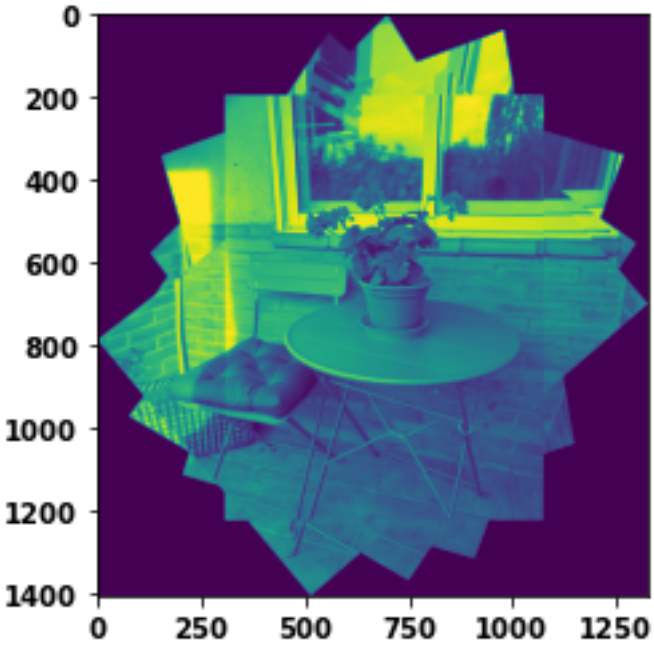

Figura 18: Panorama de un escenario 3d con SIFT

La demostración se encuentra en el video del siguiente enlace: https://drive.google.com/file/d/1SCMU8j94Am1YMyjmg7U7DEavXpMbXxEw/view?usp=sharing

5. Opcional. Debugger

Como opcional se ha decidido instalar la herramienta de Debugger para el lenguaje de programación Python Spyder con un enviroment basado en miniconda. Se ha dedido usar el enviroment de minconda ya que es mucho más ligero que otros enviroments como puede ser anaconda.
Para realizar la instalación primero es necesario instalar miniconda. Una vez se ha instalado miniconda, se crea el enviroment de Pyhton que se va a usar con los comandos de conda necesarios. A continuación se va a instalar la herramienta Spyder. Una vez instaladas ambas herramientas solo queda configurar el enviroment de Python en Spyder. Para ello accedemos en Tools > Preferences > Python interpreter. En este momento debemos encontrar la ventana contenida en la imagen 19. Seleccionamos la opción *Use the following interpreter* y buscamos en nuestro directorio el enviroment de anaconda que hemos instalado. Para ello, hay que buscar en el directorio donde hayas instalado miniconda y en la carpeta *envs* se encuentran todos los enviroments que hayas creado. Seleccionas el que más te interese y solo queda refrescar la terminal del debugger para poder trabajar con el enviroment deseado.

6. Esfuerzos dedicados

	Día	Hora	Tareas
Víctor	17/04/2023	3	Sesión de prácticas: L4
	27/04/2023	3	Sesión de prácticas: L5
	30/04/2023	2h	fixing panorama
	02/05/2023	3 30h	fixing panorama + tutoria
	03/05/2023	2 30h	fixing panorama
	04/05/2023	2 40h	Memoria
	Total	16,4 horas	

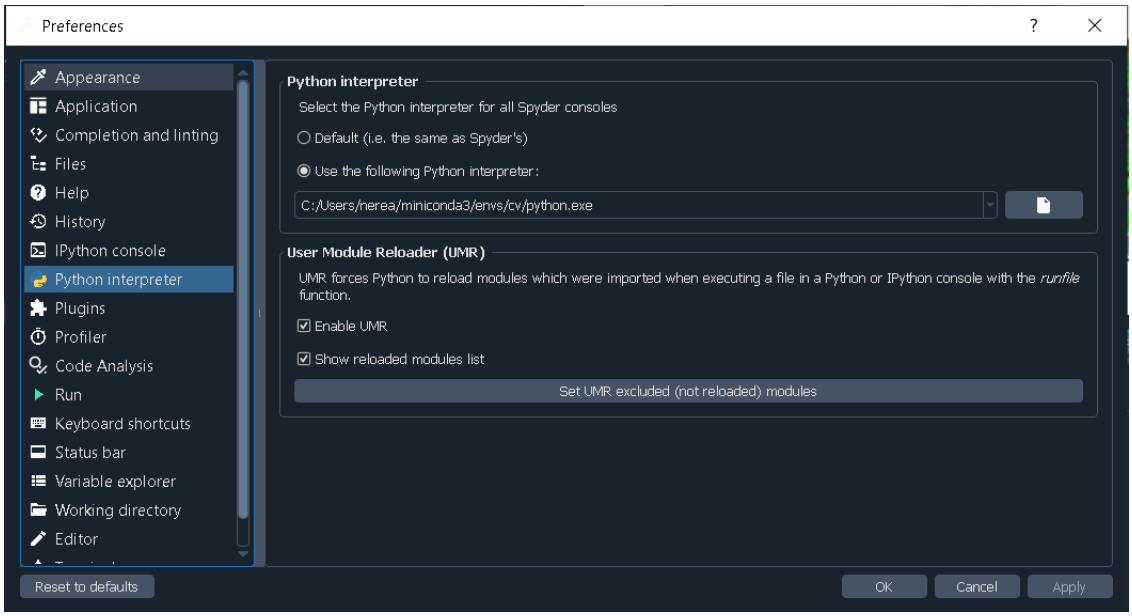

Figura 19: Ventana de configuración del enviroment

	Día	Hora	Tareas
	17/04/2023	2,5	Sesión de prácticas: L4
	26/04/2023	0,17	poster scene
	26/04/2023	0,78	modificación de parámetros
	27/04/2023	3	Sesión de prácticas: L5
	29/04/2023	1,75	construct panorama
Nerea	02/05/2023	3,28	construct panorama
	02/05/2023	0,5	tutoria
	03/05/2023	2,58	construct panorama
	03/05/2023	0,33	tutoria
	03/05/2023	1	panorama en vivo
	04/05/2023	3,67	memoria
	Total	19,57 horas	