

Tarea 2: Keypoints & Descriptor Matching

Estudiante: Luis Escares

Profesor: Javier Ruiz del Solar

Auxiliar: Patricio Loncomilla

4 de Septiembre de 2019

I. INTRODUCCIÓN

En este informe se presenta el desarrollo de dos actividades principales. La primera es el análisis visual de la robustez ante transformaciones geométricas de tres tipos de detectores de puntos de interés; para lo cual se implementa una función que escala a la mitad del tamaño a las imágenes y otra que las rota. Luego, se implementa la detección de puntos de interés mediante ORB, DoG (SIFT) y GFTT, sobre las imágenes originales, las escaladas y las rotadas. Finalmente se analiza visualmente el tamaño de los pares punto de interés-descriptor y se analiza comparativamente la cantidad de puntos de interés generado por cada método.

La segunda actividad consiste en programación de la generación de matches entre pares de imágenes, usando puntos de interés DoG y descriptores SIFT. Para lo cual se implementa la detección de keypoints con DoG mediante funciones de OpenCV en 5 pares de imágenes y se programa manualmente el matching entre descriptores de cada par de imágenes usando el método de fuerza bruta. Finalmente, se palotea el par de imágenes y sus matches, además, se analiza la cantidad y calidad de las correspondencias obtenidas.

Las secciones del informe corresponden a: Un marco teórico donde se expone en qué consiste un punto de interés y los tipos existentes (blob, esquina). Se describe en qué consiste un descriptor especificando en el descriptor SIFT. Finalmente, se describe el proceso de generar correspondencias entre dos pares de matrices con descriptores de imágenes, describiendo el método de fuerza bruta (haciendo hipótesis sobre los resultados que se obtendrán).

La sección de desarrollo en general presenta las imágenes obtenidas como resultado del procesamiento, el funcionamiento general de los scripts entregados y las partes más importantes del código. La sección se separa en dos partes.

La primera parte es de análisis de invariancia en detección de puntos de interés. Parte explicando como funciona el script *main_parte1.cpp* para procesar todas las imágenes de una vez por cada detector de puntos de interés. Se explican las funciones importantes de OpenCV para extracción. Luego, se presentan las imágenes con sus

puntos de interés, se analizan sus largos y también, se analiza una tabla con la cantidad de puntos detectados por cada detector.

Del mismo modo en las secciones de escalamiento y rotación, se explica la función implementada, se analizan los largos, orientaciones y cantidad de los puntos obtenidos bajo diferentes detectores.

La segunda parte es de generación de correspondencias, es decir, matching entre descriptores. Se explica como funciona el script *main_parte2.cpp* que procesa las 10 imágenes de una sola vez, creando y guardando todas las imágenes solicitadas en la tarea. Luego, se explica brevemente como se obtienen los puntos de interés DoG y los descriptores SIFT de cada par de imágenes Train/Query. Se explica como se implemento la función *do_match()* encargada de generar las correspondencias. Se explica la función *drawMatches*, que es especial para dibujar las líneas de matches. Finalmente, se presentan los 5 pares de imágenes obtenidas y se analiza el funcionamiento de la función de matching implementada.

Por último, en la conclusión se hace una mirada más general de los resultados obtenidos y de sus análisis. Se discute hasta qué punto se observan los comportamientos teóricos esperados por los detectores de keypoints y los matches generados. Se comenta sobre las dificultades superadas para realizar esta experiencia y se finaliza con los aprendizajes y habilidades desarrolladas.

II. MARCO TEÓRICO

II-A. Puntos de interés

Los puntos de interés son discontinuidades en la imagen o en la imagen escalada. Matemáticamente son píxeles y el objetivo de buscarlos es que de alguna manera estos píxeles caractericen la imagen y que no sea necesario procesar toda la imagen para lograr tareas típicas del campo de procesamiento de imágenes como detección de instancias de objetos o alineamiento de imágenes. Por lo que la velocidad y calidad de los puntos obtenidos son lo que optimizan los detectores. Los puntos de interés se obtienen de detectores de puntos de interés, lo más importante es que la detección sea repetible, es decir, que siempre resulten los mismos puntos al procesar la misma imagen

Hay dos principales familias de detectores de puntos de interés: los detectores de esquinas y los detectores de tipo blob.

1) Punto de interés tipo blob:

2) Punto de interés tipo esquina: Los puntos de interés tipo esquina de presentan en un píxel en que se tienen variaciones o bordes en dos sentidos no paralelos que se interceptan en el punto que llamaremos esquina. Más gráficamente, en la imagen 1, nos fijaremos en la parte superior de la imagen en los puntos detectados en las esquinas superiores de la caja, encerrados en círculos lila y blanco. Se observa que si recorremos la imagen en la dirección vertical(aplicando previamente filtros pasa alto para detectar borde) hay un borde por el borde superior de la caja, al recorrerla en dirección horizontal se detectan dos bordes. Son estos puntos de intersección de los bordes en direcciones horizontales y verticales (o algún par de direcciones no paralelas) donde aparecen los puntos de interés basados en esquinas. Es por esto que son más distinguibles al ojo humano.



Figura 1. Ejemplo detección punto esquina.

II-B. Descriptores

Los descriptores son vectores de características, es decir, son arreglos que describen matemáticamente la vecindad de un punto de interés. En general, mientras más grande sea la vecindad que se considera como local más grande sera el descriptor. Descriptores más pequeños son más rápidos de computar, aunque menos robustos a transformaciones geométricas o variaciones de iluminación.

Existen descriptores que en sus componentes pueden tener números reales y enteros, que se comparan entre si, es decir, se determina matemáticamente que tan distintos son con metrica euclidiana L_2 o también con metrica L_1. También existen descriptores basados en arrays binarios, como ORB, BRIEF y BRISK; que se comparan con la métrica de Hamming[1].

Los descriptores también pueden ser multiescala, que pasarían a ser un tensor que describe la vecindad del keypoint en la escala actual, la escala superior y la inferior, como en nuestro caso del descriptor SIFT que es el único que se ocupa en esta tarea. Realmente, la definición puede abstraerse y expandirse en muchas dimensiones. Sin embargo, no

tiene mucho sentido inventar un descriptor de la nada sin basarse en los actuales; porque se cuenta con un excelente estado del arte basado en décadas de investigación, que son un excelente equilibrio entre velocidad de computo he invarianza a transformaciones geométricas.

En fin, lo esencial es que el objetivo de un descriptor es poder ser comparado con otro descriptor buscando los pares de descriptores que tengan menor distancia. Optimizando robustez ante transformaciones geométricas como escalamiento y rotación. O optimizando tiempo de computo. Es entre esos dos aspectos en que se da un trade-off por lo que existen bastantes y buenos descriptores según la aplicación que se quiera dar.

II-C. Generación de correspondencias usando descriptores

La generación de correspondencias entre descriptores es esencialmente hacer un match entre los pares de descriptores que tengan una menor distancia bajo una métrica específica. Aterrizando esto, en un ejemplo muy común de buscar una imagen (query) dentro de otra(train). Dada la representación de los descriptores como una matriz, primero se parte tomando la primera columna de la matriz de descriptores de la imagen de query, luego se toma la primera fila de matriz de descriptores de la imagen de train, se les calcula la distancia euclíadiana, la cual siempre tienen sentido porque ambos descriptores son del mismo tipo, es decir, los vectores que representan al descriptor tiene la misma dimensión.

Luego, se toma el siguiente descriptor de la imagen de train y se calcula la distancia euclíadiana. Una vez calculadas todas las distancias euclidianas entre el descriptor proveniente de la imagen de query y todos los de la imagen de train. Se toma el índice del descriptor de train con que se tenga la menor distancia. Finalmente se hace un match entre el descriptor de índice que se parte tomando de la imagen de query con el índice del descriptor que daba la menor distancia de la imagen de train.

Normalmente, la métrica, es decir la formula que se usa para medir la distancia entre descriptores, es la L2 o euclíadiana. Sin embargo, también se puede usar la métrica L1, dichas normas son apropiadas para los descriptores SIFT y SURF. Por otro lado, para descriptores que se expresan como vectores en que sus elementos son binarios como los descriptores ORB, BRIEF y BRISK deben usar distancia de Hamming.

Explicando concisamente el punto anterior, la distancia de Hamming se calcula comparando elemento a elemento de un par de vectores. Es 0 cuando dos componentes son iguales y 1 cuando son diferentes. Luego, se suman dichas distancias entre componentes. Entonces, dos descriptores hacen match cuando tienen la más baja distancia de Hamming.

En suma, ya sea utilizando métricas continuas como L2 o discretas como la distancia de Hamming. Lo más importante es comprar descriptores obtenidos con las mismas métricas y hacer el match entre los pares de descriptores que tengan las distancias mínimas.

III. DESARROLLO

III-A. Invarianza de puntos de interés

1) **Detección Keypoints imagen original:** Primero que todo se deba aclarar que el código *main_parte1.cpp* funciona de la siguiente manera. Para cada una de las tres imágenes, el código calcula y guarda la imagen escalada, rotada y los puntos de interés dibujados en ellas. Ahora bien, esto lo hace para un descriptor, entonces, para obtener todas las imágenes se debe ir comentando y descomentando las líneas del siguiente bloque para ir cambiando de tipo de descriptor. Entonces, en total se debe compilar y correr 3 veces el código.

```

1 Ptr<FeatureDetector> featureDetector = ORB::create();
2 string name_keyPointsDetector = "ORB";
3
4 //Ptr<FeatureDetector> featureDetector = xfeatures2d::SIFT::create();
5 //string name_keyPointsDetector = "SIFT";
6
7 //Ptr<FeatureDetector> featureDetector = GFTTDetector::create();
8 //string name_keyPointsDetector = "GFTT";

```

El procesamiento de las tres imágenes cada vez que se corre el código se implementa con un ciclo for de 3 iteraciones; cada iteración procesa una imagen. La dirección de cada imagen se le entrega creando un array estático con los nombres de las imágenes. La lectura se hace con la función *imread()* la cual recibe un string con la dirección de la imagen, lo importante de esto es mencionar que se investigó y aprendió que en cpp se pueden concatenar strings fácilmente usando "+", por ejemplo, en el código de a continuación, en la primera iteración del for se ejecutaría: *imread("casa.png");*

```

1 string img_names[] = {"casa","dibujo","figurasllenas"};
2 for (int s=0;s<3;s++){
3     input = imread(img_names[s]+".png");
4 }
5 ...

```

Una vez explicado el comportamiento general del código, se explica la detección de los puntos de interés. Las funciones más importantes están en el siguiente extracto de código;

Una vez cargada la imagen, es necesario iniciar un generador de números pseudo-aleatorios que será usado por los detectores, *time(NULL)* significa que se inicializa con un valor devuelto por una función en dependiente de la hora, lo importante, es que asegura obtener números los suficientemente pseudoaleatorios (línea 1).

Luego, se crea un objeto de la clase FeatureDetector llamado *featureDetector* que es el objeto encargado de analizar la imagen y detectar los puntos de interés, se especifica que son puntos de interés ORB con la parte *ORB::create()*. Si se desean puntos de interés DoG (en SIFT) se usa *xfeatures2d::SIFT::create()* y para keypoints Good Features Features To Track se usa *GFTTDetector::create()*.

Se crea un arreglo dinámico de objetos KeyPoint llamado *keypoints1* cuya función será guardar los

keypoints detectados por el objeto *featureDetector* (línea 3). Seguidamente, se detectan los puntos de interés usando el método *detect* del objeto *featureDetector* (línea 4), entregándole como entrada la imagen a analizar y el arreglo de Keypoints en el que guarda los puntos de interés que detecte.

Finalmente, se dibujan los puntos de interés con la función *drawKeypoints*(línea 5). Dicha función recibe la imagen original el arreglo con los keypoints, una referencia a una imagen de output que sera la imagen donde estén dibujados los keypoints. Además recibe flags, las flags son parámetros que se usan para agregar pequeñas funcionalidades a la función principal. Por ejemplo, definir con que colores se dibujarán los keypoints o en este caso, la flag: *DrawMatchesFlags::DRAW_RICH_KEYPOINTS* dibuja un círculo alrededor del punto de interés con el largo y la orientación del punto de interés.

```

1 srand(time(NULL)); //Inicia generador de numeros al azar
2 Ptr<FeatureDetector> featureDetector = ORB::create();
3 vector<KeyPoint> keypoints1;
4 featureDetector->detect(input, keypoints1);
5 drawKeypoints(input, keypoints1, output_original,
6 DrawMatchesFlags::DRAW_RICH_KEYPOINTS);

```

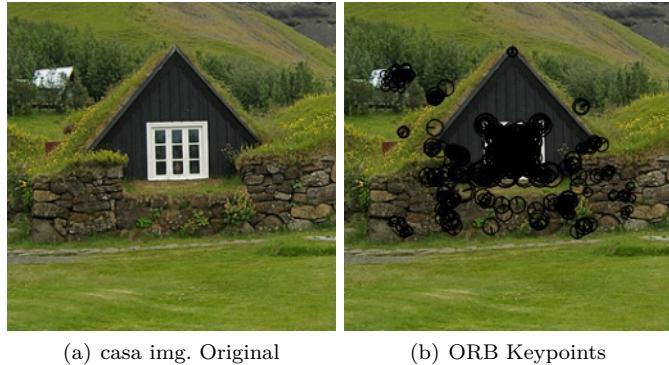
La figura 2 muestra los puntos de interés dibujados con los tres métodos diferentes para la imagen de casa. La subimagen a) muestra los puntos de interés ORB detectados en ella se observa que se detectan los puntos de interés tipo esquina en la punta del techo más cercano y el más lejano, también en las esquinas de la ventana y en algunas esquinas del muro de ladrillo lo cual esta bien porque no se detectan como puntos de interés los bordes de los techos ni los bordes de la ventana ni los bordes de los ladrillos. Esto se debe a que el detector ORB detecta punto tipo corner.

Además, en las imágenes originales procesadas con DoG y GFTT se observan muchos más keypoints y en lugares que no se perciben como esquinas, esto comprueba la teórica porque dichos detectores estan basados en keypoints tipo blob.

Img Original	ORB	SIFT	GFTT
casa	438	478	1000
dibujo	500	190	372
figurallena	194	42	5

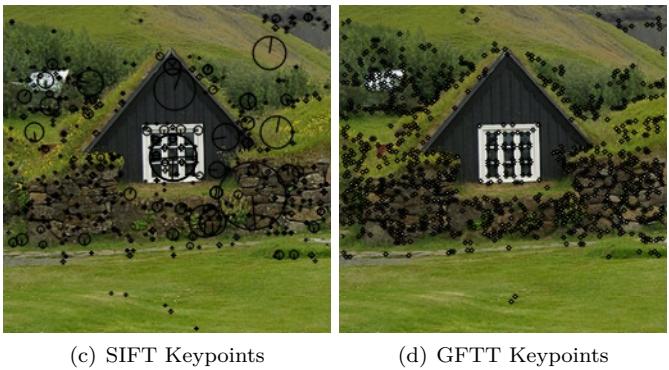
Tabla I. CANTIDAD PUNTOS DE INTERÉS DETECTADOS EN IMÁGENES SIN APLICAR TRANSFORMACIÓN GEOMÉTRICA

Un punto importante que se observara en todas las imágenes presentadas es el aspecto en los tipos de puntos de interés en que se basan los detectores. ORB se basa en puntos de interés tipo corner o esquina mientras que DoG y GFTT se basan en puntos de interés tipo blob o regiones centro-contorno. Entonces, en las imágenes con ORB se observara claramente keypoints en esquinas fácilmente distinguible para el cerebro humano. En cambio, en las imágenes procesadas con DoG y GFTT los keypoints son máximos o mínimos en relación a sus vecinos al aplicar una transformación matemática a todos los píxeles de la imagen. Luego, NO son trivialmente distinguibles para el ojo humano y se presentaran en zonas que no necesariamente se unifican como esquinas pero que son relevantes por la información que las distingue con respecto a su vecindad local.



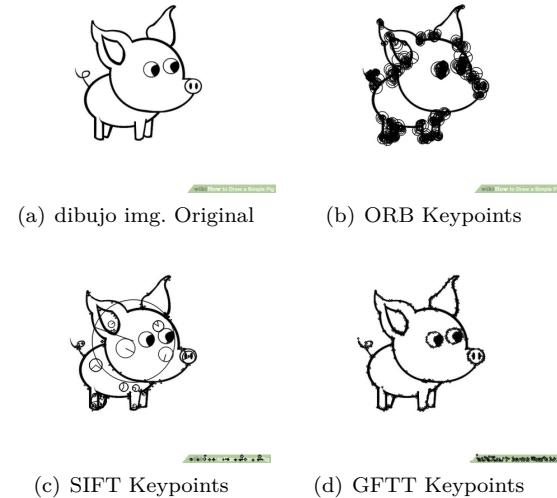
(a) casa img. Original

(b) ORB Keypoints



(c) SIFT Keypoints

(d) GFTT Keypoints

Figura 2. Visualización puntos de interés detectados en imagen *casa.png* con diferentes métodos.Figura 3. Visualización puntos de interés detectados en imagen *dibujo.png* con diferentes métodos.

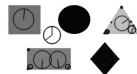
2) Análisis de cantidad de keypoints generados sin escalamiento:

3) Programación función de escalamiento *do_downsample*: Esta función se implementa con la función nativa de OpenCV llamada *pyrDown()*. La función *pyrDown()* recibe la imagen en escala original,



(a) figuras llenas img. Original

(b) ORB Keypoints



(c) SIFT Keypoints

(d) GFTT Keypoints



(d) GFTT Keypoints

Figura 4. Visualización puntos de interés detectados en imagen *figuras llenas.png* con diferentes métodos.

una referencia a la imagen escalada que generará y un objeto *Size* con las nuevas dimensiones, como por ejemplo: *Size(tmp.cols/2, tmp.rows/2)* en este se ingresan primero las columnas resultantes y luego las filas de la imagen resultante.

```

1 Mat do_downsample(Mat input){
2     Mat output = input.clone();
3     pyrDown(input, output);
4     return output;
5 }
```

Por defecto, la función *pyrDown()* escala las imágenes a la mitad de su tamaño original. Por último y no menos importante, se menciona que *pyrDown()* es equivalente a convolucionar por la máscara Gaussiana de la figura 5 y luego subsamplear como se hizo en la tarea pasada.

$$\frac{1}{16} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

Figura 5. Máscara con que *pyrDown* *pyrDown()* convoluciona.

4) Detección Keypoints imagen escalada: En general se observa que se generan menos puntos de interés. Recordando que parte del escalamiento es una convolución que modifica los valores de los píxeles. Analizando visualmente los puntos ORB se siguen detectando en las esquinas destables a simple vista. Los puntos SIFT se detectan en algunas zonas planas blancas (por estar basado parcialmente en blob) y los GFTT se detectan solo en esquinas (como en la imagen *figuras llenas*) aunque es teóricamente también pudieron haber aparecido en zonas planas blancas por estar totalmente basados en keypoints tipo blob.

5) Análisis de cantidad de keypoints generados en escalamiento: Lo más importante de entender de la invarianza al escalamiento es los puntos de interés de la imagen escalada tienen que estar en la imagen sin escalar. Lo cual se observa

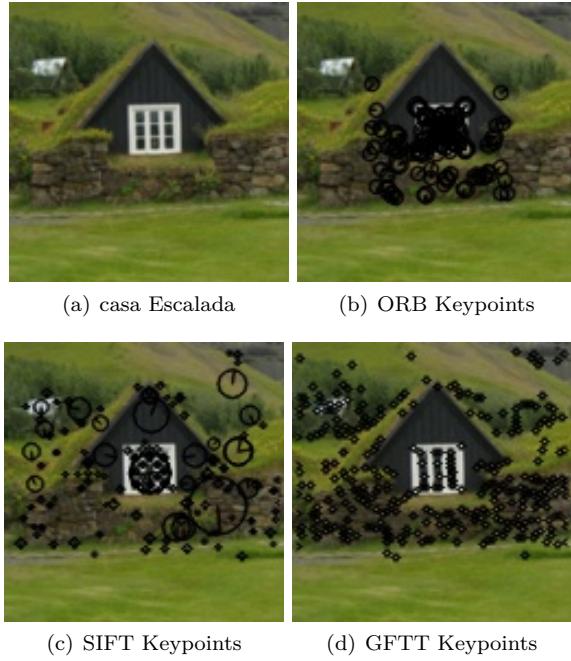


Figura 6. Visualización puntos de interés detectados en imagen *casa.png* escalada con diferentes métodos.

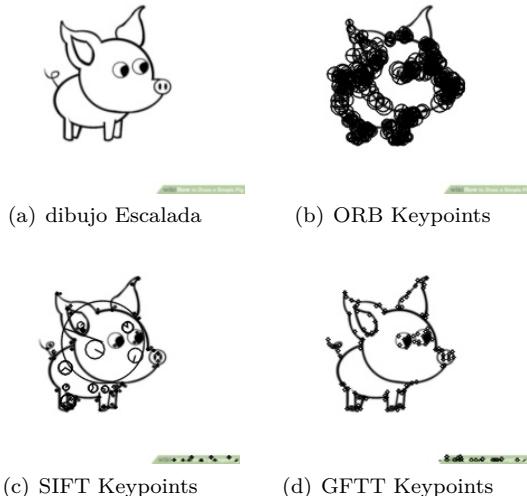


Figura 7. Visualización puntos de interés detectados en imagen *dibujo.png* escalada con diferentes métodos.

en las figuras 6, 7 y 8. Lo teóricamente esperado es que se detecten la misma cantidad de Keypoints para ORB y DoG. Lo cual no resulta así, disminuyendo las cantidades aproximadamente a la mitad, aunque teóricamente, se sabe que ORB sacrifica robustez en detección en escalamiento por velocidad de computo. En el caso de GFTT teóricamente que no es invariante a la escala por lo que se comprueba esta varianza. En conclusión se observa que los detectores ORB y DoG tienen un bajo nivel de invarianza

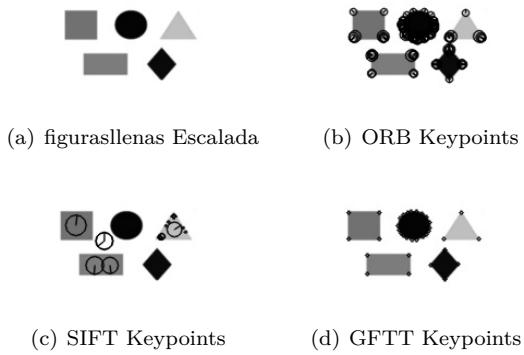


Figura 8. Visualización puntos de interés detectados en imagen *figurasllenas.png* escalada con diferentes métodos.

al menos para estas imágenes

Img Escalada	ORB	SIFT	GFTT
casa	112	151	319
dibujo	409	118	128
figurallena	79	16	34

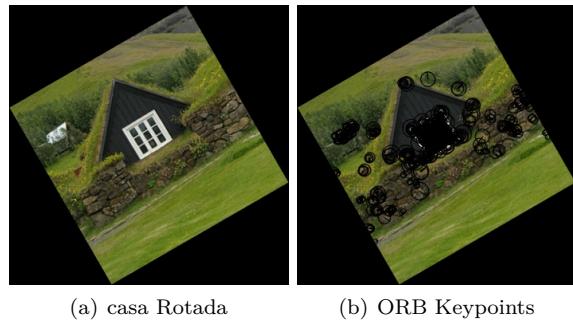
6) **Detección Keypoints imagen rotada:** A continuación se presentan los resultados de detección de keypoints sobre las imágenes rotadas. Analizando visualmente, se observa que no se mantiene la orientación ni el tamaño de los keypoints detectados. Se observa que la cantidad varía muy poco. Se observa que los puntos ORB basados en corner se mantienen en posiciones distinguibles para el ojo humano. Mientras que los puntos DoG y GFTT, basados en blob (DoG parcialmente basado) se siguen detectando en posiciones no trivialmente observables y que algunos cambian de posición con respecto a la imagen original y otros se quedan en el mismo píxel.

Img Rotada	ORB	SIFT	GFTT
casa	493	571	1000
dibujo	500	208	348
figurallena	258	50	75

7) **Análisis de cantidad de keypoints generados en rotación:** Teóricamente ORB, DoG y GFTT son invariantes a la escala. Se calculó la diferencia porcentual entre la cantidad de keypoint detectado antes y después de la rotación, se observa que la mayor variación fue un aumento en un 36 % en la imagen figuras llenas con el detector GFTT. Todas las demás variaciones para todos los detectores e imágenes fueron menores. Con lo anterior, se concluye que se comprobó bajo un cierto rango de error que los detectores con que se trabaja son invariantes a la rotación.

III-B. Generación de correspondencias

1) **Detección y gráfico de Keypoints de pares de imágenes:** Esta actividad está casi hecha en el código entregado. Lo nuevo que se implementó es que sólo se compile y corra una vez el código, el que de una sola vez procesa los 5



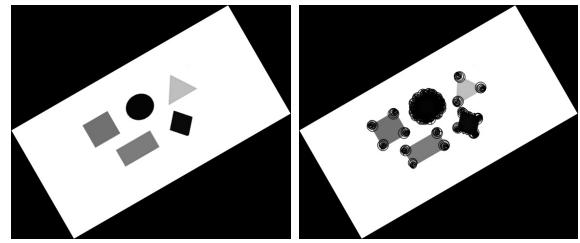
(a) casa Rotada

(b) ORB Keypoints



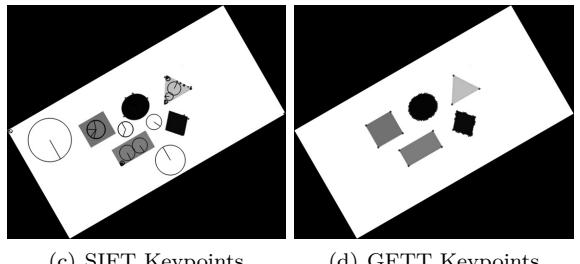
(c) SIFT Keypoints

(d) GFTT Keypoints

Figura 9. Visualización puntos de interés detectados en imagen *casa.png* rotada con diferentes métodos.

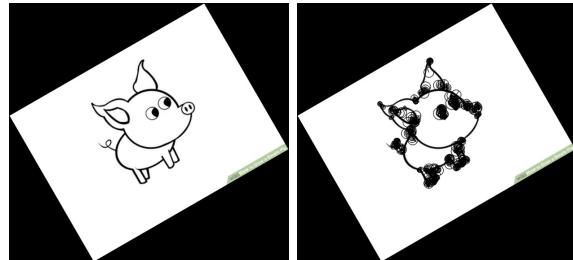
(a) figurasllenas Rotada

(b) ORB Keypoints



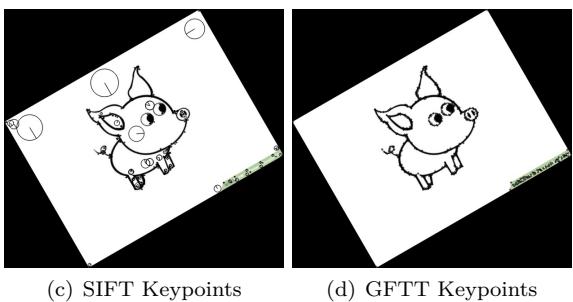
(c) SIFT Keypoints

(d) GFTT Keypoints

Figura 11. Visualización puntos de interés detectados en imagen *figurasllenas.png* rotada con diferentes métodos.

(a) dibujo Rotada

(b) ORB Keypoints



(c) SIFT Keypoints

(d) GFTT Keypoints

Figura 10. Visualización puntos de interés detectados en imagen *dibujo.png* rotada con diferentes métodos.

pares de imágenes guardando los gráficos de keypoints y las imágenes de matches. Esto se implementa con un ciclo *for* de 4 iteraciones. Primero se declara un array estático con los nombres de los archivos de las imágenes. Luego, en cada ciclo del *for* se carga un par de imágenes al variar el string de entrada de la función *imread* al ir concatenando

los nombres de los arrays creados usando el signo +. Esto es importante y útil porque permite procesar todas las imágenes de una carpeta y en general hace más cómodo el trabajo.

```

1  string train_img_names[]={"dentifrice","ice1",...,"uch098a"};
2  string query_img_names[]={"dentifrice2","ice2",...,"uch098b"};
3  for (int i=0; i<5 ;i++){
4      input1 = imread("im/" +train_img_names[i]+".jpg"); // TRAIN image
5      input2 = imread("im/" +query_img_names[i]+".jpg"); // QUERY image
6  }

```

El ejemplo más usado para explicar matching es el de buscar una instancia de un objeto dentro de una imagen de escena con más objetos de más clases. A la imagen con más objetos de más clases en la documentación se le llama **TRAIN** en nuestro código son las cargamos como *input 1*. La imagen de la instancia de un objeto, en los ejemplos de las documentaciones, se le llama **QUERY** y la cargamos como *input 2*.

Entonces, como ya se explicó los que eran los keypoints en la parte 1, aquí se explica que en el código implementado el *array Keypoints1* y la *Mat Descriptors1* son obtenidos de la imagen de **TRAIN**. Análogamente, el *array Keypoints2* y la *Mat Descriptors2* son obtenidos de la imagen **QUERY**.

```

1  vector<KeyPoint> keypoints1;//keypoints de TRAIN
2  vector<KeyPoint> keypoints2;//keypoints de QUERY
3  Mat descriptors1;//Descriptores de Keypoints de TRAIN
4  Mat descriptors2;//Descriptores de Keypoints de QUERY

```

La figura 12 presenta los keypoints detectados para el par de imágenes de *dentifrice*. Se observa que se detectan satisfactoriamente los puntos de interés principalmente en las esquinas de letras, lo cual es lo esperado teóricamente. Se producen algunas detecciones en sectores planos uniculares que se explican porque en esa vecindad local

el algoritmo los detecta como esquinas los peaks blancos debido al reflejo por la iluminación del flash de la cámara.



Figura 12. Visualización puntos de interés detectados en las imágenes "dentrífice" usando DoG detector.

En la figura 13 se grafican los keypoints de las imágenes ice. Se obtienen excelentes resultados detectando las esquinas de las letras, de los números y de la caja. Además, se detectan los patrones del piso, que también son esquinas (no de nuestro interés pero el algoritmo no tiene esa información a priori).

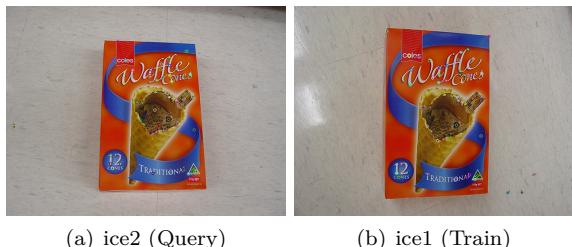


Figura 13. Visualización puntos de interés detectados en las imágenes "ice" usando DoG detector.

La figura 14 son los keypoints del par uch006. Se obtienen detecciones en las esquinas de letras y números del cartel, y en los enchufes de la pared, lo que es esperado. Esta excelentemente bien que no se hayan detecciones en la pared plana ni en las áreas negras planas del cartel.

Las imágenes de 15 son fotos de páginas amarillas por lo que presentan alto contenido en frecuencias altas. Las detecciones son en esquinas de números y letras de acuerdo a los esperado.

Finalmente, en la figura 16 las detecciones son también en números y letras, además de esquinas de las figuras del afiches. En la imagen (b) Train hay detecciones en la parte inferior derecha por peaks de iluminación externa.

2) Programación función *do_match*: La función *do_match* es la encargada de recibir una matriz con los



Figura 14. Visualización puntos de interés detectados en las imágenes "uch006" usando DoG detector.



Figura 15. Visualización puntos de interés detectados en las imágenes "uch084" usando DoG detector.

descriptores de la imagen query, la otra matriz con los descriptores de train y un array dinámico de objetos Dmatch vacío. En su ejecución rellena el array de Dmatch con los pares de descriptores que tengan mínima distancia. En esta versión no retorna nada, sólo modifica el array de Dmatch.

El funcionamiento interno es partir iterando sobre los descriptores de la imagen query con el primer for sobre i. Para cada descriptor de query crea un objeto Dmatch llamado match. Los objetos Dmatch tienen 4 parámetros:

```
match.imgIdx = El índice de la imagen de train, generalmente es 0.
match.queryIdx = El índice del descriptor de query (la fila en la Mat query).
match.trainIdx = El índice del descriptor de train (la fila en la Mat train).
match.distance = La distancia entre los descriptores.
```

Entonces, ahora sé esté llenando los parámetros del objeto match. Se declara siempre que la imagen de train es cero (es diferente cuando hay varias imágenes de train)(línea 4). Se declara que el índice del descriptor de query es i (línea 5). En total, se llenarán 100 objetos Dmatch (porque se detectan 100 keypoints) cada uno irá aumentando i en +1, partiendo de 0. Además, se crea una variable auxiliar llamada *distancia_minima* para guardar la distancia entre el par de descriptores que minimicen su distancia euclidiana; la idea es que aparte con un valor alto para que la primera distancia calculada con la que se comparará se guarde como la distancia mínima.

```
void do_match(Mat queryDp, Mat trainDp, vector<DMatch>& matches){
    for (int i = 0; i < queryDescriptors.rows; i++){
        DMatch match;
        match.imgIdx = 0;
        match.queryIdx = i;
        float distanciaMinima = float(1000000.0);
        ...
    }
}
```



(a) uch098b (Query)

(b) uch098a (Train)

Figura 16. Visualización puntos de interés detectados en las imágenes "uch098" usando DoG detector.

Luego, se empieza a recorrer los descriptores de la imagen train con un ciclo for sobre *j*. *j* ira tomando el valor del índice de los descriptores dentro de la matriz de la imagen train. Se crea una variable auxiliar llamada *distancia_entre_descriptoresij* para guardar la distancia entre los dos descriptores que estamos analizando. Luego, se itera sobre cada componente de estos dos descriptores con otro ciclo for sobre *k*, que llega hasta 127 porque los descriptores SIFT tienen largo 128. Entonces, la parte importante es el calculo de la norma euclidiana; para esto, se toma la componente del descriptor de query y se resta con la componente del descriptor de train, el resultado se guarda en una variable auxiliar llamada *d*. Se eleva *d*, es decir, la diferencia entre componente, al cuadrado y se suma al valor que tenia del *d* en la iteración anterior de *k*. Una vez recorridos las 128 ciclos, la distancia acumulada se divide por 128.0 y se le calcula la raíz cuadrada.

```

1  for (int j = 0; j < trainDescriptors.rows; j++){
2      float distancia_entre_descriptoresij = float(0.0);
3      for (int k=0, k<128; k++){
4          float d;
5          d = queryDescriptors.at<float>(i, k) -
6              trainDescriptors.at<float>(j, k);
7          distancia_entre_descriptoresij += d*d;
8      }
9      distancia_entre_descriptoresij =
10     sqrt(distancia_entre_descriptoresij/128.0);
11 }
12 }
```

Ahora bien, el objetivo es encontrar la menor de las distancias. Entonces, con un bloque if se pregunta si la distancia calculada es menor a la que se tenia guardada (para cada nuevo *i* (query) la primera distancia calculada siempre es la menor). Si lo es la distancia mínima se actualiza a la distancia calculada. En el objeto Dmatch, el índice del descriptor de train pasa a ser el valor de *j* con *match.trainIdx = j* y la distancia *match.distance* a guardar pasa a ser la distancia mínima calculada.

Si no se cumple la condición anterior, se pasa al siguiente *j* (siguiente descriptor de train), hasta terminar de recorrer la matriz de descriptores train. Finalmente y una ves construido el objeto Dmatch, este se agrega al final del array dinámico con *matches.push_back(match)*.

```

1  for (int i = 0; i < queryDescriptors.rows; i++){
2      for (int j = 0; j < trainDescriptors.rows; j++){
3          if (distancia_entre_descriptoresij < distancia_minima){
4              distancia_minima = distancia_entre_descriptoresij;
5              match.trainIdx = j;
6              match.distance = distancia_minima;
7          }
8      }
9      matches.push_back(match);
10 }
```

```

4
5      if (distancia_entre_descriptoresij < distancia_minima){
6          distancia_minima = distancia_entre_descriptoresij;
7          match.trainIdx = j;
8          match.distance = distancia_minima;
9      }
10     matches.push_back(match);
11 }
```

En suma, como estamos buscando el objeto query (una instancia de un objeto) en el objeto train (conjunto de muchas instancias de muchos objetos) tiene todo el sentido partir iterando sobre query porque a priori se esta buscando algo (query) que se sabe que está en la imagen en que sé esté buscando (Train). Si se hiciese al revés, a priori sabemos que mas de las instancias de objetos que están en train no tienen porque estar en la imagen de query, entonces, se aumentara la posibilidad de cometer errores en el matching.

3) Calculo y grafico de matching entre pares de imágenes: EL calculo de matches se realiza invocando a la función *do_match* que ya se explicó como los obtuvo. Los gráficos se hacen con la función *drawMatches* en la cual, los primeros dos argumentos son la imagen de query y sus keypoints (se dibujaran a la izquierda de la imagen final). Luego, se ingresaN la imagen de train y sus keypoints (que se dibujaran a la derecha). Finalmente, el array de objetos Dmatch (en este código se llama *matches*) y la imagen final que se creara *img_matches*.

```
drawMatches(input2, keypoints2, input1, keypoints1,
           matches, img_matches);
```

Es importante colocar los argumentos en orden correcto porque si no al intentar trazar matches (dibujar líneas) habrán keypoints que no tienen par (siendo que el array matches dice que tienen) y el script lanzara error.

4) Análisis visual calidad de matches: En general, es importante mencionar que como la función *do_match* recorre primero los descriptores de la imagen que se presenta a la izquierda. En todas las siguientes imágenes, todos los puntos de interés de la imagen de la izquierda siempre tendrán un match en la de la derecha, no así para los de la imagen de la derecha, porque se puede dar que uno de sus descriptores tenga distancia grande con los descriptores de la izquierda, entonces, nunca tendrá un match y se visualizará solo un pequeño círculo.

La figura 17 muestra los matches de las imágenes dentrífice. Se observan matches correctos en la zona que dice "*LICOR POLO*". Teóricamente, se esperaban matches correctos en la zona que dice "+33% gratis" pero no se obtuvieron. Se observan muchos matches incorrectos en la donde dice *JUNIOR* y en +6 AÑOS. Son esperables estos mis-matches porque realmente la imagen query no es una instancia que aparezca en la imagen train. Los mis-matches son coincidencias porque son esquinas de caracteres de la misma fuente. Estos mis-matches se eliminan, valga la redundancia, implementando un filtro de mis-matches.

La figura 18 es el matching de la caja de conos de helado. Se espera que se visualicen muchos matches correctos



Figura 17. Matches entre imágenes "dentifrice".

porque simplemente se hace un leve cambio de escala y rotación siendo SIFT en teoría invariante a estos. Se observa que se obtienen bastantes matches correctos en el área que dice *waffle cones*, la parte de *tradicional*, en el círculo de *12 cones* y el triángulo inferior derecho. También, se observan mis-matches entre las esquinas del cono de helado y la textura del piso, que pueden ser eliminados con un filtro.

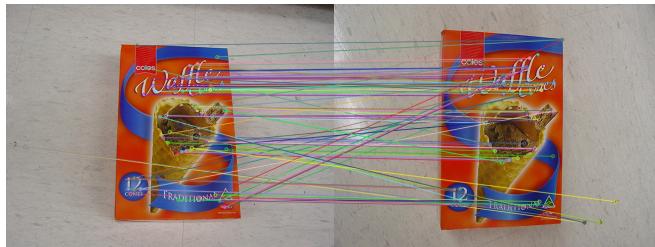


Figura 18. Matches entre imágenes "ice".

En la figura 19 se muestra el matching de un afiche de la *RoboCup*. Se observan buenos matches en la zona que dice textit"RoboCup 2005 OSAKA" y letras japonesas, esto esta correcto de acuerdo a la teoría. Lo malo es que se observan demasiados matches incorrectos entre un punto de interés de la imagen de train (ubicado arriba y a la izquierda) y varios keypoints del afiche (en la imagen de query) y de los enchufes de la pared. Esto se debió a que en particular justo ese descriptor local (de train) con la mayoría de los descriptores de query daba una distancia baja en comparación con las otras distancias con los keypoints de train.

Del mismo modo que se explico más atrás al definir la función *do_match* de acuerdo a la teoría es totalmente esperable que aumente la cantidad de mis-matches. Porque lo que se esta haciendo es invertir el rol de la imágenes. Me explico la imagen del afiche en la pared debería ser de train porque contiene varias instancias de objetos (dos

afiches distintos, las persianas y los enchufes) y la imagen de query debería ser la que tiene el afiche de protagonista.

Si fuese así, la imagen de train contendría descriptores de varias instancias de objetos y la de query solo descriptores de una instancia de objeto en particular(el afiche de la robocup). Y al hacer el matching, de obtendrían menos errores. Entonces, en nuestro caso como es al revés, siempre pueden darse coincidencias locales entre descriptores de otras instancias y algunos descriptores de la instancia de objeto en particular, lo que provoca que aumente la probabilidad de error y son los que se observan en la imagen presentada.

En fin, esto se soluciona con un filtro de mismatches de crossCheck que, en conciso, lo que hace es partir recorriendo un descriptor de la imagen de train y va calculando las distancias con los de query y genera *pre-matches*. Luego, los matches que son agregados al array *Dmatch* con los que se encuentran con ambas formas de recorrer las matrices. En suma soló sobreviven los matches que constituyen una suerte de biyección entre matrices.

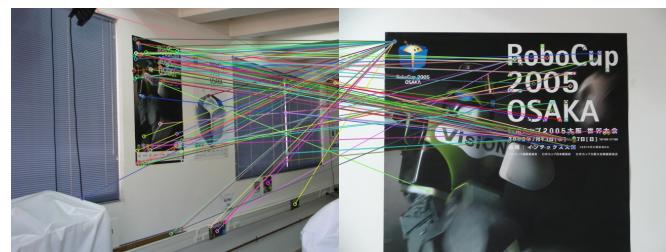


Figura 19. Matches entre imágenes "uch006".

En la imagen 20 se observan los matches de paginas amarillas, aquí hay rotación y cambio de escala, además son imágenes bastante diferentes que tienen muchas instancias (letras y figuras) diferentes entre ellas. Entonces, lo esperado es que haya muchos mis-matches. Se observan algunos matches correctos en zonas centrales donde dice *QUAT-PRO*. Se observan muchos mismatches entre "minavisos" diferentes en que se da que localmente coinciden las esquinas pero no coinciden en significado global, porque son instancias distintas. Realmente, un filtro eliminaría la mayoría de los matches simplemente porque entre imágenes se agregan instancias nuevas que nunca coincidirán.



Figura 20. Matches entre imágenes "uch084".

Finalmente, en la imagen 21 se observan muchos mat-

ches correctos en las esquinas de las figuras y en las esquinas de las letras. Aquí también se da el problema de invertir los roles de train/query por lo que se observan mis-matches entre los pliegues luminosos de la bolsa de la imagen izquierda con puntos internos del afiche. Estos se solucionan invirtiendo el rol de las imágenes (invirtiendo el orden en que se llama la función), y también, implementando el filtro de mis-matches.



Figura 21. Matches entre imágenes "uch098".

IV. CONCLUSIÓN

Finalizada la experiencia se logran verificar que algunas las hipótesis estaban de acuerdo a la teoría con respecto invariabilidad (o variabilidad) al escalamiento y rotación de los detectores de puntos de interés ORB, DoG y GFTT; al analizar la cantidad de pares keypoint-Descriptor generados por dichos métodos. Se logra implementar manualmente el matching entre los descriptores SIFT de 5 pares de imágenes. Se logra explicar que el método implementado genera matches falsos porque no se implementa un método de filtrado.

En particular en la primera parte de la experiencia se logra programar las funciones *do_downsample* y *do_rotate*. Las imágenes se procesan con dichas funciones y se presentan con los puntos de interés dibujados. Al observar visualmente y comparar cantidad de keypoints detectados (información que se presentó en tablas), se comprueba que DoG y ORB son invariantes al escalamiento no tan robustamente; y que GFTT es variante a la escala. Lo importante es concluir que no se puede afirmar categóricamente que los detectores están mal implementados en OpenCV porque en un par de imágenes particular no se observe la invarianza. Con respecto a la rotación, se comprueba que todos los detectores son invariante a la rotación con un cierto rango de error.

Con respecto a la segunda parte se logra detectar los keypoints DoG usando funciones predefinidas, se comprueba que se obtienen puntos de interés en puntos esquinas. Se logra implementar el matching por fuerza bruta y se plotean los 5 pares de imagen. Se comprueba que teóricamente deberían haber keypoints en la imagen de train sin matches, los cuales se logra visualizar. También, se comprueba que existen mis-matches debido

a que no se implementa un filtro en los matches, a diferencia de las funciones predefinidas de OpenCV, como por ejemplo un filtro de crossCheck.

En esta experiencia se profundizó la programación en C++ y el trabajo con la distribución de Linux Ubuntu. Lo más importante es que se aprendió a trabajar con objetos para labores específicas para estas técnicas como *Ptr*, *KeyPoint*, *DMatch*. También, se aumentan los conocimientos en temas operativos muy útiles como concatenar strings, guardar imágenes e importar librerías desde archivos. Finalmente, se aprende a escribir scripts de rápida ejecución en la maquina por ser de c++ y que procesan todas las imágenes de una carpeta fuente sin necesidad de compilar/correr el script varias veces.

Las dificultades que se encontraron durante el desarrollo de la experiencia fue que se debía aprender C++ que es un lenguaje más alejado del usuario, es decir, los códigos son menos intuitivos. La programación y el debug se hacen más complejos porque se compila y debugea desde la consola con lo que fue difícil identificar los errores. También, se presentaron dificultades con invocar librerías desde archivos. Finalmente, las dificultades se solucionaron preguntando a los compañeros de clase, investigando en Stackoverflow y en la documentación de OpenCV.

REFERENCIAS

- [1] OpenCV Documentation Descriptors Matching, disponible: docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_matcher/py_matcher.html, versi