Universidad Zaragoza
1542

Trabajo Fin de Máster

Sistema de geolocalización basado en imágenes para dispositivos móviles

Image-based geolocation system for mobile devices

Autor

Mario Subías Pérez

Director

Carlos Orrite Uruñuela

Escuela de Ingeniería y Arquitectura de la Universidad de Zaragoza

2016

Gracias a mi tutor Carlos,
por su tiempo, dedicación y ayuda,
sin él no hubiera sido posible.

Gracias a los compañeros
del grupo de investigación CV_Lab,
a Miguel Ángel, Eduardo y Mario
por la ayuda ofrecida.

Gracias a mis compañeros,
y amigos, los 'esmochaos',
por apoyarme durante estos años,
ha sido un gustazo encontraos.

Gracias a mis amigos de siempre,
por seguir estando ahí a pesar
del paso del tiempo como
si nada hubiera cambiado.

Y gracias a mis padres y hermano,
por aguantar lo bueno y lo malo
y apoyarme en todo momento.

Sistema de Geolocalización basado en imágenes para dispositivos móviles

En este proyecto fin de máster, se muestra una aplicación de realidad aumentada capaz de geolocalizar a un usuario en un entorno conocido. El sistema ha sido entrenado para funcionar en una localización real, la plaza San Felipe de Zaragoza. Al tomar una foto de la plaza, en función de los edificios que contenga la imagen, el sistema es capaz de determinar la posición desde la cual se ha tomado. Una vez realizada esta ubicación tridimensional, se superpone en la fotografía tomada una imagen 3D de la 'Torre Nueva', una antigua torre mudéjar que se encontraba en esa misma plaza hasta 1892 que fue derruida. Esta aplicación funciona de forma externa, enviando la imagen tomada por el terminal a un servidor remoto que realiza los cálculos. Todo este proceso resulta costoso en tiempo, lo que provoca que la aplicación no se pueda ejecutar en tiempo real. Tanto el tiempo de envío al servidor como la extracción de características de las imágenes en el proyecto previo requieren de un tiempo superior al deseado en una aplicación de tiempo real.

En este proyecto, se pretende implementar todas las operaciones del cálculo de la localización en el mismo terminal en el que se realiza la fotografía. Además, se muestra un estudio de técnicas de extracción de características para mejorar este tiempo de cómputo. Estas características serán los *keypoints* o puntos relevantes de la imagen. Estos *keypoints* se extraen mediante algoritmos de visión por computador llamados descriptores. En el proyecto previo se utiliza el descriptor SIFT que, como ya se ha mencionado, resulta costoso computacionalmente. En este proyecto el descriptor SIFT es sustituido por el descriptor BRISK, mucho más veloz, aunque menos preciso en su cometido.

Una vez se han obtenido los puntos relevantes de dos imágenes distintas, realiza un emparejamiento entre ellos con un algoritmo de *matching*. Es de esta forma en la cual el sistema se localiza en el entorno 3D. Los algoritmos de *matching* emparejan los *keypoints* más probables de ambas imágenes. Sin embargo, este proceso suele presentar falsos emparejamientos o espurios que deben ser eliminados. En este proyecto se presentan nuevas técnicas de realizar este filtrado para asegurar que los emparejamientos producidos sean robustos y coherentes entre sí.

En una base de datos de imágenes de la plaza San Felipe, se pueden emparejar las imágenes entre sí siguiendo el proceso anterior para obtener un modelado 3D del entorno. Con este modelado del mundo 3D, el sistema es capaz de emparejar una fotografía nueva y localizar la posición de la cámara para superponer la 'Torre Nueva' en la posición correcta.

1. Introducción ... 1
1.1 Antecedentes ... 1
1.2 Objetivos ... 2
1.3 Organización de la memoria ... 3
2. Emparejamiento de imágenes basado en descriptores ... 4
2.1 Funcionamiento del código previo ... 4
2.2 Mejoras propuestas ... 6
2.2.1 Tipo de descriptor (BRISK) ... 6
2.2.2 MatchFeatures ... 7
2.2.3 Filtrado por ángulos y escala ... 7
2.2.4 Filtrado por triángulos de Delaunay ... 9
2.2.5 Recuperación de puntos ... 11
2.2.6 Ajuste de parámetros ... 12
3. Resultados de Matlab ... 13
3.1 Parámetros a modificar ... 13
3.2 Bases de datos ... 14
3.3 Comparación de algoritmos ... 15
3.4 Resultados obtenidos ... 17
4. Visual Studio ... 25
4.1 Cambios introducidos ... 25
4.2 Resultados obtenidos ... 28
5. Emparejamiento con el modelo del mundo ... 30
6. Conclusiones ... 35
Bibliografía ... 36
Anexo A ... 37
Anexo B ... 39

1. Introducción

Este proyecto fin de Master ha sido realizado en el grupo de investigación del laboratorio de visión por computador, CV_LAB, perteneciente al Instituto Universitario de Investigación en Ingeniería de Aragón (I3A). Dicho grupo lleva trabajando en los últimos años en temas relacionados con la Realidad Aumentada y más concretamente con su aplicación al campo del estudio del patrimonio cultural perdido u oculto. Es en este ámbito de investigación donde se centra mi proyecto, el cual parte de un desarrollo previo que paso a explicar para dar a conocer el punto de partida del mismo.

1.1 Antecedentes

Este trabajo fin de máster parte de uno anterior [1] donde se realizaba la localización geoespacial mediante imágenes. Esta aplicación, tomaba una fotografía de la plaza San Felipe, la enviaba a un servidor que realizaba los cálculos de la localización 3D y retornaba la fotografía al terminal móvil con la imagen de la torre ya superpuesta con un procesado que simulaba el efecto de foto antigua, Figura 1.1.1.

Figura 1.1.1. Ejemplo del funcionamiento de la aplicación. A la izquierda, la imagen tomada, en el medio, imagen del objeto 3D superpuesto y a la derecha imagen con post-procesado.

Sin embargo, este proceso resulta costoso computacionalmente. Todo este proceso tardaba un total de 6,18 segundos, lo cual resulta no ser suficiente para una aplicación en tiempo real. Observando la tabla de tiempos mostrada en el trabajo previo, Tabla 1.1.1, se observa que uno de los puntos críticos es el envío y recibo de las imágenes mediante comunicación 3G. Por ello se pensó en proveer un código que ejecutase todo el proceso de localización en la misma aplicación y no en un servidor externo. Con esto, se evitaría el retraso ocasionado por el envío de datos, que asciende a un total de 2,97 segundos. También se pensó en reducir la resolución con la que se tomaba la fotografía para simplificar todos los cálculos, hecho que se estudiará en este proyecto en las secciones posteriores.

Proceso	*Tiempo (s)*
Envío de imagen al servidor (red 3G)	1.68
Estimación de la posición (esperada)	0.98
Renderizado y post-procesado	2.23
Envío del resultado del servidor (red 3G)	1.29
TOTAL	**6.18**

Tabla 1.1.1. Tiempos de ejecución de la aplicación en segundos

La extracción de características de la imagen, o *keypoints*, son los puntos que se utilizan en el código para realizar la localización. En este caso, se utilizaron los descriptores tipo SIFT [2], al ser muy robustos en su cometido. Tras esta extracción de puntos clave, se realizaba emparejamiento y un filtrado de aquellos que no son relevantes en la localización 3D como pueden ser los formados por sombras, cambios de iluminación o distractores como, por ejemplo, personas paseando por la plaza San Felipe. Sin embargo, estos descriptores SIFT, a pesar de ser muy óptimos en su cometido, producen un gran retardo en el cálculo de la posición. Por ello, se decidió explorar otro tipo de descriptores capaces de alcanzar un resultado similar.

Por otra parte, el filtrado utilizado en el desarrollo anterior se basaba en el método RANSAC[3], que se explicará posteriormente. Este filtrado, resulta insuficiente para nuestro cometido, más incluso si utilizamos un descriptor menos robusto que SIFT para optimizar el apartado de tiempo. En este trabajo se explicarán técnicas adicionales para filtrar emparejamientos falsos o no coherentes.

1.2 Objetivos

El objetivo principal de este trabajo es la mejora de un sistema de localización 3D en un entorno conocido para poder ser ejecutado en el mismo terminal móvil sin necesidad de enviar la información a un servidor remoto, ejecutándose en un tiempo razonable. Como se explicará posteriormente, la extracción de características en la imagen es uno de los puntos más relevantes en cuanto a tiempo computacional se refiere.

Por tanto, los objetivos a cumplir desglosados serían:

1) Cambio a un descriptor más veloz.
2) Añadir técnicas de filtrado que aseguren unos emparejamientos más fiables y robustos.
3) Validación de estos resultados en Matlab.
4) Implementación de la aplicación en lenguaje c++.

1.3 Organización de la memoria

La memoria cuenta con una estructura dividida en 5 partes:

- Apartado 2: se explica detalladamente el funcionamiento del código previo, así como las mejoras introducidas posteriormente.

- Apartado 3: se analizan los datos obtenidos en Matlab. Se detalla la modificación de los parámetros, así como las bases de datos utilizadas.

- Apartado 4: se desarrolla la implementación del código de Matlab en c++, así como los problemas habidos y los resultados obtenidos.

- Apartado 5: se ilustra el funcionamiento de la herramienta de VisualSFM para la creación de la nube de puntos en 3D.

- Apartado 6: se recogen las conclusiones obtenidas.

2. Emparejamiento de imágenes basado en descriptores

Como ya he mencionado anteriormente, este trabajo parte de un código previo ya implementado en Matlab con el cual se realizaba el emparejamiento entre imágenes. En este apartado de la memoria se explicará parte de este código y, posteriormente, se presentarán los cambios y mejoras introducidos para que la aplicación funcione en un terminal móvil sin la necesidad de un servidor remoto.

2.1 Funcionamiento del código previo

El código dado es capaz de emparejar los puntos similares de dos imágenes (descriptores) pertenecientes a la misma localización, aunque éstas sean tomadas desde distintos puntos de vista, ya sea respecto a escala, rotación y/o perspectiva. Para ello, se deben extraer estos descriptores de la imagen y efectuar un emparejamiento con los descriptores de la base de datos.

Los descriptores utilizados como puntos relevantes de las imágenes han sido tomados con el algoritmo SIFT (Scale-Invariant Feature Transform) [2]. Estos descriptores son invariantes a escala y rotación y, parcialmente, invariantes a cambios de iluminación, lo que lo hace idóneo para este tipo de aplicaciones pensadas para ser utilizada en exteriores. La localización de los puntos se desarrolla con información tanto del dominio espacial como del frecuencial para reducir errores debidos a oclusiones o ruido. Además de la localización del punto, se extrae información perteneciente a la escala y orientación del descriptor, lo cual resulta útil de cara al emparejamiento de descriptores.

Sin embargo, estos descriptores SIFT presentan como principal inconveniente el alto coste computacional que conllevan. Por ello, se plantea explorar otro tipo de descriptores cuyas prestaciones sean parecidas pero que conlleve un menor tiempo de cálculo, al tener que realizarse en el terminal móvil.

Una vez calculados los descriptores en ambas imágenes, se utiliza una métrica para determinar lo que se parecen los unos a los otros. El emparejamiento entre descriptores es en función de su distancia de Bhattacharyya [4]. Esta distancia, mide la similitud entre dos distribuciones de probabilidad y viene dada por la siguiente función (1).

$$D_B(p,q) = -\ln\big(BC(p,q)\big)$$
$$BC(p,q) = \sum_{x\in X}\sqrt{p(x)q(x)} \qquad (1)$$

Diremos que dos puntos cuyos descriptores presenten un alto valor en el coeficiente de Bhattacharyya (BC) presentan una alta similitud.

El problema surge cuando comparamos miles de descriptores entre imágenes. Para minimizar el tiempo de cálculo se utiliza una representación de los mismos en forma de árbol binario que se conoce como kd-tree.

El algoritmo de árbol k-d, es un árbol binario en el que cada nodo hoja es un punto de la dimensión k. Este algoritmo, permite crear un buscador del vecino más cercano a partir de unos datos dados. El árbol se crea con el conjunto de puntos encontrados en la 'imagen uno'. Posteriormente, se testean en ese modelo los puntos encontrados en la 'imagen dos', creando así el emparejamiento entre puntos de ambas imágenes.

La Figura 2.1.1, muestra un ejemplo de emparejamiento en el que se utilizó el algoritmo kd-tree para realizar el emparejamiento usando la distancia Euclídea.

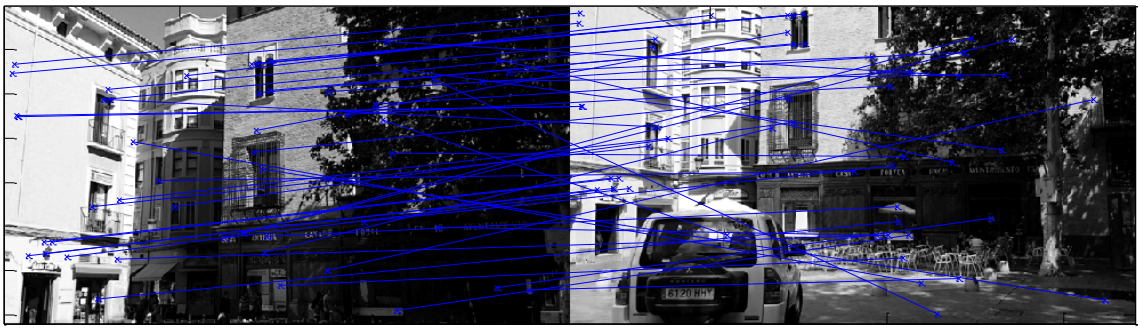

Figura 2.1.1 Emparejamiento entre dos imágenes de la plaza San Felipe con kd-tree

Como podemos ver, el emparejamiento resulta bastante caótico y se pueden apreciar ciertos puntos que han sido mal emparejados. Por ello, se deben realizar filtrados de estos puntos erróneos que se explicarán en los siguientes puntos.

En concreto, en el código anterior se utilizaba el algoritmo de RANSAC (Random sample consensus) [3], que consiste en un método iterativo que busca descartar los *outliers* de un conjunto. RANSAC estudia y valora los datos dados creando un modelo para descartar aquellos valores que no estén sujetos a este modelo, es decir, valores atípicos.

En una primera instancia, el algoritmo escoge un subconjunto de puntos catalogándolos como posibles *inliers* o puntos válidos. Con estos *inliers*, se crea un modelo hipotético y se testean el resto de datos. Si el modelo consigue clasificar bien varios datos, se da como bueno. Este algoritmo se repite un número fijo de veces, quedándose al final con el modelo que más éxito haya tenido. En la Figura 2.1.2. se puede ver un ejemplo 2D de su funcionamiento; la línea azul representa el modelo creado por RANSAC que descartaría los *outliers* rojos.

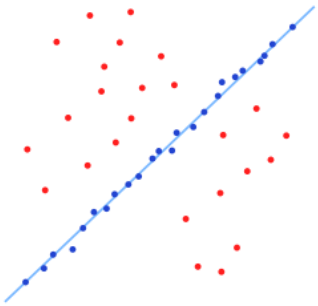

Figura 2.1.2 Representación del funcionamiento de RANSAC

Si continuamos con el ejemplo anterior, al aplicar el filtrado con RANSAC se aprecia una mejora sustancial respecto al resultado anterior. En la Figura 2.1.3, podemos ver en azul los emparejamientos correctos y en rojo, algunos *outliers* que se presentan a pesar del filtrado con este método.

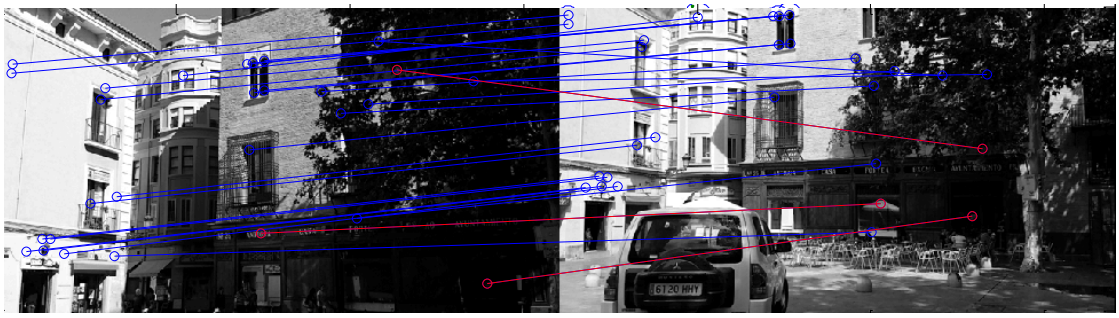

Figura 2.1.3 Filtrado tras RANSAC, mostrando en rojo los outliers.

Una vez analizado el estado de la técnica de los algoritmos de emparejamientos, en este proyecto se dio la constancia de que era insuficiente para realizar un emparejamiento óptimo. Nuestro objetivo es que los emparejamientos sean muy fiables para reconstruir posteriormente la nube de puntos del mundo 3D. Tras filtrar con el algoritmo RANSAC, el problema todavía presenta espurios que han de ser eliminados. Por tanto, se decidió añadir nuevas técnicas de filtrado que pasamos a describir a continuación.

2.2 Mejoras propuestas

Las mejoras propuestas van en dos direcciones principalmente: la búsqueda de otro tipo de descriptores con menor coste computacional y la eliminación de emparejamientos espurios que se ha comprobado que el algoritmo de RANSAC no es capaz de eliminar completamente.

2.2.1 Tipo de descriptor (BRISK)

Uno de los principales retos a abordar en este proyecto es el decremento del tiempo de cómputo del algoritmo de la aplicación. El tiempo que la aplicación tarda en realizar el cálculo de descriptores SIFT resulta altamente costoso. Es por ello, que en esta nueva versión del algoritmo se optó por utilizar otro tipo de descriptores que, si bien no son tan precisos como los originales, realizan su función en un menor tiempo de cómputo. En el Anexo A, se introducen distintas alternativas a SIFT y que han sido utilizados en este proyecto para realizar comparaciones con el mismo.

A diferencia de SURF [5], que debe calcular un vector de 64 dimensiones o SIFT que aumenta este vector a 128 dimensiones y además de este cálculo realiza operaciones de filtrado y convolución; BRISK [6] es un descriptor binario de 512 puntos que efectúa el cómputo sólo entre los puntos más cercanos del posible descriptor (Figura 2.2.1.1). Como se explicó en el Anexo A, de los descriptores binarios estudiados, resulta ser el más óptimo al ser invariante respecto a escala y rotación.

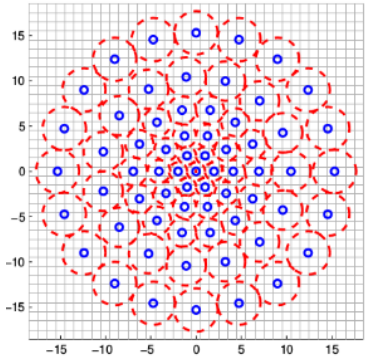

Figura 2.2.1.1 En azul, la localización de los keypoints y en rojo los píxeles cercanos con los que se realiza el cómputo

2.2.2 MatchFeatures

A diferencia de SIFT, BRISK proporciona un descriptor binario, no siendo por tanto la distancia Euclídea o la de Bhattacharyya las mejores alternativas para comparar entre dos descriptores. Afortunadamente, la herramienta Matlab dispone de una función interna para realizar el emparejamiento de puntos entre imágenes *matchFeatures*[1]. A partir de dos conjuntos de datos, realiza el emparejamiento en función del tipo de descriptor que está siendo utilizado. En esta función se puede cambiar el valor de ciertos parámetros como el umbral o el máximo ratio permitido.

2.2.3 Filtrado por ángulos y escala

El estar trabajando con descriptores nos da una información extra de cada *keypoint* de la imagen. Éstos, utilizan un gradiente calculado por el algoritmo del descriptor para obtener los parámetros de ángulo y escala de cada punto clave. Podemos pensar que el mismo punto, en dos imágenes distintas, debe tener la misma escala y ángulo, lo que nos puede servir para filtrar aquellos puntos que no cumplan este requisito. El filtrado realizado por ángulos y escala, consiste en comparar el ángulo y la escala de un emparejamiento y descartarlo si no cumple un umbral.

En cuanto al filtrado por ángulos, primero se realiza el cálculo de la diferencia de ángulos entre todos los *keypoints* emparejados y se calcula la mediana de estos valores diferencia. Posteriormente, se descartan aquellos emparejamientos cuyo valor de la diferencia de ángulos esté comprendido entre el valor de la mediana ± π/8. Destacar que el rango de valores de los ángulos obtenidos por el descriptor se encuentra entre -π y π.

[1] http://es.mathworks.com/help/vision/ref/matchfeatures.html

En cuanto a la información de escala, se realiza un proceso parecido al de los ángulos, salvo que, en este caso, en lugar de usar la diferencia como medida, se utiliza la fracción entre escalas. Una vez calculada esta fracción entre todos los *keypoints* emparejados, se obtiene la mediana de estos valores y se descartan aquellos *keypoints* cuya fracción no esté comprendida entre la mitad y el doble del valor de la mediana.

Por hacerlo más visual y entendible, diremos que el criterio a seguir para realizar el filtrado por ángulos y escala es mantener aquellos puntos que cumplan los siguientes criterios (2) y (3).

a) Filtrado por ángulos

$$angulo1 - angulo2 < mediana + \frac{\pi}{8}$$

o

$$angulo1 - angulo2 > mediana - \frac{\pi}{8} \qquad (2)$$

b) Filtrado por escala

$$\frac{escala1}{escala2} < 2 * mediana \quad \text{o} \quad \frac{escala1}{escala2} > \frac{mediana}{2} \qquad (3)$$

Podemos extraer la escala y el ángulo del ejemplo anterior (Figura 2.2.3.1) para ver cómo se produce este filtrado. Los círculos que rodean a cada punto representan la escala del descriptor, mientras que el vector verde indica el ángulo que se ha calculado.

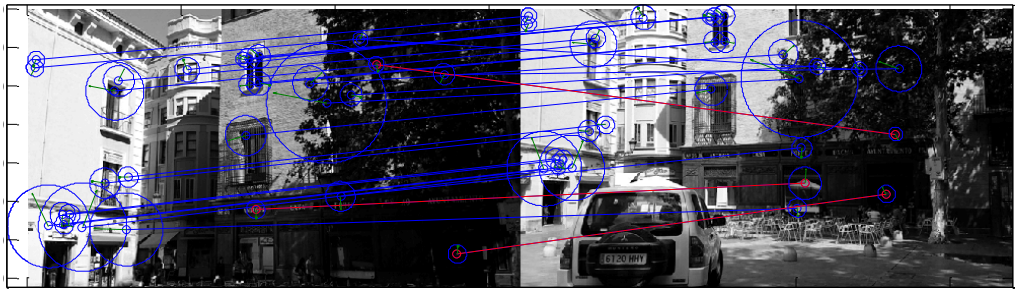

Figura 2.2.3.1 Emparejamientos con representación de ángulo y escala de los descriptores

Si observamos detenidamente el caso anterior, se puede ver que los dos *matches* erróneos inferiores no cumplen las condiciones impuestas anteriormente. En la Figura 2.2.3.2, hay un zoom de uno de estos errores en el que se ve que el ángulo dista en torno a 180º entre descriptores, por lo que no cumpliría el umbral fijado y será eliminado.

Figura 2.2.3.2 Zoom de un emparejamiento erróneo

Descartando estos emparejamientos erróneos, obtenemos el resultado de la Figura 2.2.3.3, todavía con un emparejamiento erróneo que eliminaremos con otro tipo de filtrado en el siguiente apartado.

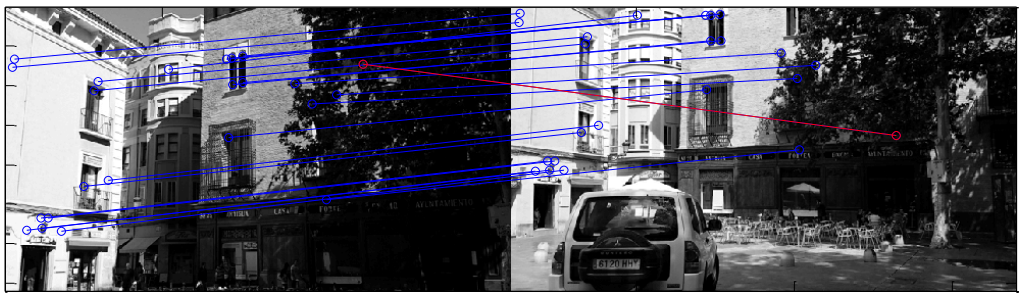

Figura 2.2.3.3 Filtrado tras la comparación de ángulo y escala de los descriptores

2.2.4 Filtrado por triángulos de Delaunay

Una vez se han filtrado los puntos mediante la información de ángulo y escala, se procede a realizar un filtrado algo más complejo teniendo en cuenta la información de los puntos adyacentes. La triangulación de Delaunay es una malla o red de triángulos donde todos los triángulos cumplen la condición de Delaunay, que nos dice que los 3 vértices que forman cada triángulo del conjunto de puntos, deben formar una circunferencia que no contenga ningún otro vértice. Además, la triangulación de Delaunay presenta las siguientes propiedades:

- Todos los puntos están conectados entre sí formando el mayor número de triángulos posible sin que se crucen sus aristas.
- Los triángulos se forman de la manera más regular posible, maximizando el ángulo mínimo de cada uno de ellos.
- Es unívoca si en ningún borde de la circunferencia hay más de tres vértices.

En la Figura 2.2.4.1, podemos ver como se forma una arista ilegal que no tendría cabida en la definición propuesta anteriormente de la condición de Delaunay. El triángulo formado por los 3 vértices que une esta arista, circunscribe a un vértice extra, 4 en total, por lo que no cumple la condición impuesta anteriormente.

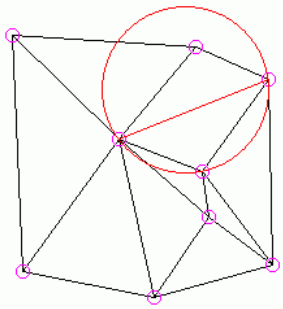

Figura 2.2.4.1 Arista ilegal en la triangulación de Delaunay

Esto nos permite crear una malla de triángulos en una de las imágenes que estamos emparejando para realizar una comparación coherente. Puesto que la triangulación de Delaunay debe ser unívoca, resulta comprensible pensar que, si estamos emparejando puntos similares en las imágenes, su diagrama de Delaunay debe ser similar.

Una vez creada la malla de triángulos que cumplen esta condición en ambas imágenes en las que estamos trabajando, se procede a su comparación entre ellos. El algoritmo de filtrado escoge un triángulo de la primera imagen y su homónimo de la segunda imagen (el formado por los puntos emparejados de cada vértice del primer triángulo).

De cada uno de estos dos triángulos, se obtiene el baricentro y el ángulo formado entre el baricentro y cada vértice del correspondiente triángulo como se muestra en la Figura 2.2.4.2 en los ángulos α_1, α_2, α_3. Posteriormente, se realiza la operación diferencia entre estos ángulos y los dados por el descriptor, α_1', α_2', α_3'.

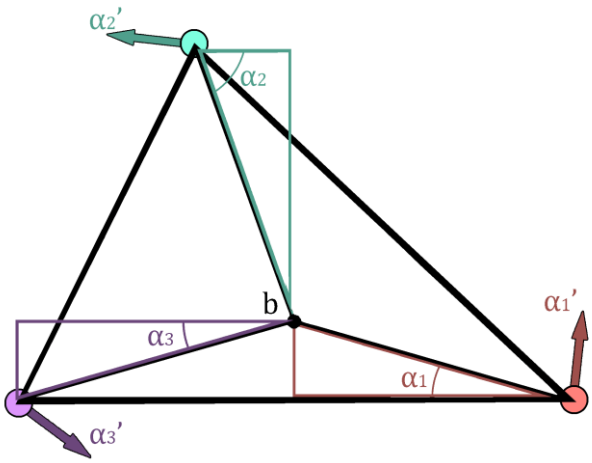

Figura 2.2.4.2 Cálculo de baricentro y ángulos correspondientes con los vértices. También se representan los ángulos dados por el descriptor en forma de flechas.

Este valor obtenido se compara entre puntos emparejados de ambas imágenes y si la diferencia entre dos puntos no supera el umbral de $\pi/4$ el punto lo consideraremos válido. En caso contrario, se descartarán los puntos del emparejamiento final. Viendo el ejemplo anterior, crearíamos la malla de triángulos Delaunay para realizar los cálculos ya explicados como se muestra en la Figura 2.2.4.3.

Figura 2.2.4.3 Filtrado tras la comparación de ángulo y escala de los descriptores con la malla de triangulos de Delaunay

Se realizarían los cálculos explicados anteriormente para cada uno de los triángulos, (baricentros, ángulos, comparaciones con umbrales…). Tras descartar los puntos que no cumplen la condición impuesta, obtenemos el resultado de la Figura 2.2.4.4, eliminando el *match* erróneo y dejándonos un emparejamiento totalmente correcto.

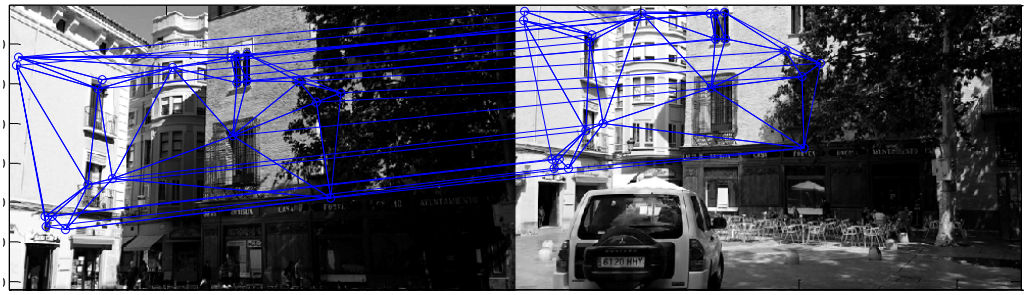

Figura 2.2.4.4 Emparejamientos tras el filtrado de Delaunay

2.2.5 Recuperación de puntos

Al final de este algoritmo, se introdujo una nueva función para recuperar emparejamientos correctos que hemos descartado tras los filtrados. Una vez se ha realizado la extracción de los descriptores BRISK, el emparejamiento entre ambas imágenes y los filtrados explicados anteriormente, podemos asumir que los puntos resultantes son válidos con una fiabilidad suficientemente alta. Con estos puntos emparejados, podemos crear una matriz base acorde al cambio de coordenadas entre ambas imágenes dado por los descriptores.

Una vez se ha creado la matriz, se validan los puntos que hemos descartado previamente con los diferentes tipos de filtrado para comprobar si son fieles al emparejamiento calculado.

Si el punto obtenido tras la reconstrucción con esta matriz se encuentra próximo al punto descartado, lo recuperaremos como punto válido para realizar la localización espacial. En función de la resolución, se establecerá el umbral que valide la reconstrucción de los puntos.

2.2.6 Ajuste de parámetros

Si bien se trabajará siguiendo el mismo esquema que el código previamente implementado, se han de ajustar los parámetros existentes en la nueva propuesta. En el siguiente apartado de la memoria se explicarán los parámetros modificados, así como las diferentes pruebas de test realizadas para evaluar el problema abordado.

3. Resultados de Matlab

3.1 Parámetros a modificar

Tanto el algoritmo BRISK como el resto del código de la aplicación presentan muchas variables y parámetros que se pueden configurar para conseguir un mejor resultado de nuestro problema concreto. Para este proyecto, se realizó un estudio de estos parámetros.

Parámetros de la extracción de descriptores

La función para extraer los descriptores BRISK en Matlab[1] nos permite modificar tres parámetros para mejorar su resultado. Estos parámetros son:

- **MinContrast**: siendo éste la mínima diferencia de intensidad entre un punto y la región que lo rodea. En la práctica, se dio la circunstancia de que, si utilizábamos este valor de forma muy restrictiva, la mayoría de los puntos que detectaba eran provocados por *outliers* provenientes de árboles o sombras. Estos puntos no son útiles para realizar la detección geoespacial, pues son variantes en el tiempo y no formarían parte de una buena reconstrucción 3D de la plaza. Por tanto, se tomaron valores de este parámetro no muy restrictivos para coger la mayoría de puntos, independientemente de su contraste, filtrando posteriormente aquellos que no sean de interés. Se trata de un valor porcentual.

- **MinQuality**: es el parámetro que impone la calidad mínima que deben tener los descriptores para que la función los devuelva como correctos. Al igual que MinContrast, se trata de un valor porcentual.

- **NumOctaves**: Los descriptores son detectados mediante capas organizadas en octavas de la imagen piramidal. Con este parámetro podemos elegir el número de octavas con el que realizará los cálculos. El rango de valores recomendado en Matlab se hallaría en valores enteros entre 1 y 4.

- **ROI (Rectangular region)**: con este parámetro podemos delimitar los píxeles en los que buscar descriptores de las imágenes. Sin embargo, como en nuestro problema no partimos de un patrón concreto en las imágenes, utilizaremos toda la región de la imagen para detectar los puntos BRISK.

Parámetros del emparejamiento

La función utilizada para realizar el emparejamiento de descriptores en Matlab fue la de *matchFeatures*[2], pues en descriptores BRISK daba un mejor resultado que la ya implementada tipo *kd-tree*. Esta función, tiene dos parámetros a seleccionar.

- **MatchThreshold**: es el umbral con el que se realiza el emparejamiento. Viene dado como el porcentaje de la distancia que se debe hallar para un obtener un emparejamiento perfecto. Admite valores entre 0 y 100.

[1] http://es.mathworks.com/help/vision/ref/detectbriskfeatures.html

[2] http://es.mathworks.com/help/vision/ref/matchfeatures.html

• **MaxRatio**: Este parámetro se utiliza para rechazar emparejamientos ambiguos y al aumentar este valor se obtienen más puntos emparejados. Se encuentra en un rango entre 0 y 1.

3.2 Bases de datos

Antes de realizar la búsqueda de parámetros con las imágenes de la plaza San Felipe, se procedió a comprobar la eficacia de los descriptores BRISK en una base de datos más sencilla. Esta base de datos la denominaremos como 'toy story' y está formada por una misma ilustración que sufre cambios de rotación y escala como se puede ver en la Figura 3.2.1. Consta de un total de 100 imágenes con una dimensión de 640x480 píxeles cada una. Al no tener imágenes no emparejables en ella, el problema resulta más sencillo que en un entorno complejo como la plaza San Felipe. Todas las imágenes deberían ser capaz de emparejarse entre ellas.

Figura 3.2.1 Imágenes de la base de datos 'toy story'

Tras realizar diversas pruebas, variando los parámetros de *BRISK*, de *matchFeatures* y *RANSAC*, se vio que el algoritmo de emparejamiento de *keypoints* funcionaba bastante bien, aunque no es perfecto, siendo necesario la utilización de los filtrados posteriores propuestos. En la Figura 3.2.2, podemos ver un ejemplo de un buen emparejamiento producido entre dos imágenes de esta base de datos.

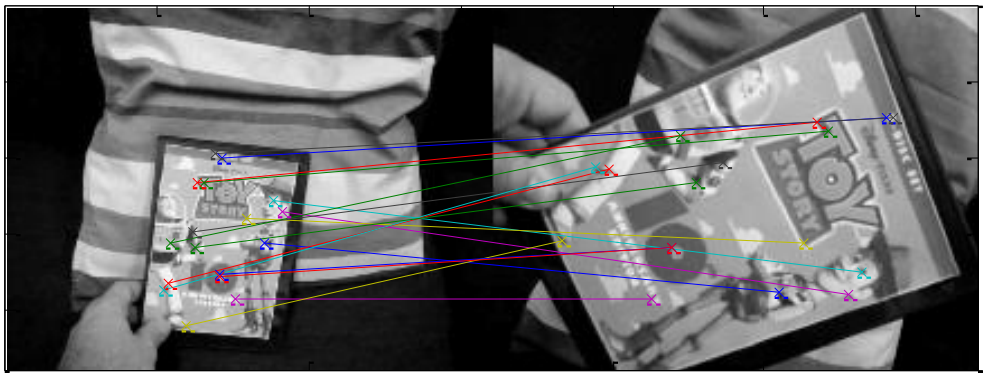

Figura 3.2.2 Emparejamiento de la base de datos 'toy story' utilizando el descriptor BRISK

Una vez validado que se producían emparejamientos correctos, se procedió a trabajar con la base de datos de la plaza San Felipe, más compleja y con el problema añadido de que no sólo hay que emparejar las imágenes correctamente, sino que, además, se debe asegurar que las imágenes que no son emparejables no sean emparejadas por el sistema de manera errónea; es decir, que no se produzcan falsos positivos.

La base de datos de San Felipe está formada por un conjunto de 107 imágenes de 2048 píxeles de alto y con un alto variable de 1152 o 1536 píxeles en función de si la imagen fue tomada en formato 16:9 o 4:3 respectivamente. Son fotos tomadas en la misma plaza San Felipe de Zaragoza con varios terminales móviles para obtener distintos tipos de colorimetrías, exposición o ISO de la imagen, así como ser capaces de tratar diversos problemas procedentes del tipo de dispositivo que captura la imagen. Además, fueron tomadas a distintas horas del día para contemplar múltiples escenarios, posibles sombras o cambios de iluminación. En la Figura 3.2.3 podemos ver varias imágenes ejemplo de esta base de datos.

Figura 3.2.3 Algunas imágenes de la base de datos 'San Felipe'

3.3 Comparación de algoritmos

Antes de proceder con los resultados obtenidos en Matlab, en esta parte de la memoria se va a exponer un análisis de varios detectores de puntos clave y descriptores. Se analizará el tiempo medio que tardan en computar los algoritmos, así como el número medio de puntos que se obtienen. Decir, que un mayor número de puntos no indica unos puntos mejores, pero es un parámetro más que podríamos tener en cuenta para comparar este tipo de algoritmos y que se suele utilizar en este tipo de comparaciones. Además, posteriormente al hacer el emparejamiento de puntos clave, el tiempo aumentará a mayor cantidad de puntos.

Como nuestro parámetro crítico es el tiempo, se realizó el estudio en la base de datos San Felipe con varias resoluciones para percibir la mejora que supone trabajar con imágenes de más baja resolución. Para ello, se redimensionaron las bases de datos y se procesaron los algoritmos de BRISK[6], HARRIS[7], HoG, MSER[8], SIFT [2] y SURF[5]. Los resultados obtenidos en función de esta resolución se pueden apreciar en las siguientes tablas adjuntas:

160x90	*BRISK*	*HARRIS*	*HoG*	*MSER*	*SIFT*	*SURF*
Tiempo (s)	0.08	0.11	0.005	0.05	0.02	0.02
Nº de puntos	79.33	68.53	7.09e+03	60.83	58.82	66.41

Tabla 3.3.1 Comparación de descriptores con la base de datos en resolución 160x90 píxeles

320x180	*BRISK*	*HARRIS*	*HoG*	*MSER*	*SIFT*	*SURF*
Tiempo (s)	0.1	0.16	0.01	0.11	0.11	0.14
Nº de puntos	311.77	269.51	3.22e+04	171.6	253.06	241.11

Tabla 3.3.2 Comparación de descriptores con la base de datos en resolución 320x180 píxeles

640x360	*BRISK*	*HARRIS*	*HoG*	*MSER*	*SIFT*	*SURF*
Tiempo (s)	0.09	0.16	0.03	0.3	0.34	0.14
Nº de puntos	1.08e+03	833.87	1.35e+05	536.40	989.37	785.08

Tabla 3.3.3 Comparación de descriptores con la base de datos en resolución 640x360 píxeles

1280x720	*BRISK*	*HARRIS*	*HoG*	*MSER*	*SIFT*	*SURF*
Tiempo (s)	0.13	0.29	0.09	0.5	0.98	0.35
Nº de puntos	3.6e+03	2.66e+03	5.09e+05	1.51e+03	3.36e+03	2.87e+03

Tabla 3.3.4 Comparación de descriptores con la base de datos en resolución 1280x720 píxeles

2048x1152	*BRISK*	*HARRIS*	*HoG*	*MSER*	*SIFT*	*SURF*
Tiempo (s)	0.74	2.14	0.74	3.97	6.57	2.16
Nº de puntos	7.68e+03	5.48e+03	1.38e+06	3.82e+03	9.09e+03	5.94e+03

Tabla 3.3.5 Comparación de descriptores con la base de datos en resolución 2048x1152 píxeles

En la Figura 3.3.1, podemos ver una comparación del número de puntos obtenidos. Estos valores no son tan relevantes en nuestro problema, pues un mayor número de puntos implica un mayor tiempo de cómputo que es nuestro parámetro más crítico. Aunque si nos fijamos en otros trabajos de descriptores, [9], [10] o [11], es un parámetro que se suele utilizar para realizar la comparación de los mismos.

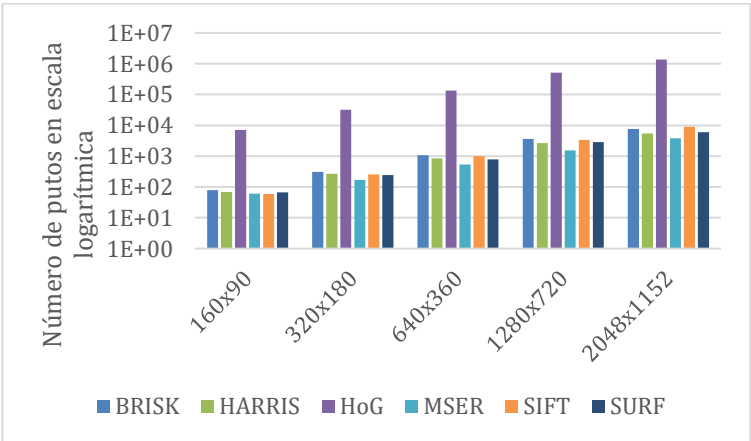

Figura 3.3.1 Número de puntos obtenidos con cada descriptor para cada tipo de resolución de 'San Felipe'

En cuanto al tiempo, se puede apreciar que el cambiar de BRISK a SIFT o a la versión rápida de este último, SURF, no es tan importante para resoluciones bajas, sin embargo, en cuanto la resolución aumenta, el tiempo de procesado resulta ser relevante. En la Figura 3.3.2, podemos ver que BRISK mejora en casi un factor de 8 para la resolución de 720p y en un factor de 9 en la resolución máxima.

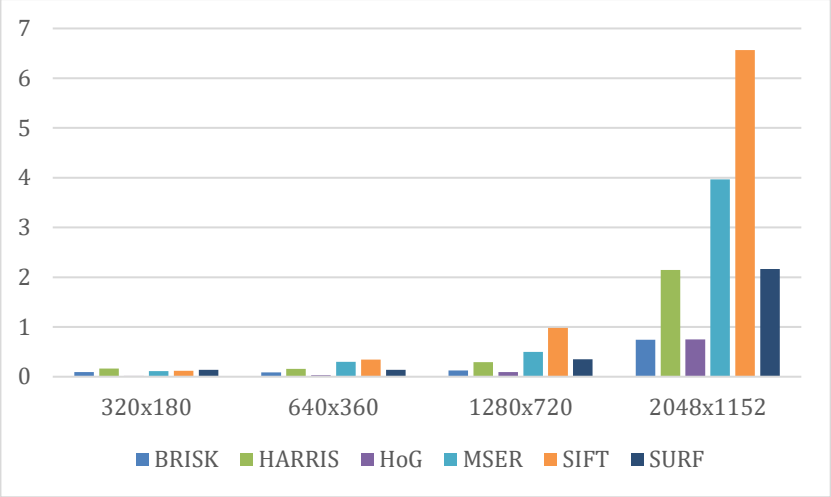

Figura 3.3.2 Tiempos de ejecución de cada descriptor en función de la resolución de 'San Felipe'

3.4 Resultados obtenidos

Evaluar los resultados obtenidos en este tipo de aplicaciones puede llegar a resultar complicado. Cuando dos imágenes son emparejadas por el algoritmo, esto no nos asegura que el emparejamiento que se ha producido sea el correcto, sino que se debe comprobar visualmente. En nuestro caso, daremos un emparejamiento válido entre dos imágenes cuando éste contenga al menos 8 puntos emparejados tras realizar los filtrados.

Para poder realizar la evaluación de algoritmos de una forma más eficiente, se creó una tabla de emparejamientos entre imágenes con el descriptor SIFT (Figura 3.4.1) que podemos asumir que su funcionamiento es el correcto al tratarse del utilizado en el proyecto previo a éste. Cuando se evalúa la funcionalidad de un sistema nuevo, se compara con esta tabla de emparejamientos. Esto nos permite evaluar el sistema no sólo en número de aciertos, sino en cuestión de falsos positivos o falsos rechazos.

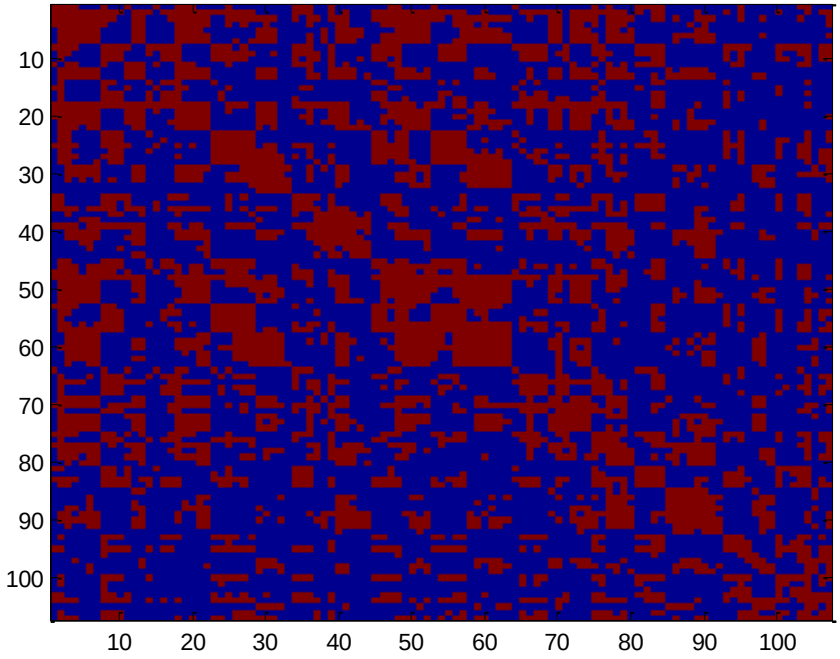

Figura 3.4.1 Imagen binaria mostrando los emparejamientos entre imágenes utilizando SIFT

Cada columna y cada fila de la Figura 3.4.1 corresponden al número de la imagen. La matriz marca en rojo cuando dos imágenes son emparejadas correctamente (se producen más de 8 emparejamientos tras los filtrados). Esta matriz, la compararemos con las obtenidas en los resultados posteriores para evaluar su funcionamiento. Nuestro principal interés no es tanto minimizar el falso rechazo, es decir, que no empareje dos imágenes que deben emparejarse, sino que no se creen falsos positivos o lo que sería lo mismo, que dos imágenes que según SIFT no se emparejan resulten ser emparejadas.

El algoritmo utilizando SIFT como descriptor es capaz de emparejar una media de 37,78 imágenes para cada imagen de la base de datos de San Felipe como se muestra en la Figura 3.4.2 .

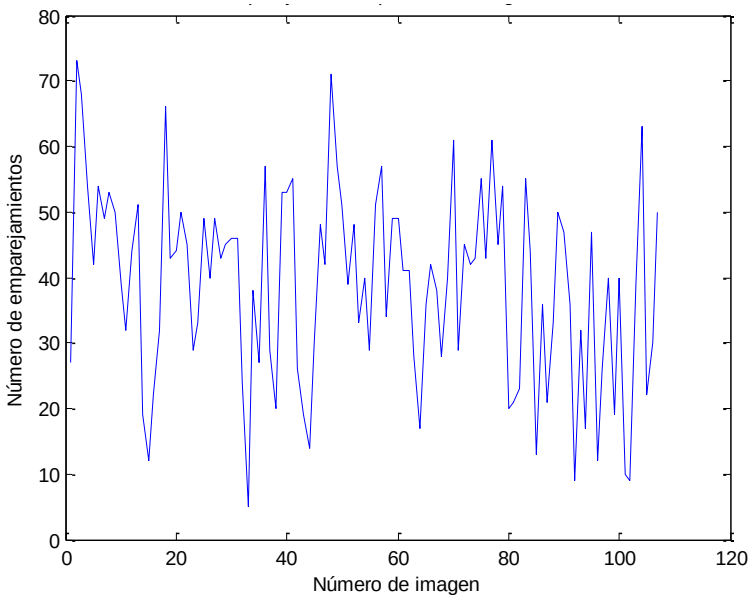

Figura 3.4.2 Número de emparejamientos para cada imagen utilizando SIFT

Por tanto, el objetivo marcado en esta fase del proyecto fue el de ajustar los parámetros del algoritmo BRISK para conseguir minimizar los falsos positivos consiguiendo a su vez el máximo número de emparejamientos posible. Añadir también, que esta forma de testear los próximos algoritmos no nos indica un resultado preciso al 100%, puesto que puede darse el caso en el que algunas imágenes que se debieran emparejar no fueran detectadas por el algoritmo SIFT y sí por el algoritmo BRISK. Sin embargo, es una forma cómoda y rápida de poder evaluar este tipo de algoritmos de visión por computador sin tener que ver todos los emparejamientos de forma manual.

A continuación, se expondrán los resultados obtenidos con BRISK (nuestro principal objetivo), así como con los algoritmos de MSER y SURF para poder realizar una comparación entre ellos. Las pruebas se realizaron con estos dos algoritmos, y no con otros, puesto que no resultaba demasiado complejo modificar el código para estos dos descriptores en Matlab, una vez teníamos el código de BRISK ya funcional. Para analizar cómo de bueno es cada algoritmo analizado, se evaluará la cantidad de imágenes que se emparejan entre sí, el porcentaje de falsos positivos que genera, así como la exactitud (4) y la precisión (5) definidas de la siguiente forma:

$$Exactitud = \frac{ciertosPositivos + ciertosNegativos}{ciertosPositivos + falsosPositivos + ciertosNegativos + ciertosNegativos} \qquad (4)$$

$$Precision = \frac{ciertosPositivos}{ciertosPositivos + falsosPositivos} \qquad (5)$$

BRISK 640x360

El primer caso que se expondrá, es el formado con la base de datos de la plaza San Felipe en resolución de 640x360 para las primeras imágenes y 640x480 a partir de la imagen 70. Tras realizar una búsqueda de los parámetros más óptimos en la configuración de BRISK, el mejor caso obtenido fue con los parámetros mostrados en la Tabla 3.4.1:

MinQuality	0,0443
NumOctaves	6
MinContrast	0,0167
MatchTh	99,7551
MaxRt	0,7291

Tabla 3.4.1 Tabla de parámetros empleados en BRISK

En la Figura 3.4.3 podemos ver los emparejamientos entre las imágenes en color rojo. Se observa que los emparejamientos resultan ser muy escasos y que entre imágenes de diferentes alturas no se logra formar emparejamientos de forma correcta.

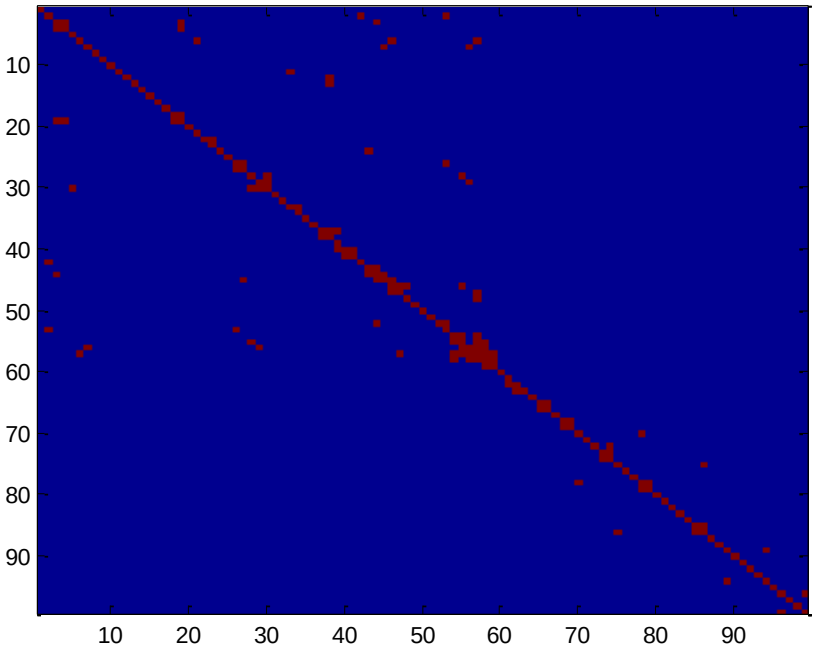

Figura 3.4.3 Emparejamientos entre imágenes utilizando BRISK

En cuanto a los emparejamientos producidos, en la Figura 3.4.4, podemos ver los emparejamientos que recibe cada imagen con el resto de la base de datos. Vemos que prácticamente la mitad de las imágenes, un total de 43, no logran emparejarse con ninguna otra imagen de la base de datos y que cada imagen se empareja con una media de 1,01 imágenes de la base de datos.

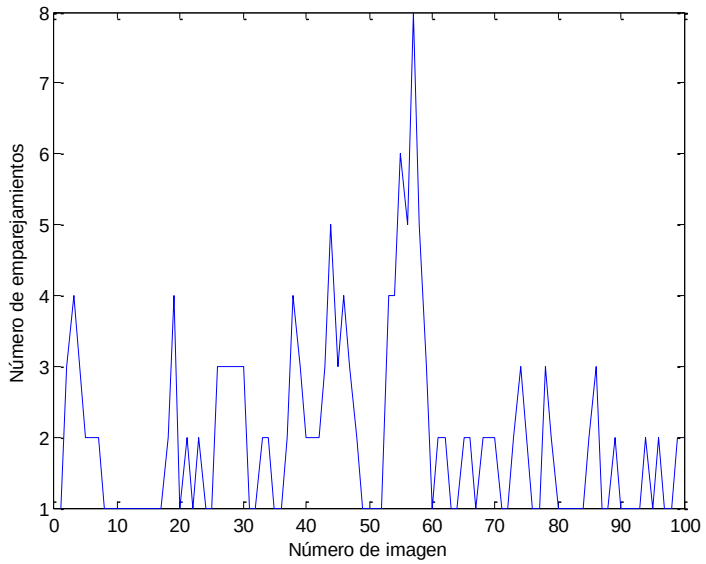

Figura 3.4.4 Número de emparejamientos para cada imagen utilizando BRISK

En la Figura 3.4.5, vemos que en términos de buscar que el algoritmo no produjese falsos positivos, funciona correctamente, ya que tan sólo se produce un 0,01% de falsos positivos. También en términos de precisión y exactitud cumple con los requerimientos de nuestra aplicación.

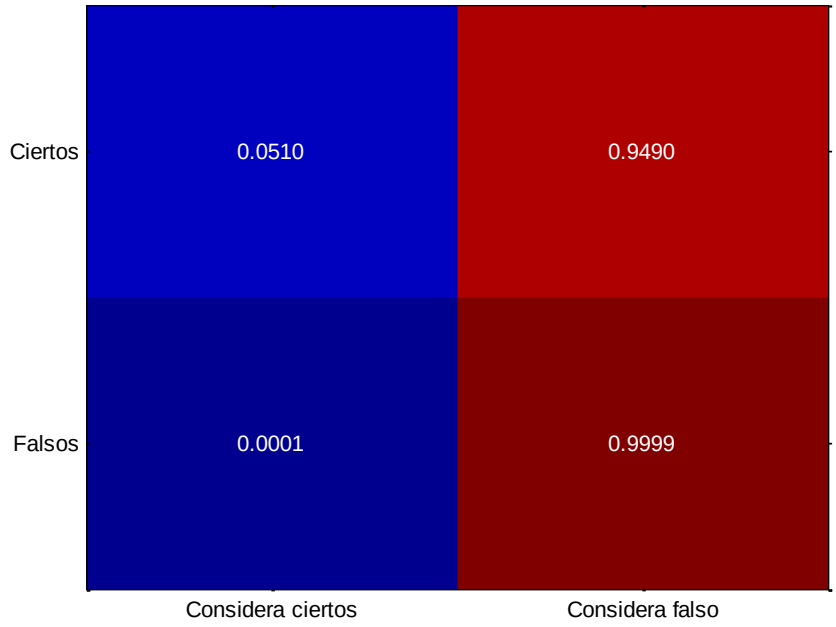

Figura 3.4.5 Matriz de confusión obtenida para el caso de BRISK de baja resolución

Accuracy	0.81
Precision	0.99

Sin embargo, el número de emparejamientos dista mucho de ser suficiente para conseguir una nube de puntos lo suficientemente amplia como para que la aplicación funcionase correctamente. Por ello, se decidió ampliar la resolución de la base de datos para comprobar si se podrían obtener mejores resultados.

BRISK FULL RESOLUTION

En este caso, se procedió a utilizar la base de datos San Felipe original, es decir, contábamos con imágenes de resoluciones de 2048x1152 y 2048x1536 píxeles. Los parámetros BRISK debieron ser modificados en este caso para que el código funcionara correctamente y se pueden ver en la Tabla 3.4.2.

MinQuality	0,0207
NumOctaves	3
MinContrast	0,0506
MatchTh	60,76
MaxRt	0,7041

Tabla 3.4.2 Tabla de parámetros empleados en BRISK

En la Figura 3.4.6, vemos como los emparejamientos son mucho más abundantes, aunque se sigue produciendo esa distinción entre imágenes de diferentes resoluciones como sucedía en el caso anterior de menor resolución.

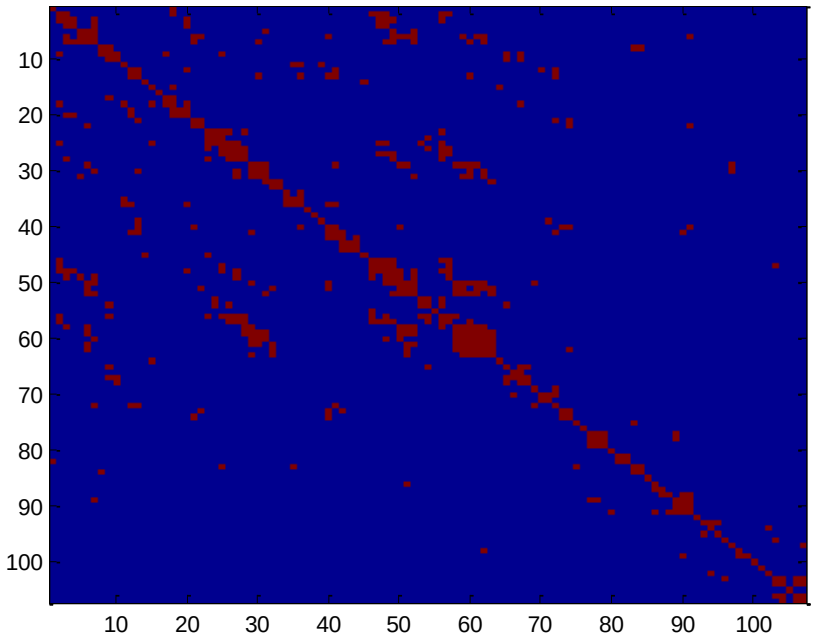

Figura 3.4.6 Emparejamiento entre imágenes utilizando BRISK

Con esta resolución, se logran emparejar todas las imágenes con alguna de la base de datos salvo 10, un valor mucho más favorable que en el caso anterior. En la Figura 3.4.7 se muestra la cantidad de emparejamientos que recibe cada imagen alcanzando algunas hasta 13 emparejamientos. De media, cada imagen logra emparejarse con un total de 4,38 imágenes.

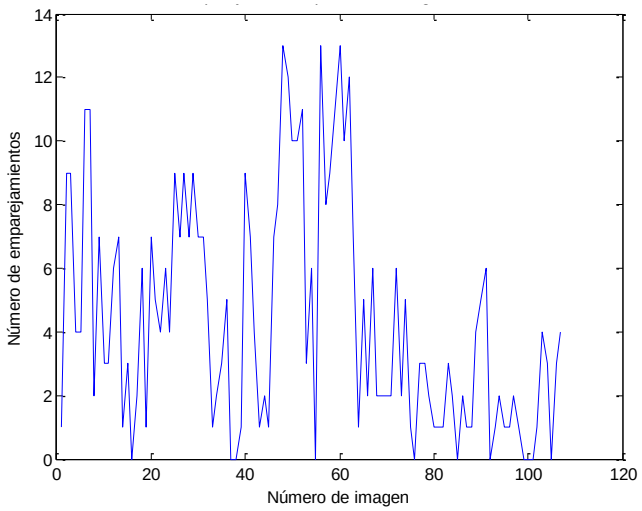

Figura 3.4.7 Número de emparejamientos para cada imagen utilizando BRISK

Realizando ahora un análisis de la cantidad de falsos positivos que se producen, Figura 3.4.8, se ve que cumple perfectamente con lo esperado, así como en términos de precisión y exactitud. Decir que el número de falsos rechazos resulta ser elevado y no se logra emparejar de una forma tan buena como con el algoritmo SIFT. Sin embargo, este es un dato que ya era de prever por la diversa literatura que habla de ello al respecto al comparar ambos algoritmos.

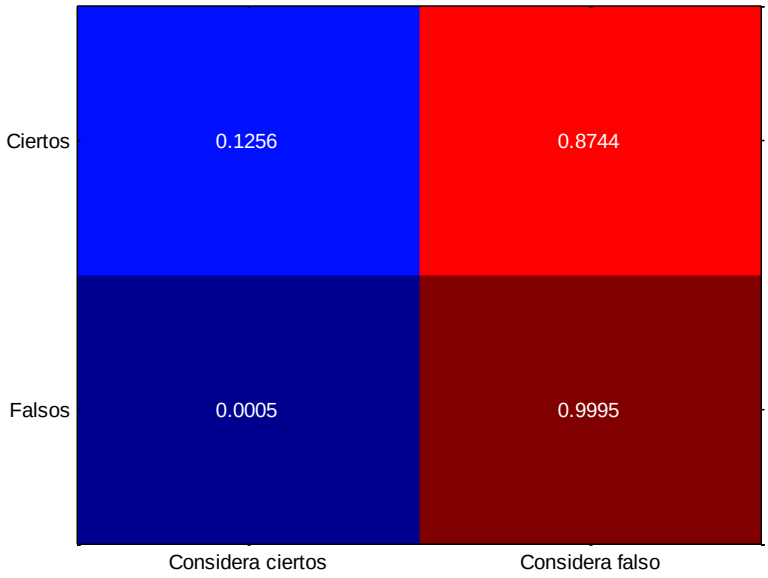

Figura 3.4.8 Matriz de confusión obtenida para el caso de BRISK de alta resolución

Accuracy	0.68
Precision	0.99

Con estos resultados, podemos validar la funcionalidad del algoritmo BRISK para proceder a su implementación en c++.

MSER y SURF

Por realizar una comparación con otros algoritmos, se analizaron también los resultados con MSER y SURF. En el Anexo B, se explican los resultados obtenidos siguiendo el mismo esquema que con BRISK. A continuación, se han añadido unas gráficas que muestran la comparación de algoritmos.

En la Figura 3.4.9, podemos ver los valores obtenidos de exactitud y precisión para cada uno de los casos analizados. Del mismo modo, en la Figura 3.4.10, se muestra cómo se producen los emparejamientos entre las imágenes de las distintas bases de datos.

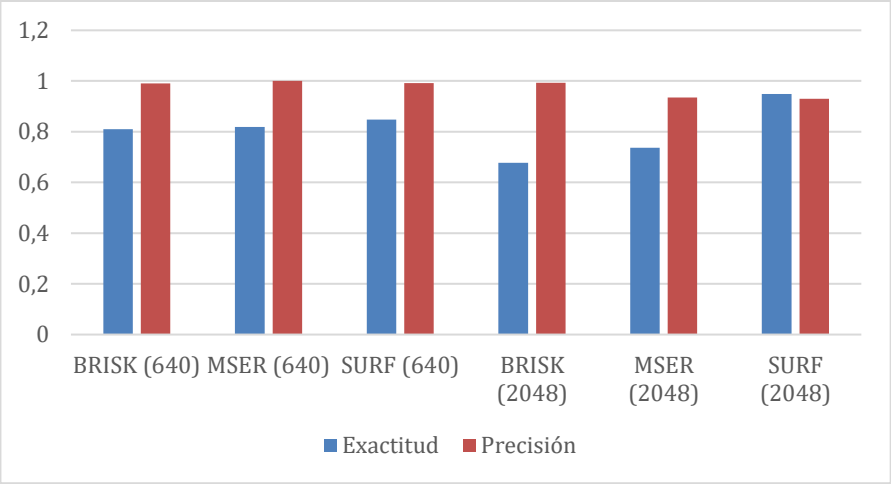

Figura 3.4.9 Parámetros de Exactitud y Precisión para los casos estudiados

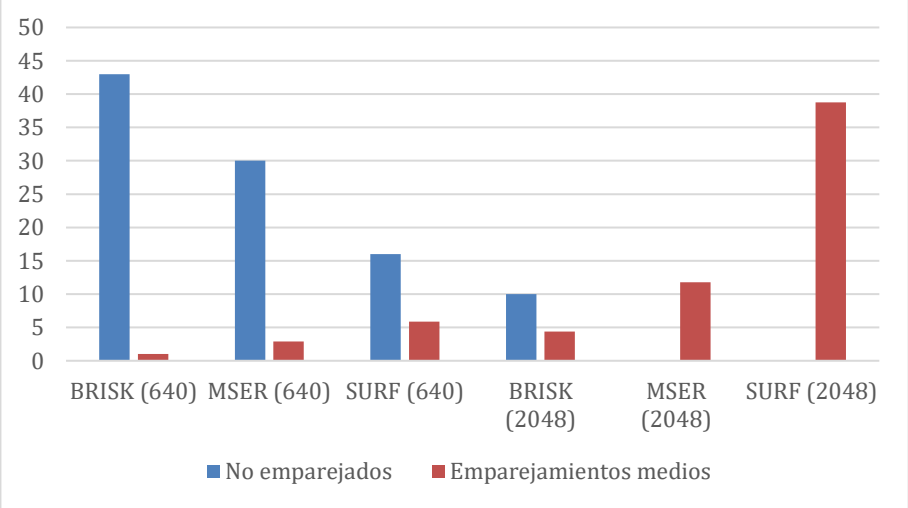

Figura 3.4.10 Cantidad de imágenes no emparejadas y emparejamientos medios para cada uno de los casos estudiados

Vemos que SURF es el algoritmo que mejor resultado da. Sin embargo, en este proyecto se ha optado por utilizar BRISK ya que es un algoritmo computacionalmente más rápido que SURF y MSER, como vimos en la sección 3.3.

4. Visual Studio

Una vez validada la funcionalidad del descriptor BRISK en Matlab, se procedió a su implementación en un lenguaje mucho más eficiente computacionalmente y que se podría trasladar de manera relativamente sencilla a una aplicación móvil. El lenguaje empleado es c++ y el IDE que se ha utilizado en este proyecto es *Visual Studio* ya que permite una fácil integración con la librería de *OpenCV* [12] de la cual hablaremos posteriormente.

La aplicación previa desarrollada por el grupo de investigación de visión por computador (CV_Lab), presentaba un desarrollo en el cual se realizaba una fotografía con el terminal móvil que, posteriormente, era enviada a un servidor remoto que realizaba los cálculos de los puntos SIFT y hallaba la localización geoespacial en el entorno de la plaza San Felipe. Para realizar esta geolocalización, el programa comparaba los *keypoints* obtenidos con una nube de puntos 3D de la plaza modelada que se mostrará en los siguientes apartados. Esta nube de puntos era generada mediante un software que extraía los emparejamientos en una base de datos de imágenes y generaba el modelado 3D de la plaza. Sin embargo, este programa sólo permitía la opción de extraer de manera automática los puntos SIFT. Para la nueva aplicación, deberemos extraer los puntos BRISK, el nuevo descriptor, de manera externa para que el programa sea capaz de generar esta nueva nube 3D de los puntos obtenidos con este descriptor. Por tanto, el siguiente objetivo del trabajo consistió en implementar en Visual Studio todo el código utilizado en Matlab: extracción de descriptores BRISK, emparejamiento, RANSAC, filtrado por ángulos y escala, filtrado por triángulos y reconstrucción. Con ello, se conseguirán extraer los puntos BRISK de la base de datos de 107 imágenes dada para poder generar la nube de puntos 3D requerida en nuestra aplicación.

La principal librería empleada en este tipo de aplicaciones es la librería de *OpenCV*, desarrollada por Intel, la cual cuenta con numerosos algoritmos de visión por computador como puede ser el algoritmo BRISK que queremos utilizar. Esta librería se cargó en el programa de Visual Studio para el uso de la aplicación. El esquema a seguir en el programa de Visual Studio es el mismo que el seguido en Matlab, sin embargo, al utilizar otras funciones, los parámetros requeridos varían y por tanto también los resultados. A continuación, se relatarán los cambios más significativos, así como los problemas presentados en la implementación en c++.

4.1 Cambios introducidos

Cambio de parámetros BRISK

Si bien en Matlab contábamos con cuatro parámetros (tres que manejábamos) para ajustar el algoritmo para obtener un mejor resultado en nuestro problema concreto, en este caso los parámetros a variar son sólo 3, y además distintos a los utilizados en Matlab:

• **Thresh**: es el umbral de detección que utiliza FAST/AGAST. En la práctica, es el que restringía el número de puntos BRISK obtenidos por el algoritmo. Podemos decir, que cuanto mayor sea este umbral, menos puntos obtendremos. A priori puede que parezca interesante obtener el mayor número de puntos posible, sin embargo, esto dificulta la precisión a la hora de realizar el emparejamiento puesto que se deben comparar con más posibles puntos. Además, incrementa el tiempo de cómputo que es uno de nuestros parámetros más importantes en nuestro problema.

• **Octaves**: el número de octavas empleado en la detección. Es similar al empleado en Matlab.

• **PatternScale**: escala aplicada al patrón utilizado para muestrear los vecinos del *keypoint*. Tras varias pruebas, se determinó que alterar este parámetro de escala empeoraba los resultados, así que se optó por dejar el valor dado por defecto.

Como veremos posteriormente, estos valores varían mucho en función del problema a abordar. En nuestro caso, se trabajó con diferentes resoluciones de la misma base de datos, puesto que a menor resolución parece claro que se reduciría el tiempo de cómputo. Al cambiar esta resolución de la base de datos, notamos que se necesitaban modificar estos parámetros sustancialmente en función de cada problema. Al igual que en la parte de Matlab, nos encontramos con el problema de no poder evaluar mediante código los buenos o malos resultados que se obtienen al variar los parámetros. Se necesita realizar un procedimiento de testeo hasta encontrar unos parámetros lo suficientemente óptimos para cada problema.

Cambio de matcher

Si bien en Matlab utilizábamos una función que emparejaba los *keypoints* con el algoritmo más adecuado en función del descriptor extraído, en Visual Studio no tenemos una función que lo realice de una forma tan directa. Por ello, se procedió a la implementación de dos tipos de *matchers* para validar los resultados.

El primero de ellos es *FLANN* (Fast Library for Approximate Nearest Neighbors) [13], que nos permite implementar un algoritmo de emparejamiento tipo árbol explicado anteriormente en la sección 2.1. Como particularidad, decir que esta función requería transformar los descriptores obtenidos anteriormente a un formato CV_32 para su correcto funcionamiento; una vez solventado este problema pudimos realizar el emparejamiento correctamente.

Otra opción para realizar el *matching*, es utilizando el algoritmo de *Fuerza Bruta (BFMatcher)*. Al tratarse de un descriptor binario, la referencia de *OpenCV* nos indica que suele tener un mejor funcionamiento el emparejamiento de fuerza bruta al utilizar la distancia de Hamming como valor. También, esta función permite realizar el emparejamiento de tal forma que sólo empareja aquellos descriptores en los cuales la distancia de Hamming del descriptor i-th respecto del de j-th sea la menor y viceversa, es decir, también compara la distancia del descriptor j-th respecto de la i-th. Esta forma de realizar el emparejamiento mediante chequeo cruzado, mejoraba en gran parte los resultados que se obtenían al emparejar.

Finalmente, se utilizó el emparejamiento con *BFMatcher*, puesto que, al realizar diversas pruebas, se vio que se lograban mejores resultados que con la librería *FLANN*, aunque ésta fuera más veloz a la hora de realizar los cálculos del emparejamiento en la nube 3D.

La forma de guardar los *matches* correctos también varía respecto a Matlab, puesto que trabaja con una clase directamente creada para ello. A partir de ahora, para realizar el filtrado, se hará con una máscara en la que iremos guardando, o no, los índices de los *matches* que nos interesen o descartemos con los filtrados posteriores.

RANSAC

Después del emparejamiento, se realiza un filtrado de *outliers* con el algoritmo de RANSAC, que, si bien no sigue el mismo formato que Matlab, tampoco tuvo demasiadas complicaciones a la hora de implementarse. Destacar que el umbral utilizado es distinto al del código en Matlab y que esta función consta de un parámetro para ajustar el número de iteraciones máximo de ejecución del algoritmo, lo cual para limitar el tiempo de cómputo resulta muy útil, ya que RANSAC es un algoritmo iterativo que puede llegar a ser muy costoso en tiempo.

Filtrado de ángulos y escalas

El filtrado de ángulos y escala siguió la estela del empleado en Matlab. Sin embargo, se debieron introducir algunos cambios para que funcionase con una eficacia similar.

El ángulo que nos provee el algoritmo de BRISK de OpenCV no se encuentra entre el mismo rango de valores que en Matlab, por lo que, para que nuestro código de c++ siga el mismo patrón que en Matlab, se procedió a un ajuste de estos ángulos antes de realizar el cálculo explicado en la sección 2.2.3. El rango en el que se encuentra los valores de los ángulos de OpenCV es entre 0 y 2π, mientras que en Matlab estaba entre $-\pi$ y π.

En cuanto a las escalas, nos percatamos de que el umbral impuesto para realizar el filtrado resultaba demasiado restrictivo para los valores que proporcionaban los descriptores BRISK de OpenCV, por lo que se aumentó ese rango.

Otro problema que se encontró en esta parte de la implementación de Matlab a c++ es que en Visual Studio no disponíamos de una función que nos diese el valor de la mediana de una forma directa, por lo que se tuvo que codificar una función propia para ello.

Filtrado por triángulos

De nuevo, la implementación resulta ser distinta al utilizar unas otras librerías. Uno de los problemas encontrado en esta parte del código fue la falta de una función que devuelva las coordenadas del baricentro de los triángulos. Esta función, tuvo

que ser implementada. También, se debió ajustar el rango de los valores de los ángulos dados por los descriptores como en el filtrado anterior.

Reconstrucción

Esta parte del algoritmo, al igual que en Matlab, consistía en realizar una matriz de homografía con los puntos finales que nos permitiera recuperar puntos buenos perdidos por los filtrados estableciendo un umbral en función de la resolución de la base de datos. Los problemas encontrados aquí fueron dados por el cambio de variables en Visual Studio, puesto que los *keypoints* no se tratan como variables más comunes como pueden ser 'float' o 'double', sino como 'Point2f'.

4.2 Resultados obtenidos

Una vez implementado el programa, se procedió a la extracción de los *matches* entre las imágenes de la base de datos de la plaza San Felipe. Con el fin de evaluar con qué base de datos deberíamos trabajar en la aplicación móvil de una manera más eficiente, se extrajo una tabla de tiempos para las distintas resoluciones dadas. A diferencia de Matlab, no tenemos una matriz con la que podamos evaluar, aunque sea parcialmente, los emparejamientos correctos. Por ello, una vez se encontraron unos valores óptimos para cada resolución, se vieron todas las imágenes que el programa emparejaba de manera manual para verificar el correcto funcionamiento del algoritmo. Se variaron los parámetros nombrados anteriormente que no estaban en Matlab, así como algunos umbrales correspondientes tanto a los filtrados como a la reconstrucción. En la Tabla 4.2.1, se pueden observar los valores que mejor resultado han dado para cada uno de los casos estudiados.

	160X90	**320X180P**	**1280X720**	**2048X1152**
UMBRAL_HOMOGRAFIA	4	3	3	3
UMBRAL_ANGULOS	π/4	π/4	π/4	π/4
UMBRAL_ESCALAS	2.5	2.5	2.5	2.5
UMBRAL_TRIANGULOS	π/4	π/4	π/4	π/4
UMBRAL_RECONSTRUCCIÓN	3	1.9	1.5	1.5
THRESHL (BRISK)	1	12	115	115
OCTAVES (BRISK)	1	7	7	7
PATTERNSCALE (BRISK)	1	1	1	1
BEST MATCHES (RANSAC)	100	100	100	100
ITERACIONES (RANSAC)	2000	2000	2000	2000

Tabla 4.2.1 Valores de los parámetros más relevantes para cada base de datos

Si bien para las resoluciones más altas, el emparejamiento resultaba correcto, para las resoluciones de menor resolución se producían algunos fallos en el emparejamiento a pesar de los filtrados. Sin embargo, estas resoluciones nos permiten trabajar a unos tiempos mucho más veloces para una aplicación en tiempo real. A continuación, en la Tabla 4.2.2, se muestra el tiempo de cómputo en función de la resolución, así como el tiempo dedicado a cada parte del algoritmo. En la Figura 4.2.1, se muestran estos datos de forma gráfica para poder observarlos de una manera más visual.

	640X360	1280X720	2048X1152
TIEMPO BRISK	0,3366	0,4237	0,6762
TIEMPO MATCHING	0,0059	0,1198	0,5003
TIEMPO RANSAC	0,0976	0,0834	0,1342
TIEMPO FILTRADO	0,0393	0,1015	0,349
TIEMPO TRIANGULOS	0,0018	0,0086	0,0432
TIEMPO RECONSTRUCCION	0,0026	0,0068	0,0098
TIEMPO TOTAL	**0,4838**	**0,7438**	**1,7127**

Tabla 4.2.2 Tiempos de procesados en cada uno de los bloques de la aplicación

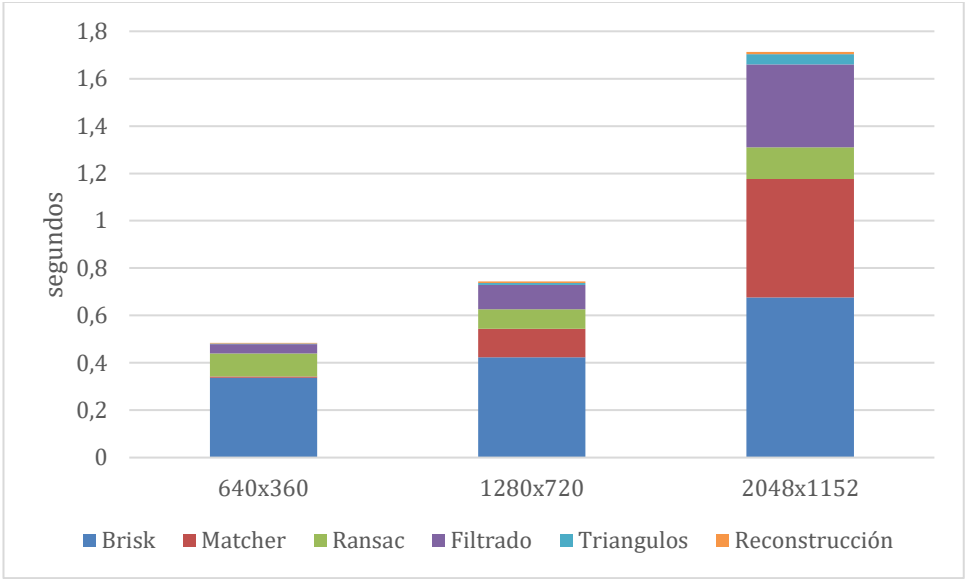

Figura 4.2.1 Tiempos de procesado para cada uno de los bloques de la aplicación

5. Emparejamiento con el modelo del mundo

Una vez se han extraído los puntos BRISK más relevantes, se procedió a la reconstrucción de la nube de puntos 3D a partir de los keypoints obtenidos. Para ello se utilizó la aplicación de interfaz gráfica *Visual SFM* que permite realizar esta reconstrucción para una base de datos dada. El programa es capaz de calcular de manera automática los puntos SIFT de un conjunto de imágenes y realizar el emparejamiento entre ellas para así obtener la nube de puntos 3D como se muestra en la Figura 5.2.1.

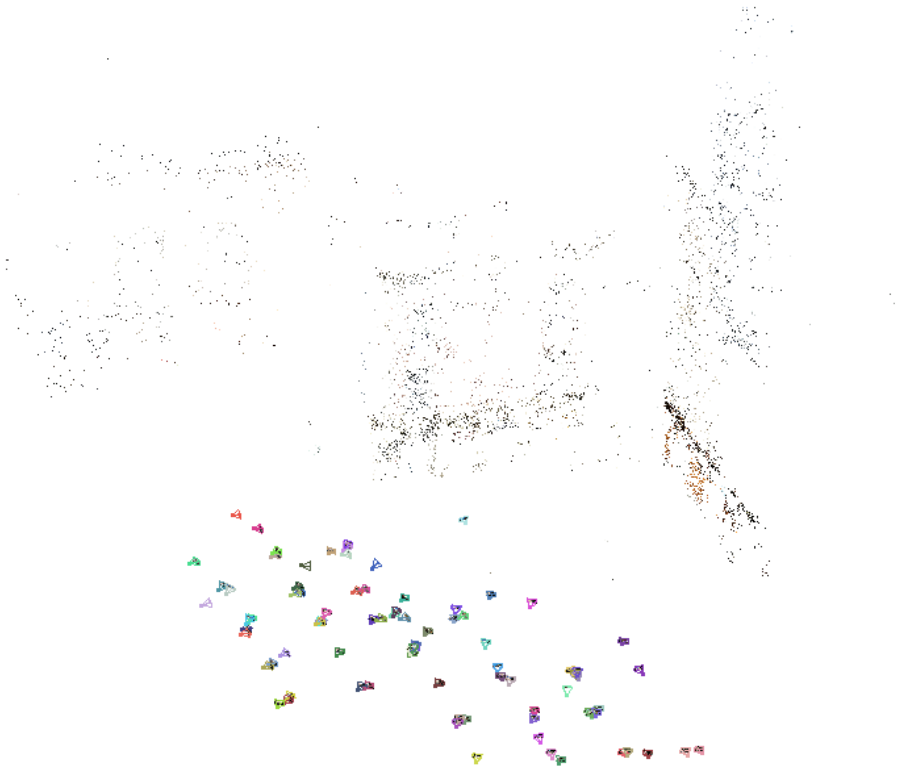

Figura 5.2.1 Nube de puntos 3D de la plaza San Felipe obtenida con SIFT

Además, ofrece una visualización en forma de árbol mostrando como se emparejan las imágenes de la base de datos (Figura 5.2.2) y una matriz de emparejamientos (Figura 5.2.3) similar a la que mostrábamos en Matlab. En ella, se puede apreciar claramente como los emparejamientos entre las imágenes tomadas con distinta relación de aspecto, no logran un emparejamiento tan eficaz como aquellas que tienen la misma relación de aspecto (16:9 hasta la imagen 75 aproximadamente y 4:3 a partir de entonces).

Figura 5.2.2 Árbol de emparejamientos de la base de datos San Felipe utilizando SIFT

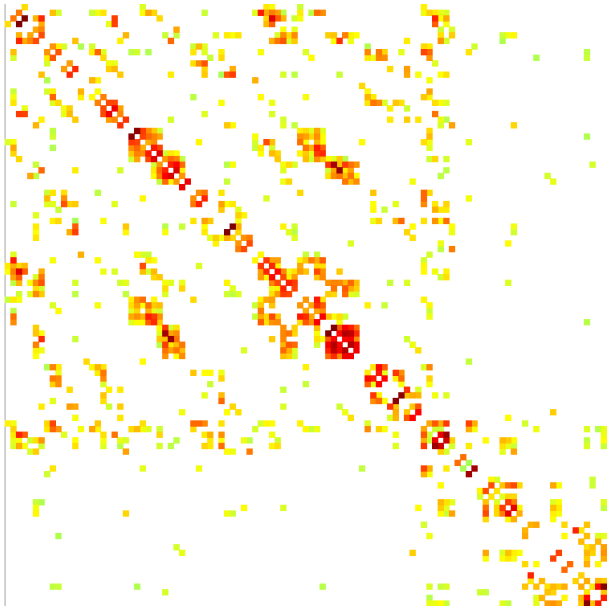

Figura 5.2.3 Matriz de emparejamientos de la base de datos San Felipe

La aplicación original del proyecto anterior obtenía de esta manera la nube de puntos, sin embargo, para la nueva aplicación (más veloz) se pretendió utilizar los keypoints ofrecidos por BRISK, por lo que era preciso cargar esta información al programa de Visual SFM. Se creó un archivo de texto para cada imagen indicando los keypoints obtenidos. Estos archivos debían tener un formato concreto que nos ofrecía en la documentación online de Visual SFM. Posteriormente, debían ser

codificadas en el formato binario de *Lowe's ASCII* y guardadas con extensión '.sift' para que fueran entendidas por la aplicación. De igual modo, se requería de otro fichero de *matches*, en el que se indicaba que puntos se emparejaban entre las imágenes de la base de datos. Una vez obtenidos estos archivos de Visual Studio, se empleó la herramienta de Visual SFM para generar la nube de puntos en 3D.

La primera prueba que se hizo fue con los puntos extraídos de las imágenes a una resolución de 720p. Aunque el resultado fue bueno, pues se emparejaban las imágenes de la base de datos correctamente, los puntos generados por el programa (Figura 5.2.4) parecían muy escasos si se comparaban con la nube de puntos SIFT.

Figura 5.2.4 Nube de puntos 3D de la plaza San Felipe en resolucion 720p obtenida con BRISK

Por ello se decidió crear la reconstrucción 3D de la plaza San Felipe a partir de los puntos BRISK de la base de datos con la resolución original. El resultado, a pesar de no ser tan bueno como el dado en el trabajo previo, sí que parece ser suficiente para realizar la localización 3D de una nueva imagen y situar así la torre. A continuación, en la Figura 5.2.5, se muestra la reconstrucción generada, donde se pueden diferenciar fácilmente varias fachadas clave de la plaza.

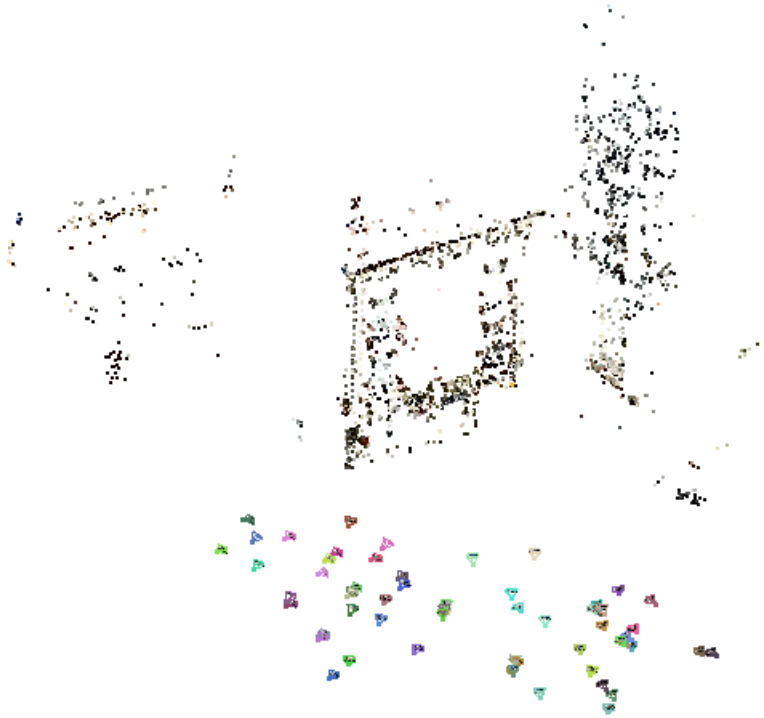

Figura 5.2.5 Nube de puntos 3D de la plaza San Felipe en resolución original obtenida con BRISK

Con esta nube de puntos, podemos generar una relación entre puntos 2D y 3D gracias a la función *solvePNP*[1] de OpenCV. A continuación, se va a explicar el funcionamiento de dicha función. En OpenCV, la función *solvePNP* viene dada por la siguiente expresión:

$$solvePnP(objectPoints, imagePoints, cameraMatrix, distCoeffs, rvec, tvec, useExtrinsicGuess, flags);$$

Donde *rvec* y *tvec* hacen referencia, respectivamente, a los vectores de rotación y traslación en la conversión 3D-2D. Esta función, requiere como parámetros la traducción entre puntos 3D y 2D que hemos podido realizar gracias a VisualSFM. En *objectPoints*, introduciríamos estos puntos 3D y en *imagePoints*, sus correspondencias 2D de las imágenes tomadas de la base de datos.

Esta función, requiere también una matriz de proyección (7) que modela los parámetros intrínsecos de la cámara. Como no se dispone de esta información a priori, pues cada usuario podría utilizar una cámara diferente en función del terminal móvil, se pueden aproximar estos parámetros de una forma suficientemente precisa en función de la resolución de la imagen a tomar (6).

$$\begin{bmatrix} fx & 0 & cx \\ 0 & fy & cy \\ 0 & 0 & 1 \end{bmatrix} \quad (7)$$

$$\begin{cases} fx = anchura \\ fy = anchura \\ cx = \frac{anchura}{2} \\ cy = \frac{altura}{2} \end{cases} \quad (6)$$

[1] http://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html

Los parámetros de distorsión de la cámara, definidos por *distCoeffs*, los introduciríamos como nulos al no saber con qué cámara se va a trabajar. En cuanto al parámetro *useExtrinsicGuess*, indicaría la opción de añadirles a los vectores rotación y traslación (*rvec* y *tvec*) unos valores iniciales, aunque en este caso concreto no se incorporarían dichos valores. Por último, el parámetro *flags*, nos permite elegir qué tipo de algoritmo queremos utilizar para realizar el cálculo de los vectores de rotación y traslación.

Con estos vectores podemos realizar una conversión 3D-2D. Sin embargo, nuestro problema requiere la operación contraria, conversión de 2D a 3D, por lo que necesitamos invertir estos resultados obtenidos.

La librería *OpenGL* (Open Graphics Library) define un estándar en aplicaciones que trabajan con gráficos 2D y 3D. Definida la matriz de proyección, que utilizaremos en la traducción 2D a 3D, por OpenGL (8), es necesario obtener esos valores.

$$\begin{bmatrix} r11 & r12 & r13 & t1 \\ -r21 & -r22 & -r23 & t2 \\ -r31 & -r32 & -r33 & t3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (8)$$

Utilizaremos el método de Rodrigues [14], como se indica en *OpenCV* (9) para convertir el vector de rotación, *rvec* en una matriz, *rmat*, interpretable para obtener los valores de la matriz de proyección (10). Los valores *t1*, *t2* y t3 (12), vendrán definidos por los 3 primeros elementos del vector *posvec*, que obtendremos siguiendo la ecuación (11).

$$Rodrigues(rvec, rmat) \quad (9)$$

$$\begin{cases} r11 = rmat(0,0) \\ r12 = rmat(0,1) \\ r13 = rmat(0,2) \end{cases} \quad \begin{cases} r21 = rmat(1,0) \\ r22 = rmat(1,1) \\ r23 = rmat(1,2) \end{cases} \quad \begin{cases} r31 = rmat(2,0) \\ r32 = rmat(2,1) \\ r33 = rmat(2,2) \end{cases} \quad (10)$$

$$posvec = -rmat.t * tvec \quad (11)$$

$$\begin{cases} t1 = posvec(0) \\ t2 = posvec(1) \\ t3 = posvec(2) \end{cases} \quad (12)$$

Obtenidos los valores de la matriz de proyección, ya podríamos realizar la conversión de puntos 2D a 3D. Por tanto, para una nueva imagen, se extraerían sus emparejamientos en 2D y posteriormente se realizaría su traducción al mundo 3D. Gracias a esta información, nuestro sistema ya sería capaz de detectar la posición de la cámara desde la que se ha tomado una imagen nueva y así, geolocalizarla en el espacio 3D superponiendo la imagen de la torre en la foto original tomada por el usuario.

6. Conclusiones

Como conclusión final, podemos decir que se ha validado e implementado la resolución del problema de emparejamiento de imágenes con un descriptor más veloz (BRISK), asegurando la robustez del sistema añadiendo sistemas de filtrados y reconstrucción de puntos. También se ha creado la nube de puntos 3D correspondiente. Como crítica, podríamos decir que no se ha llegado a la implementación en la aplicación móvil por falta de tiempo en los plazos establecidos, analizando así los tiempos de ejecución reales para compararlos con el proyecto anterior.

Como líneas futuras, podríamos contemplar la implementación de dicha aplicación, así como la verificación en otro tipo de entornos.

Con este trabajo, se han aprendido técnicas de visión por computador sobre todo relacionadas con el emparejamiento de imágenes. También se ha trabajado con programas nuevos para mí como *Visual Studio* o *VisualSFM* y se ha profundizado en lenguajes de programación poco tratados hasta la fecha como c++.

A nivel personal puedo decir que me siento satisfecho con lo aprendido en este proyecto, de haber explorado una rama de conocimiento nueva para mí y de la forma en la que se llevó a cabo.

Bibliografía

[1] C. Orrite, M. Rodríguez, M. A. Varona, E. Estopiñán, and M. A. Montañés, "Augmented Reality for Hidden or Lost Cultural Heritage," 2016, (enviado para su revisión).

[2] S. Keypoints and D. G. Lowe, "Distinctive Image Features from," *Int. J. Comput. Vis.*, vol. 60, no. 2, pp. 91–110, 2004.

[3] M. a Fischler and R. C. Bolles, "Random Sample Consensus: A Paradigm for Model Fitting with," *Commun. ACM*, vol. 24, pp. 381–395, 1981.

[4] X. Guorong, C. Peiqi, and W. Minhui, "Bhattacharyya distance feature selection," *Proc. 13th Int. Conf. Pattern Recognit.*, vol. 2, pp. 195–199 vol.2, 1996.

[5] H. Bay, T. Tuytelaars, L. Van Gool, A. Leonardis, H. Bischof, and A. Pinz, "SURF: Speeded Up Robust Features," vol. 3951, pp. 404–417, 2006.

[6] S. Leutenegger, M. Chli, and R. Y. Siegwart, "BRISK: Binary Robust invariant scalable keypoints," *Proc. IEEE Int. Conf. Comput. Vis.*, no. November, pp. 2548–2555, 2011.

[7] C. Harris and M. Stephens, "A Combined Corner and Edge Detector," *Procedings Alvey Vis. Conf. 1988*, pp. 147–151, 1988.

[8] M. Donoser, H. Riemenschneider, and H. Bischof, "Shape guided Maximally Stable Extremal Region (MSER) tracking," *Proc. - Int. Conf. Pattern Recognit.*, pp. 1800–1803, 2010.

[9] A. Satnik, R. Hudec, P. Kamencay, J. Hlubik, and M. Benco, "A comparison of key-point descriptors for the stereo matching algorithm," *2016 26th Int. Conf. Radioelektronika*, pp. 292–295, 2016.

[10] C. Schaeffer, "A Comparison of Keypoint Descriptors in the Context of Pedestrian Detection: FREAK vs. SURF vs. BRISK," *Cs229.Stanford.Edu*, pp. 1–5, 2012.

[11] J. Hartmann, J. H. Klussendorff, and E. Maehle, "A comparison of feature descriptors for visual SLAM," *2013 Eur. Conf. Mob. Robot. ECMR 2013 - Conf. Proc.*, pp. 56–61, 2013.

[12] G. & A. Kaebler-O'Reilly, "Computer vision with the OpenCV library." 2008.

[13] D. G. Muja, M., & Lowe, "Flann, fast library for approximate nearest neighbors." .

[14] O. Rodrigues, "Des lois géometriques qui regissent les déplacements d' un systéme solide dans l' espace, et de la variation des coordonnées provenant de ces déplacement considérées indépendent des causes qui peuvent les produire." pp. 380–440, 1840.

[15] M. Calonder, V. Lepetit, C. Strecha, and P. Fua, "BRIEF: Binary robust independent elementary features," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 6314 LNCS, no. PART 4, pp. 778–792, 2010.

[16] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "ORB: An efficient alternative to SIFT or SURF," *Proc. IEEE Int. Conf. Comput. Vis.*, pp. 2564–2571, 2011.

Anexo A

A continuación, se va a realizar un breve resumen de distintos descriptores que se mencionan a lo largo de la memoria.

BRISK

Binary Robust Invariant Scalable Keypoints (BRISK) [6] es un método para la obtención de puntos de interés invariante a escala y rotación. La extracción de keypoints se adquieren del método AGAST, siendo éste una mejora del FAST. Utiliza una máscara de los 9 píxeles consecutivos para realizar esta detección, analizando si estas regiones son más brillantes o menos que el pixel central y devolviendo un valor binario de 512 bits.

BRIEF

Binary Robust Independet Elementary Features (BRIEF) [15] es un método binario. Utiliza un patrón de muestreo de 128, 256 o 512 bits. No es invariante a escala y rotación. Presenta el menor coste computacional de los descriptores binarios estudiados en este trabajo.

ORB

Oriented FAST and Rotated BRIEF (ORB) [16] calcula la orientación local y mejora la falta de invariancia rotacional de BRIEF. Utiliza 256 comparaciones de intensidad que se construye mediante *machine learning* a diferencia de BRIEF. Calcula una orientación local mediante el uso de un centroide de intensidad, que es un promedio ponderado de las intensidades de píxeles asumiendo no ser coincidente con el centroide del *keypoint*. La orientación es el vector entre la ubicación de la entidad y el centroide. A pesar de ser invariante a rotación, no lo es respecto a escala.

HARRIS

Harris Corner Detector (HARRIS) [7] presenta un método en el que, para cada píxel en una imagen, se forma una matriz que está relacionada con la función de autocorrelación. La matriz captura las curvaturas principales de la imagen. No son invariantes a escala y rotación.

HoG

Histogram of Oriented Gradients (HoG) es un descriptor que utiliza la información de la intensidad basada en la orientación de sus gradientes. Transforma el espacio 2D de la imagen en 1D (histograma) dividiendo la imagen en varios bloques para calcularlo.

MSER

Maximally stable extremal regions (MSER) [8] es un descriptor que busca aquellas regiones que sean estables en escala. El algoritmo convierte la imagen en binario (blanco y negro) con diferentes umbrales determinando los blobs en las regiones más estables.

SIFT

Scale invariant Feature Transform (SIFT) [2] es un descriptor que realiza la localización de *keypoints* mediante diferencias gaussianas (DoG). Cada descriptor está formado por 128 valores. Este método es invariante tanto a escala como a rotación.

SURF

Speeded Up Robust Features (SURF) [5] es la versión rápida del descriptor SIFT. Utiliza las ideas de SIFT, pero más simplificadas. Cada vector está definido por 64 valores (la mitad que en SIFT). También resulta ser invariante tanto a rotación como a escala.

	Binario	*Rotación Invariante*	*Escala Invariante*
BRISK	Sí	Sí	Sí
BRIEF	Sí	No	No
ORB	Sí	Sí	No
HARRIS	No	No	No
HoG	No	No	Sí
MSER	No	Si	Sí
SIFT	No	Sí	Sí
SURF	No	Sí	Sí

Tabla A.1 Comparación de descriptores

Normalmente, los descriptores realizan cálculos de gradientes muy complejos y costosos computacionalmente, lo que provoca que el tiempo de ejecución aumente sustancialmente. Los descriptores binarios, hacen uso de una simple comparación de píxeles cuyo resultado es binario. La comparación de píxeles es mucho más eficiente que las operaciones con gradientes y reduce el tiempo de cómputo de manera drástica. Además, el emparejamiento posterior en descriptores binarios, se suele realizar haciendo uso de la distancia de Hamming que es mucho más rápida que computar la métrica L2 utilizada normalmente en descriptores no binarios. Otra de las ventajas de trabajar con descriptores binarios es que requieren de menos memoria en el dispositivo. Un solo gradiente de un descriptor común, requiere de 64 o 128 valores tipo float, mientras que los binarios tan sólo necesitan 512 bits, lo que supone una reducción de un factor entre 4 y 8.

Por estos motivos, resulta lógico pensar en que el cambio de descriptor sea por uno de tipo binario. Entre los descriptores binarios mostrados en la tabla comparativa de descriptores, Tabla A.1, BRISK nos ofrece las mejores características en cuanto a rotación y escala, pues en ambos resulta ser invariante.

Anexo B

MSER (640)

Otro algoritmo con el que se probó los emparejamientos de la plaza San Felipe fue MSER. Con la base de datos de 640, vemos que la matriz de emparejamiento, Figura B.1, mejora con respecto a la de BRISK. Logra tener un total de 30 imágenes sin emparejar, Figura B.2, y una media de 2,9 imágenes emparejadas.

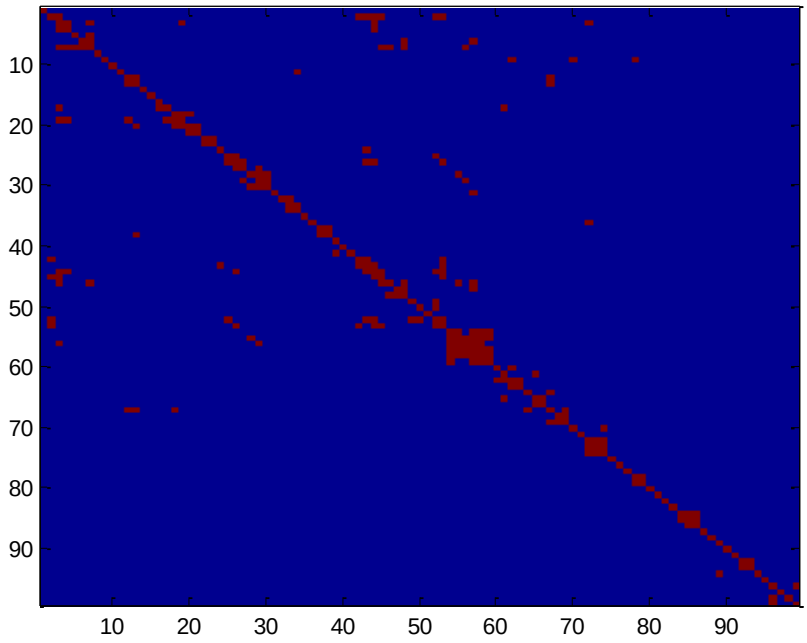

Figura B.1 Emparejamientos entre imágenes utilizando MSER

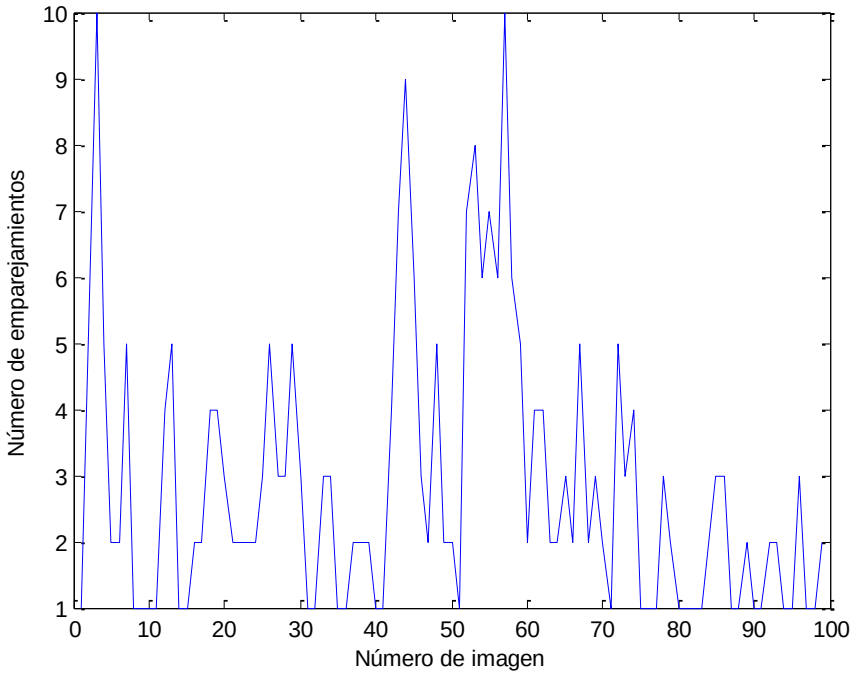

Figura B.2 Número de emparejamientos para cada imagen utilizando MSER

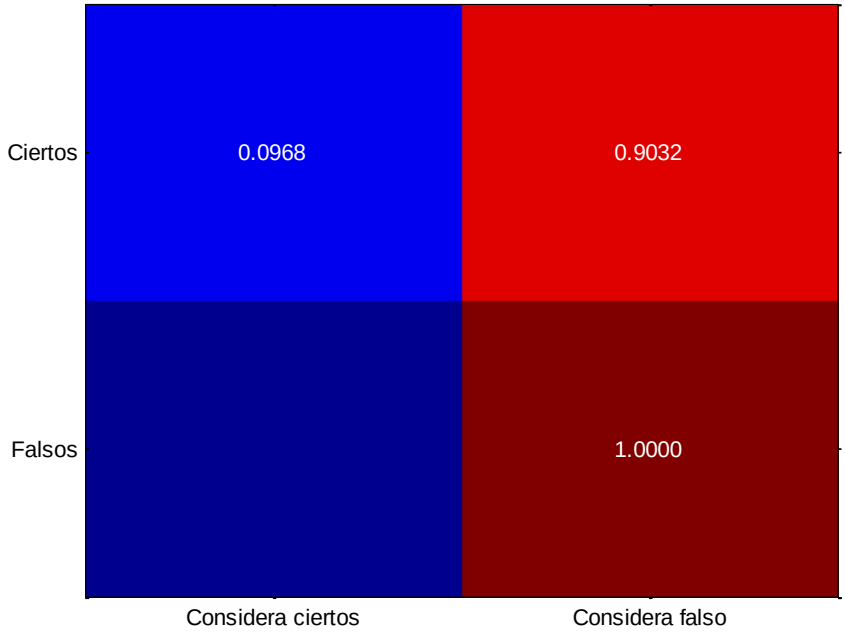

Figura B.3 Matriz de confusión utilizando MSER

Accuracy	0.82
Precision	1

Además, también cumple el no tener apenas falsos positivos, reduciendo en este caso este valor a 0 falsos positivos como se muestra en la Figura B.3. Por tanto, no es de extrañar que tantos los valores de precisión y exactitud sean también positivos.

MSER FULL RESOLUTION

Si vemos el mismo caso, MSER, pero con más resolución, vemos que estos resultados mejoran. Tanto en la Figura B.4 como en la Figura B.5 muestran que ninguna imagen se queda sin emparejar y se produce un total de 11,79 emparejamientos medios por imagen.

Tampoco se producen apenas falsos positivos, y obtenemos unos buenos valores de exactitud y precisión, 0,74 y 0,94 respectivamente, como se muestra en la Figura B.6 .

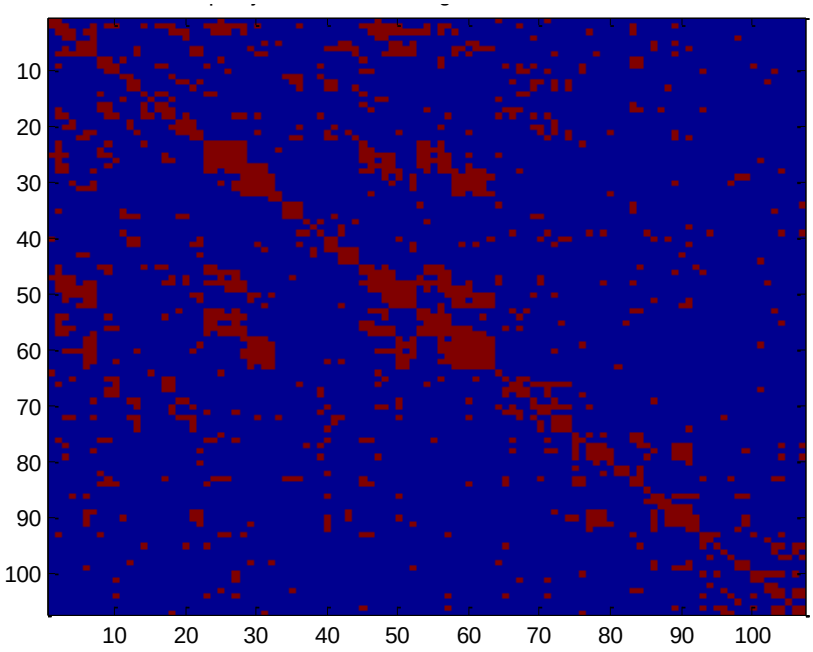

Figura B.4 Emparejamiento entre imágenes utilizando MSER

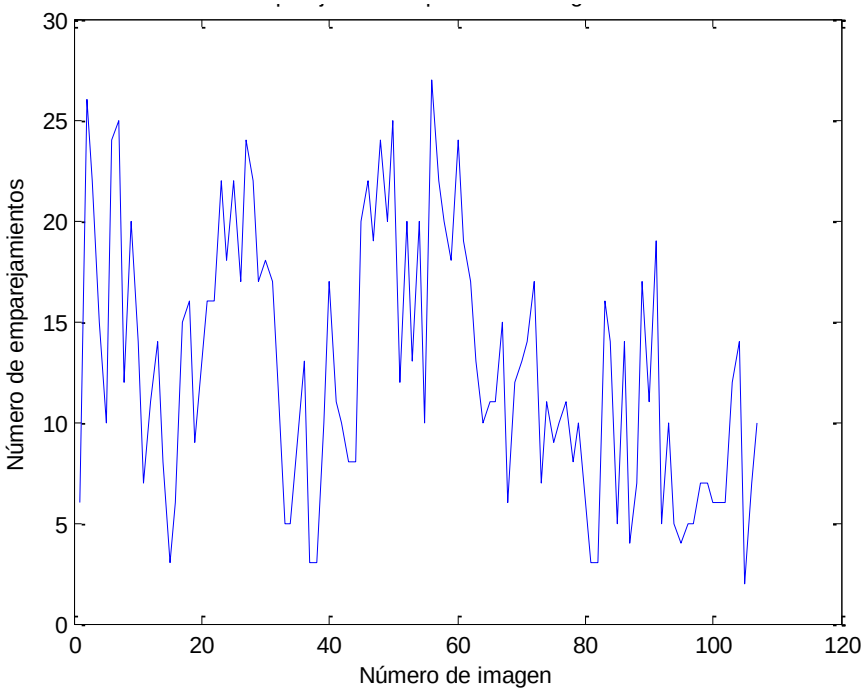

Figura B.5 Número de emparejamientos para cada imagen utilizando MSER

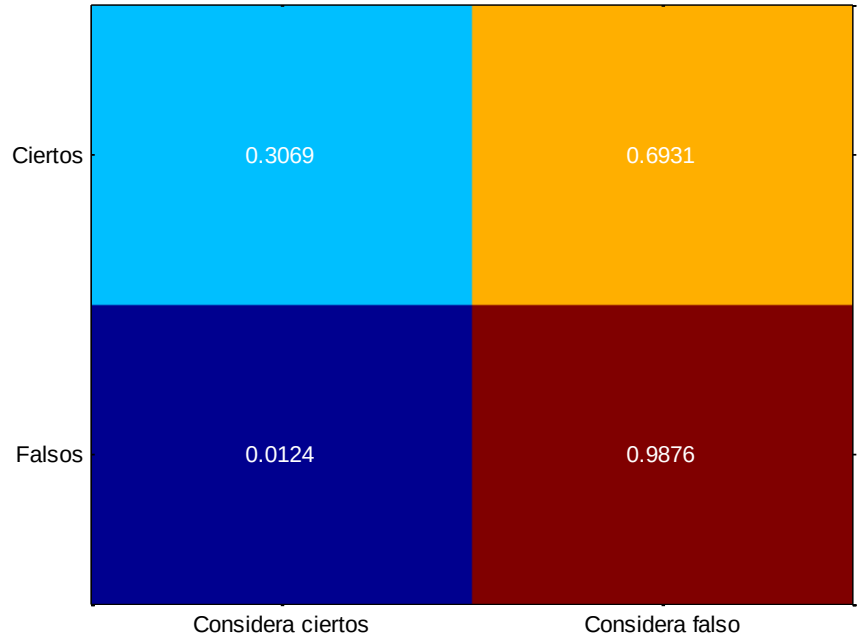

Figura B.6 Matriz de confusión utilizando MSER

Accuracy	0,74
Precision	0,94

SURF (640)

SURF es la versión rápida del algoritmo SIFT, por lo que no era de extrañar que presentase también buenos resultados para este problema.

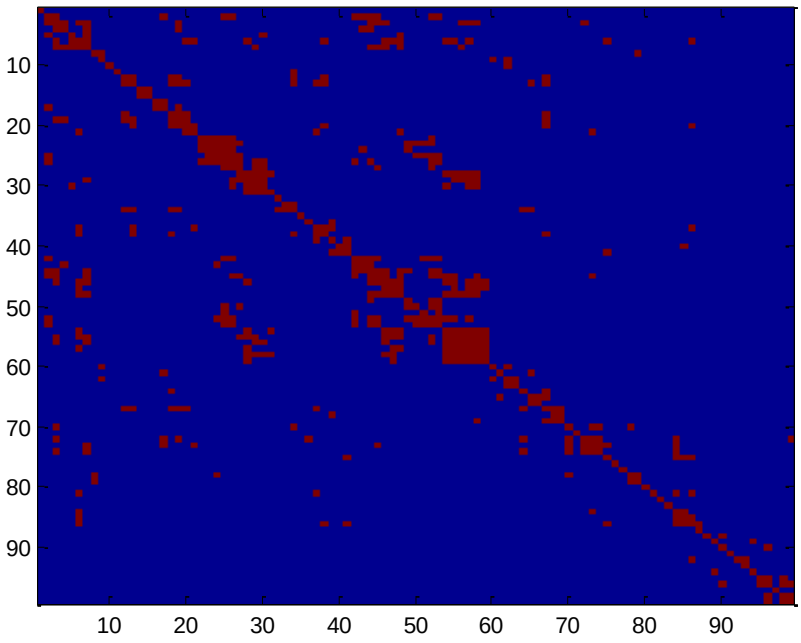

Figura B.7 Emparejamientos entre imágenes utilizando SURF

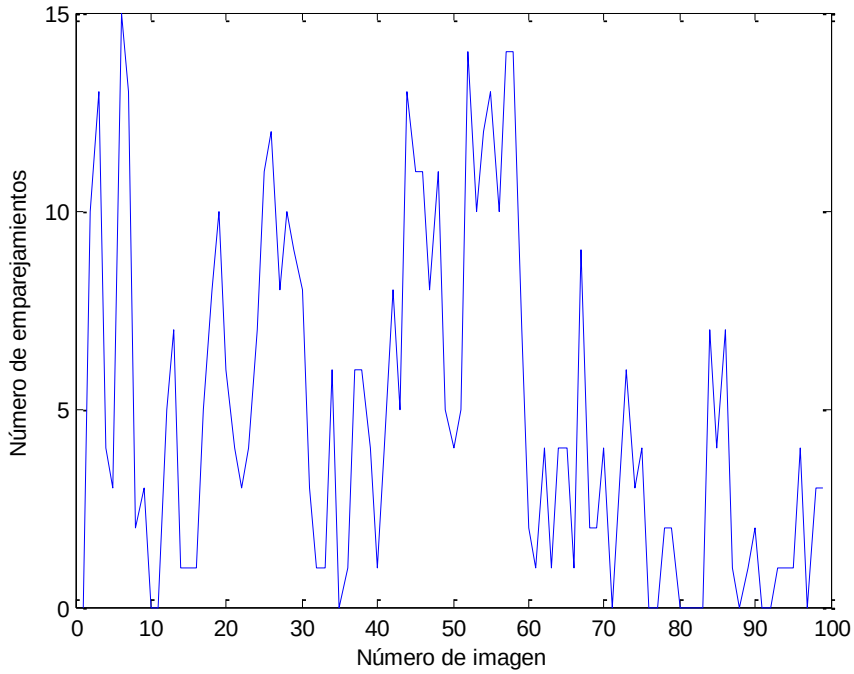

Figura B.8 Número de emparejamientos para cada imagen utilizando SURF

La Figura B.7 y Figura B.8 nos muestran el buen emparejamiento que se produce con la base de datos de resolución 640. Cada imagen se empareja de media con 5,85 imágenes y tan sólo encontramos sin emparejar 16 imágenes.

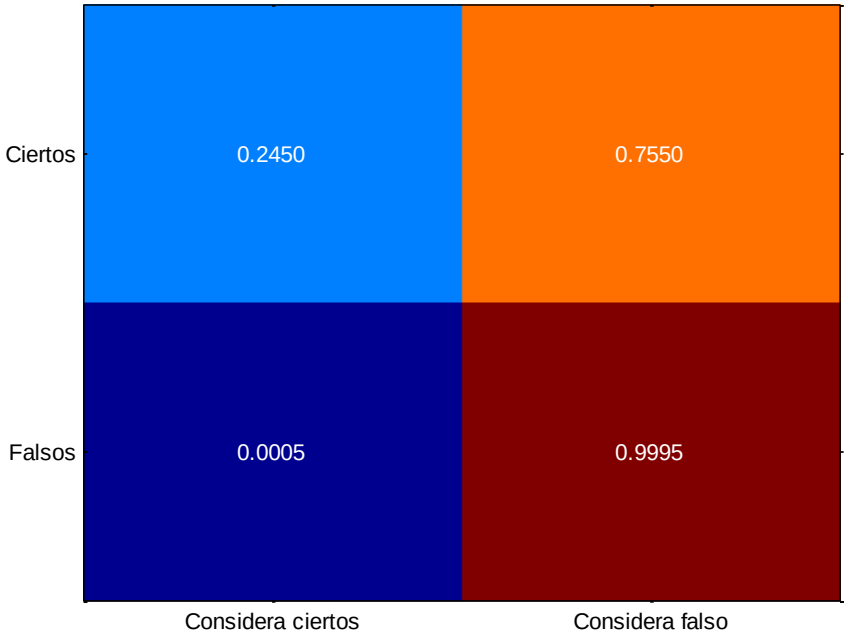

Figura B.9 Matriz de confusión utilizando SURF

Accuracy	0.85
Precision	0.99

También, la precisión y exactitud cumplen con los objetivos marcados. Como se puede ver en la Figura B.9, tan sólo un 0,05% de los emparejamientos producidos corresponden a falsos positivos.

SURF (2048)

Si mejoramos la resolución de la base de datos, no es de extrañar que el resultado obtenido, ya de por sí bueno, mejore. En la Figura B.10, vemos como la matriz de emparejamientos resulta bastante similar a la dada por SIFT, mientras que en la Figura B.11, se observa un número muy elevado de emparejamientos por imagen.

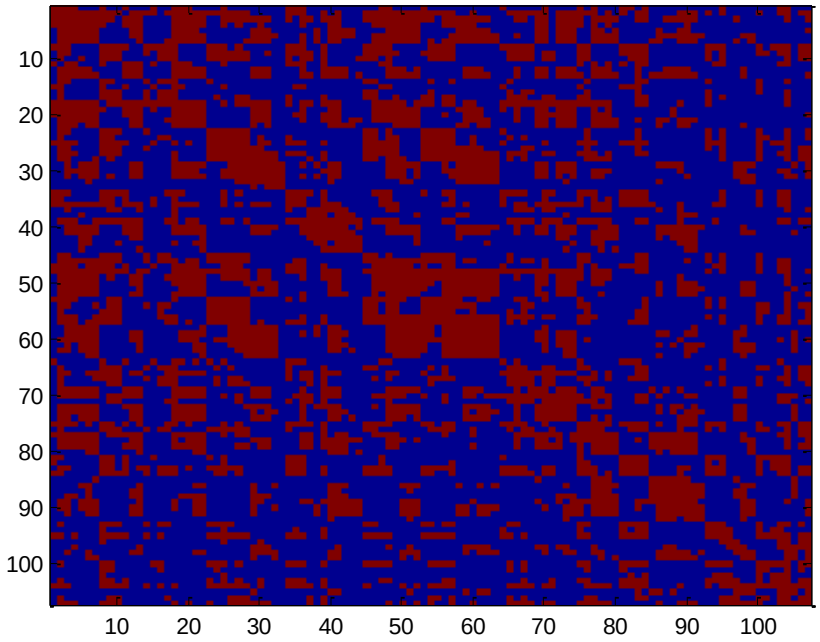

Figura B.10 Emparejamientos entre imágenes utilizando SURF

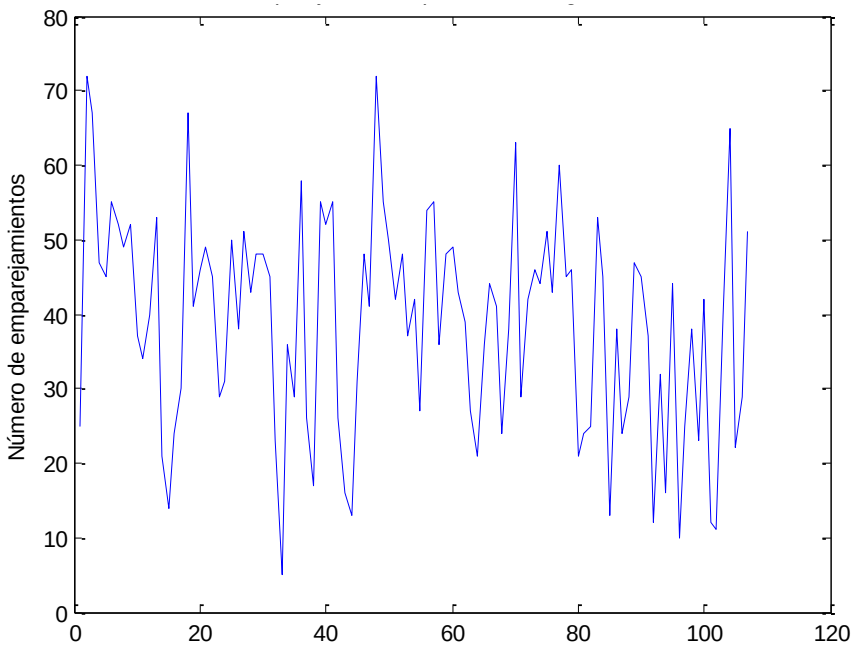

Figura B.11 Número de emparejamientos para cada imagen utilizando SURF

Este algoritmo logra una media de 38,77 imágenes emparejadas por imagen dejando sin emparejar 0 imágenes de la base de datos.

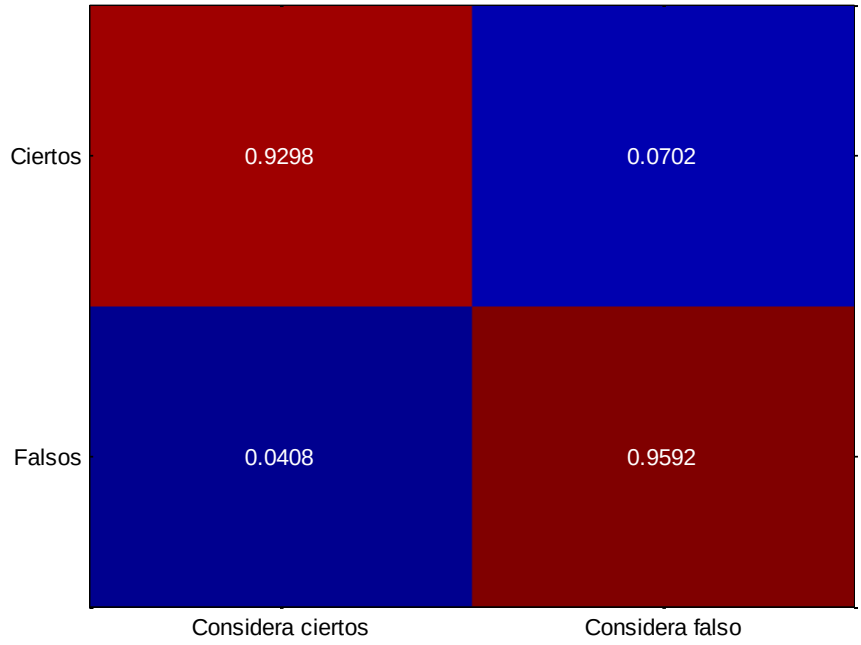

Figura B.12 Matriz de confusión utilizando SURF

Accuracy	0,95
Precision	0,93

Vemos que también cumple en términos de falsos positivos y, en este caso, se observa que el falso rechazo se reduce sustancialmente a tan sólo el 7% de los emparejamientos (Figura B.12). Por tanto, no es de extrañar que la exactitud mejore tanto respecto a los casos anteriores. La precisión también nos da un resultado óptimo para nuestro problema.