View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by idUS. Depósito de Investigación Universidad de Sevilla

Proyecto Fin de Grado

Grado en Ingeniería de las tecnologías de telecomunicación

Reconocimiento automático de escenas basado en visión por computador: Aplicaciones a la localización de vehículos autónomos aéreos

Autor: Víctor García Ramos-Catalina

Tutor: Fernando Caballero Benítez

Dpto. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2016

Proyecto Fin de Grado

Grado en Ingeniería de las tecnologías de telecomunicación

Reconocimiento automático de escenas basado en visión por computador: Aplicaciones a la localización de vehículos autónomos aéreos

Autor:

Víctor García Ramos-Catalina

Tutor:

Fernando Caballero Benítez

Profesor Contratado Doctor

Dpto. de Ingeniería de Sistemas y Automática

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2016

Proyecto Fin de Grado: Reconocimiento automático de escenas basado en visión por computador: Aplicaciones a la localización de vehículos autónomos aéreos.

Autor: Víctor García Ramos-Catalina

Tutor: Fernando Caballero Benítez

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2016

El Secretario del Tribunal

A Marta

A mi familia

Agradecimientos

Este proyecto simboliza el final de mi etapa como estudiante de grado, unos años que he dedicado intensamente al estudio de las tecnologías de las telecomunicaciones y como tal, me gustaría agradecer a todos los que han hecho posible que haya llegado a este punto.

A mis padres y familia que han sido el pilar fundamental que me ha permitido continuar durante estos largos años en esta complicada etapa, repleta de dificultades y desilusiones, así como de alegrías y esfuerzo.

A Marta, por ser tan paciente, comprensiva y generosa, que siempre ha estado allí en las alegrías y las decepciones, y espero que siga aquí siempre.

A mis amigos, por hacer improductivos los fines de semana y por seguir adelante por mucho que pase el tiempo.

A mis compañeros, por la necesaria camaradería de los estudiantes de ingeniería, que persistieron y persistirán hasta llegar hasta donde algunos llegaron, yo he llegado y muchos otros llegarán.

Resumen

La capacidad de reconocer los lugares en los que se ha estado es una importante característica en un sistema inteligente y automático especialmente para periodos largos de funcionamiento. En este documento se propone resolver este problema mediante una aplicación software de visión artificial diseñada para detectar si la imagen corresponde a una posición en la que se ha estado anteriormente. Las imágenes se captaran mediante una cámara situada en el vehículo. La principal característica que presenta esta aplicación es el uso de descriptores de tipo BRIEF lo que permitirá un análisis muy eficiente y rápido, ideal para situaciones reales de funcionamiento. La aplicación ha sido programada en código C++ haciendo uso de las funciones que proporcionan las librerías de OpenCV para el diseño de aplicaciones de visión artificial.

El sistema analiza todas las imágenes recibidas y las que considera localizaciones nuevas las añade a una base de datos interna con la que se comparan las imágenes que se van recibiendo. Cada imagen se almacena con su descriptor BRIEF y un identificador, junto con otros elementos útiles para su análisis.

Adicionalmente se proporcionan métodos para observar el comportamiento del sistema, como contadores internos para analizar el tiempo de procesamiento de cada imagen o una comparación entre una imagen descartada y la que se considera equivalente.

Abstract

The ability to recognize known places is an essential competence of any intelligent system that operates autonomously over longer periods of time. This document proposes BRIEF-Gist, a very simplistic appearance-based place recognition system based on the BRIEF descriptor. BRIEF-Gist is easy to implement and very efficient. Despite its simplicity, we can show that it performs an efficient and low error rate performance. We benchmark our approach using a real dataset.

Índice

Agradecimientos **ix**

Resumen **xi**

Abstract **xiii**

Índice **xiv**

Índice de figuras **xvii**

1 Introducción **1**
1.1 Motivaciones *1*
1.2 Objetivos *1*
1.3 Antecedentes *1*

2 Metodología **5**
2.1 Descriptor BRIEF y distancia de Hamming *5*
2.1.1 Distancia de Hamming 5
2.1.2 Descriptor BRIEF 6
2.2 C++ *7*
2.3 Librería OpenCV *7*
2.4 ROS (Robot Operating Sistem) *8*

3 Código **11**
3.1 Introducción *11*
3.2 Explicación del código *11*
3.2.1 Librerías 11
3.2.2 Estructuras y variables globales 12
3.2.3 Función inserta final 13
3.2.4 Función busca coincidencia 14
3.2.5 ImageCallback 16
3.2.6 Main() 18
3.2.7 Otros 19

4 Resultados **21**
4.1 Introducción *21*
4.2 Análisis *22*
4.2.1 1ª Sección 22
4.2.2 2ª Sección 22
4.2.3 3ª Sección (Ilustración 5) 22
4.2.4 4ª Sección (Ilustración 6) 22
4.2.5 5ª Sección (Ilustración 7) 22
4.2.6 6ª Sección (Ilustración 8) 23
4.2.7 7ª Sección (Ilustración 9) 23
4.2.8 8ª Sección (Ilustración 10) 23
4.2.9 9ª Sección (Ilustración 11) 23
4.2.10 10ª Seccion (Ilustración 12) 23

4.2.11 11ª Sección (Ilustración 13) 24
4.2.12 12ª Sección (Ilustración 14) 24
4.2.13 13ª Sección (Ilustración 15) 24
4.2.14 14ª Sección (Ilustración 16) 24
4.2.15 15ª Sección (Ilustración 17) 24
4.2.16 16ª Sección (Ilustración 18) 25
4.2.17 17ª Sección (Ilustración 19) 25
4.2.18 18ª Sección (Ilustración 20) 25
4.2.19 19ª Sección (Ilustración 21) 25
4.2.20 20ª Sección (Ilustración 22) 26
4.2.21 21ª Sección (Ilustración 23) 26
4.2.22 22ª Sección (Ilustración 24) 26
4.2.23 23ª Sección (Ilustración 25) 26
4.2.24 24ª Sección (Ilustración 26) 27
4.2.25 25ª Sección (Ilustración 27) 27
4.2.26 26ª Sección (Ilustración 28) 27
4.2.27 27ª Sección (Ilustración 29) 27
4.2.28 28ª Sección (Ilustración 30) 28
4.2.29 29ª Sección (Ilustración 31) 28
4.2.30 30ª Sección 28
4.3 Otras medidas 28
4.4 Reflexión 28

5 Conclusiones 31
5.1 Conclusiones finales 31
5.2 Perspectivas futuras 31

Referencias 33

Anexo A: Glosario 37

Anexo B: Código completo 39

Índice de figuras

Figura 1. Ejemplo distancia de Hamming 4

Figura 2. Logo OpenCV 6

Figura 3. Logo ROS 7

Figura 4. Esquema alabeo 19

Figura 5. Esquema cabeceo 19

Figura 6. Esquema guiñada 19

Figura 7. 3ª Sección del recorrido 20

Figura 8. 4ª Sección del recorrido 20

Figura 9. 5ª Sección del recorrido 20

Figura 10. 6ª Sección del recorrido 21

Figura 11. 7ª Sección del recorrido 21

Figura 12. 8ª Sección del recorrido 21

Figura 13. 9ª Sección del recorrido 21

Figura 14. 10ª Sección del recorrido 21

Figura 15. 11ª Sección del recorrido 22

Figura 16. 12ª Sección del recorrido 22

Figura 17. 13ª Sección del recorrido 22

Figura 18. 14ª Sección del recorrido 22

Figura 19. 15ª Sección del recorrido 22

Figura 20. 16ª Sección del recorrido 23

Figura 21. 17ª Sección del recorrido 23

Figura 22. 18ª Sección del recorrido 23

Figura 23. 19ª Sección del recorrido 23

Figura 24. 20ª Sección del recorrido 24

Figura 25. 21ª Sección del recorrido 24

Figura 26. 22ª Sección del recorrido 24

Figura 27. 23ª Sección del recorrido 24

Figura 28. 24ª Sección del recorrido 25

Figura 29. 25ª Sección del recorrido 25

Figura 30. 26ª Sección del recorrido 25
Figura 31. 27ª Sección del recorrido 25
Figura 32. 28ª Sección del recorrido 26
Figura 33. 29ª Sección del recorrido 26

1 INTRODUCCIÓN

1.1 Motivaciones

La visión por computador es la disciplina a la que nos referimos cuando tratamos con métodos con el objetivo de analizar, procesar o comprender imágenes reales mediante ordenadores lo que requiere una transformación de la información a un lenguaje que los ordenadores puedan manejar. Entre otros objetivos se pretende que el sistema sea capaz de trabajar de forma automática, diferenciando patrones o ayudando a una persona a identificar unos patrones concretos. La adquisición de datos se realiza normalmente por medio de cámaras de algún tipo.

Saber dónde se ha estado previamente es clave para el control automático de sistemas móviles, en nuestro caso un vehículo aéreo no tripulado, ya que esto facilita enormemente el corregir los algoritmos de posicionamiento para compensar el error existente en estos. De modo que deseamos una aplicación que sirva de apoyo a la hora de posicionar un sistema móvil para que se facilite su control, especialmente durante largos períodos de funcionamiento, lo que provocaría un aumento cada vez mayor del error, en caso de no recurrir a aplicaciones como la aquí mostrada.

Normalmente en las aplicaciones de visión por computador se pretende analizar una parte de la imagen, que previamente hay que localizar. Pero en este caso se trabaja con la imagen entera, ya que lo que pretendemos en comparar imágenes completas.

1.2 Objetivos

En esta aplicación se ha tomado como objetivo principal el realizar un sistema automático que sea capaz de determinar si una imagen recibida es lo suficientemente diferente a las imágenes englobadas en la base de datos, que se actualiza automáticamente. De esta forma, si la aplicación detecta una imagen recibida, que difiere de las demás, la interpretará como un lugar nuevo.

Las características de este sistema en las que se quiere hacer hincapié radican en que el sistema sea muy eficiente y tremendamente rápido de modo que sea útil en sistemas con pocas capacidades de procesamiento disponibles. Así mismo se pretende crear una base de datos de organización coherente y eficaz.

Adicionalmente se pretenden añadir herramientas que se utilicen para medir la eficiencia del sistema y para poder acceder a las imágenes de la base de datos.

Este sistema estará implementado sobre ROS lo que facilitará su uso en entornos reales.

1.3 Antecedentes

Los sistemas SLAM modernos están basados normalmente en la optimización eficiente de las restricciones probabilísticas o en gráficas de factores. Estos sistemas se dividen normalmente en *back –end* y *front-end* [1]. Los *back-end* optimizan creando un mapeado buscando la solución óptima con la trayectoria y el punto de aterrizaje dados por los sistemas *front-end*. El *front-end* se encarga de la asociación de datos, en general.

La detección fiable de lugares es difícil, especialmente en exteriores o lugares muy amplios, ya que las estructuras repetitivas y el sensado ambiguo conllevan un importante reto para los sistemas de reconocimientos de lugares. Las optimizaciones basadas en sistemas *back-end* para SLAM como iSAM [2], Sparse Pose Adjustment [1], iSAM2 [3], o g2o [4] no son robustas frente a excepciones por lo que un solo cierre de bucle erróneo provocaría un error catastrófico en el proceso de mapeado. Algunos desarrolladores han conseguido un 100% de precisión, por ejemplo concentrándose en prevenir falsos positivos, pero estos sistemas son

extremadamente complicados y no son aptos para aplicaciones en tiempo real.

Algunos de los trabajos más importantes sobre detección han sido dirigidos por Sivic y Zisserman [5] los cuales, tomando ideas de la recuperación de sistemas de texto, introdujeron el concepto de vocabulario visual. La idea fue posteriormente extendida a arboles de vocabulario por Nister y Stewenius [6], permitiendo usar eficientemente grandes vocabularios. Más tarde, Schindler [7] demostró un reconocimiento de un gran área, como una ciudad, usando tres estructuras: FAB-Map [8], descriptores de tipo SURF [9], y arboles Chow Liu, para aproximar la distribución de probabilidad de las imágenes y la correlación entre estas. Esto permitió reconocer diferentes lugares pese a la ambigüedad visual.

Recientemente, Cadena [10] combinó los métodos de reconocimiento basados en el aspecto con *Conditional Random Fields* para filtrar error causados por la ambigüedad visual entre distintos espacios. Maddern [11]n desarrolló mejoras en la robustez del FAB-Map incorporando información odométrica en el proceso de reconocimiento. Estos métodos mencionados anteriormente describen la apariencia de un lugar mediante distintas marcas (puntos de interés) y sus descriptores.

Los descriptores son formas de sintetizar la información de una imagen, como color, texturas o formas, de forma que sea más cómodo y rápido trabajar con ellas.

Los métodos anteriores describen las imágenes entorno a puntos de interés. Otra estrategia a seguir consiste en usar descriptores holísticos, que consiste en describir la imagen completa, no sus puntos característicos. Esta idea no es nueva y fue examinada por Oliva y Torralba [12][13] con la introducción del descriptor *Gist.* Este descriptor se refiere a una imagen completa y se construye a partir de filtros orientables con diferentes orientaciones y escalas. Más recientemente [14], se ha demostrado el reconocimiento de lugares usando el descriptor *Gist* en imágenes panorámicas en ambientes urbanos.

Otro descriptor muy eficaz es BRIEF (Binary Robust Independent Elementary Features) que fue introducido por Calonder [15]. El descriptor BRIEF es un vector de bits que se construye mediante un test binario en torno a un punto de la imagen. Calonder sugirió usar una simple comparación de valores de intensidad de pixel. Para una longitud del descriptor n se cogen n parejas de píxeles en la zona del punto que se haya designado como central y se comparan. Si se tienen un conjunto de imágenes a la misma resolución y se usa el mismo punto como central se podrá obtener un descriptor para cada imagen de forma que estos se pueden comparar, con por ejemplo la distancia de Hamming entre dos de los descriptores lo que daría un entero que supondría una razonable estimación de la diferencia entre las dos imágenes. Calonder demostró que este descriptor es superior a los descriptores SIFT [16] y SURF [9] tanto en el reconocimiento de imágenes como en su comportamiento temporal.

2 METODOLOGÍA

2.1 Descriptor BRIEF y distancia de Hamming

El descriptor BRIEF consiste en una cadena binaria que describe eficientemente los puntos característicos de una matriz. Es altamente selectiva incluso cuando se tienen relativamente pocos bits y tiene poca carga computacional, especialmente útil para sistemas con poca capacidad de procesamiento disponible. Además, la similitud entre descriptores puede hacerse mediante la distancia de Hamming que es muy eficiente. Como resultado tenemos un descriptor muy rápido de computar y comparar.

2.1.1 Distancia de Hamming

La distancia de Hamming fue descubierta por Richard Hamming [17], que introdujo el término para establecer una métrica capaz de establecer un código para la detección y auto-corrección de códigos en 1950. Se utiliza en telecomunicaciones para contar los bits corruptos y poder estimar el error de un sistema, a veces también se le llama distancia de señal. También se utiliza para medir distancias en genética [18].

En teoría de la información la distancia de Hamming entre dos cadenas de bits de igual tamaño es el número de posiciones donde los bits no son iguales en las dos cadenas. Para las cadenas binarias *a* y *b* la distancia de Hamming es igual al número de unos en la operación *a* XOR *b*. La distancia de Hamming se utiliza para definir algunas nociones esenciales en teoría de códigos, tales como códigos detectores de errores y códigos correctores de errores. En particular, se dice que un código C detecta k-errores si cualesquiera dos palabras $c1, c2 \in C$ que tienen una distancia de Hamming menor que k coinciden. Dicho de otro modo, un código detecta k-errores si y solo si la distancia de Hamming mínima entre cualesquiera dos palabras en él es al menos k+1.

Se dice que un código C corrige k-errores si para cada palabra W en el subyacente espacio de Hamming H existe al menos una palabra $c \in C$ tal que la distancia de Hamming entre w y c es menos que k. En otras palabras, un código corrige k-errores si y solo si la mínima distancia de Hamming entre cualesquiera dos de sus palabras es por lo menos 2k+1. Esto es más fácil de comprender geométricamente como que cualesquiera dos bolas cerradas de radio k centradas en distintas palabras son disjuntas. En este contexto se conoce a estas bolas como Esferas de Hamming. De esta manera, un código que tiene distancia de Hamming mínima d entre sus palabras puede detectar como máximod-1 errores y puede corregir [(d-1)/2] errores. Este último número es también conocido como el radio de empaquetado o la capacidad de corrección del código.

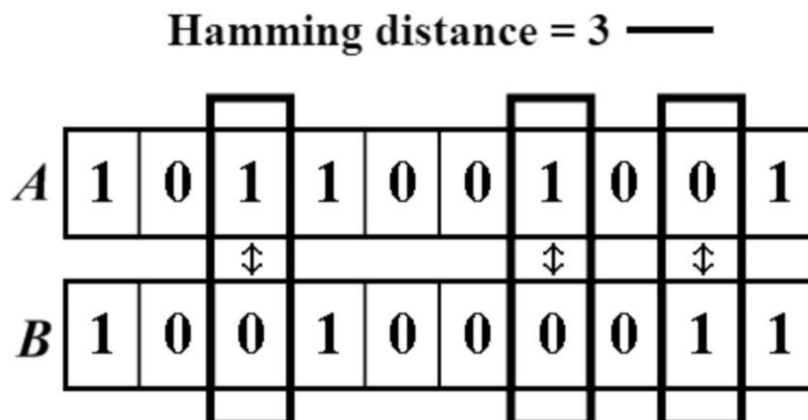

Ilustración 1: ***Distancia de Hamming. Esta imagen representa de forma bastante visual como se calcula la distancia de Hamming para cadenas de bits, que será para lo que se usara en este documento.***

2.1.2 Descriptor BRIEF

Los descriptores de puntos característicos, como la reconstrucción 3D o la localización por cámaras, son muy comunes en los sistemas de visión por computador de hoy en día. Estas aplicaciones muchas veces están asociadas a dispositivos portátiles, que debido a sus recursos computacionales limitados y al hecho de que tienen cada vez más cantidad de datos para procesar precisan de un mayor número de descriptores locales, que deben ser cada vez más rápidos y eficientes.

Una manera de acelerar las comparaciones y reducir el uso de memoria es usar descriptores cortos. Estos pueden obtenerse aplicando métodos de reducción como PCA [19] o LDA [20] para descriptores como SIFT o SURF. También es posible hacer reducciones mayores que puedan convertir un descriptor SIFT en una cadena binaria [21]. Estas conversiones, a pesar de ser efectivas, son muy costosas computacionalmente y requieren que se procese el descriptor completo antes de poder trabajar con él.

Todo este proceso de reducción se puede acortar computando directamente las cadenas binarias de las imágenes comparando la intensidad de pares de pixeles [22], esto es lo que nos referimos como BRIEF.

El descriptor BRIEF está basado en trabajos anteriores [23] que mostraban que una imagen podía ser clasificada en base a un pequeño número de comparaciones de intensidad. El resultado de estos test se usó para crear árboles aleatorios de clasificación (cita) o un clasificador Naive Bayesian [22] para reconocer zonas desde diferentes puntos de vista.

Se ha comprobado experimentalmente [15] que con 256 elementos es suficiente para conseguir unos excelentes resultados. Por lo tanto, este descriptor supera a otros descriptores rápidos anteriores como el SURF o el U-SURF e incluso superando en algunas situaciones a estos en términos de tasa de fallos.

El descriptor SIFT es muy preciso [16], pero tratándose de un vector de 128 elementos ejecutaría un proceso demasiado lento. Esto hace que no sea adecuado para aplicaciones en tiempo real, que implican manejar muchos puntos y descriptores en poco tiempo. Esto ha llevado a una amplia investigación para conseguir descriptores más rápidos sin perder la precisión de SIFT.

El descriptor SURF [9] es uno de los más conocidos. Al igual que el descriptor SIFT, el descriptor SURF se basa en gradientes de histogramas locales pero a diferencia de éste usa las imágenes integrales para el proceso. La versión de 64 elementos es muy precisa, tiene buen rendimiento, es muy popular y es estándar *de facto*. Sin embargo, cabe destacar que las direcciones son el factor limitante en los sistemas con descriptores BRIEF ya que un descriptor de 64 elementos de punto flotante implica 256 bytes. Esto es relevante cuando se deben almacenar millones de descriptores.

Hay tres métodos principales para reducir este número y mejorar de esta forma el rendimiento de los descriptores. El primero implica aplicar técnicas de reducción de tamaño como son el *Principal Component Analysis* (PCA) y el *Linear Discriminant Embedding* (LDE). El PCA es muy fácil de realizar y puede reducir el tamaño de los descriptores sin perder calidad [19]. Por otro lado, el método LDE requiere etiquetar los datos para ver qué descriptores deben unirse por lo que es más difícil de obtener. Se puede mejorar su rendimiento [20] pero esto puede llevar a que se degrade.

Una segunda forma de conseguir descriptores más cortos es convertir los elementos de punto flotante a enteros, que están codificados con menos bits. Tuytelaars [24] demostró que el descriptor SIFT puede codificarse con solo 4 bits por coordenada. Esta simple operación permite tanto ganar espacio en memoria como conseguir comparaciones más rápidas al computar las distancias con vectores más cortos, lo que los procesadores modernos realizan muy eficientemente. Sin embargo, con este método no se podría usar la distancia de Hamming ya que, al contrario que en el descriptor BRIEF, los bits están divididos en bloques de cuatro y no pueden ser procesados independientemente.

La tercera forma, y la más radical, consiste en binarizar el descriptor. Por ejemplo [21], utilizando Locality Sensitive Hashing (LSH) [25] para transformar vectores en punto flotante a cadenas binarias. Esto se consigue poniendo umbrales en los vectores y después multiplicándolos con una matriz adecuada. Luego la similitud entre descriptores será medida con la distancia de Hamming. El mismo algoritmo fue aplicado al descriptor GIST para obtener un descriptor binario de una imagen completa [26]. Otra manera de binarizar un descriptor GIST es usar *Nonlinear Neighborhood Component Analysis* [27], que es más potente pero es más lento.
A pesar de que estas tres técnicas obtienen resultados satisfactorios resultan ineficientes para descriptores largos, cuyo acortamiento implica mucho gasto a nivel de tiempo de computación.

En definitiva, el descriptor BRIEF se basa en un test que mide la diferencia de intensidad en una imagen para representar un descriptor de esta en forma de cadena binaria. Tanto su construcción como la comparación entre descriptores del mismo tipo es mucho más rápida que con los descriptores utilizados habitualmente y suele ser mejor a nivel de reconocimiento, siempre que no haya grandes zonas muy uniformes.

2.2 C++

C++ es un lenguaje de programación creado por Bjarne Stroustrup en los laboratorios de At&T en 1983 [28]. El C++ es un derivado del lenguaje C, el cual fue creado en la década de los 70 por Dennis Ritchie para la programación del sistema operativo, el cual surgió como un lenguaje orientado a la programación de sistemas (System Programming) y de herramientas (Utilities) recomendado sobre todo para programadores expertos, y que no llevaba implementadas muchas funciones que hacen a un lenguaje más comprensible. Sin embargo, aunque esto en un inicio se puede convertir en un problema, en la práctica es su mayor virtud, ya que permite al programador un mayor control sobre lo que está haciendo. Años más tarde, el programador Bjarne Stroustrup, creó lo que se conoce actualmente como C++.

Las necesidades de ciertas facilidades de programación, que conforman muchos de los principios usados en la programación actual, como son las llamadas clases y objetos, llevaron a un rediseño de C. De esta forma el lenguaje C pasó de no soportar, al menos directamente, dichas características a ampliar sus posibilidades pero manteniendo su mayor cualidad, la de permitir al programador en todo momento tener control sobre lo que está haciendo, consiguiendo así una mayor rapidez que no se conseguiría en otros lenguajes.

A modo de síntesis, C++ pretende llevar a C a un nuevo paradigma de clases y objetos con los que se realiza una comprensión más humana basándose en la construcción de objetos, con características propias solo de ellos, agrupados en clases.

2.3 Librería OpenCV

OpenCV (Open Source Computer Vision) es una plataforma libre que contiene más de 500 algoritmos y funciones optimizados para su uso en visión por computador y procesado de videos. Desde su comienzo en 1999 ha sido usada por los principales desarrolladores del campo de la visión por computador como principal herramienta de desarrollo [29].

Ilustración 2

Inicialmente, el proyecto OpenCV fue creado como una iniciativa de Intel para desarrollar aplicaciones con mucha carga de CPU, entre los que se incluían otros proyectos, como por ejemplo la representación de muros 3D. Los principales colaboradores del proyecto incluían un buen número de expertos en optimización de Intel Rusia, así como el equipo de desarrollo de librerías de Intel.

Entre los objetivos del proyecto que daría lugar a la creación de OpenCV destacan:

- Investigación de visión por computador, de código abierto y optimizado para la infraestructura básica de los códigos de visión por computador.
- Propagar el conocimiento visión por computador dando acceso a una infraestructura común sobre la cual los desarrolladores puedan trabajar, lo que ayudaría también a que los códigos sean más comprensibles y transferibles.
- Promover el desarrollo de aplicaciones avanzadas basadas en visión por computador creando un código gratuito, transportable y optimizado.

La primera versión alfa de OpenCV fue mostrada al público en la conferencia del IEEE sobre visión por computador y reconocimiento de patrones en el año 2000, y cinco betas fueron lanzadas entre 2001 y 2005. La versión 1.0 fue lanzada en 2006. A mediados del 2008, OpenCV obtuvo el apoyo corporativo de Willow Garage, lo que reactivo su desarrollo. Una versión 1.1 fue lanzada en 2008.

El segundo gran lanzamiento de OpenCV, OpenCV2, fue en octubre de 2009, éste incluía importantes cambios

en la interfaz C++, nuevas funciones y mejores implementaciones de funciones ya existentes Desde entonces se lanzan nuevas versiones oficiales cada seis meses [30] y el desarrollo lo realiza un equipo independiente ruso con el apoyo comercial de algunas corporaciones.

En agosto de 2012, el soporte de OpenCV fue tomado por una organización sin ánimo de lucro OpenCV.org la cual mantiene una web para desarrolladores [31] y otra para usuarios.

OpenCV está escrito en C++ y su interfaz principal también está en C++. También tiene conexiones a Python, Java y Matlab.

Para construir aplicaciones de visión por computador debes de ser capaz de acceder a las imágenes. En OpenCV las imágenes se tratan como matrices, usando la estructura de datos *cv::Mat* Cada elemento de la matriz representa un pixel. Para una imagen en blanco y negro cada pixel se compone de un valor de 8-bits sin signo lo que implica que si el valor 0 es el negro el 255 es el blanco. Para una imagen de color tres de los tipos de valores anteriores son necesarios, para representar los tres canales habituales (rojo, verde, azul). Luego se construye una matriz con tríos de valores.

2.4 ROS (Robot Operating Sistem)

ROS es un sistema para el desarrollo de software para robots que provee la funcionalidad de un sistema operativo en un grupo de computadores homogéneo. ROS se desarrolló originalmente en 2007 bajo el nombre de *switchyard* por el Laboratorio de Inteligencia Artificial de Stanford para dar soporte al proyecto del Robot con Inteligencia Artificial de Stanford (STAIR2 [32]). Desde 2008, el desarrollo continúa primordialmente en Willow Garage [33][34].

Ilustración 3

ROS provee los servicios estándar de un sistema operativo tales como abstracción del hardware, control de dispositivos de bajo nivel, implementación de funcionalidad de uso común, paso de mensajes entre procesos y mantenimiento de paquetes. Está basado en una arquitectura de grafos donde el procesamiento toma lugar en los nodos que pueden recibir, mandar y multiplexar mensajes de sensores, control, estados, planificaciones y actuadores, entre otros. La librería está orientada para un sistema UNIX (Ubuntu (Linux)) aunque también se está adaptando a otros sistemas operativos como Fedora, Mac OS X, Arch, etc.

ROS tiene dos partes básicas: la parte del sistema operativo, ros, como se ha descrito anteriormente y ros-pkg, una suite de paquetes aportados por la contribución de usuarios (organizados en conjuntos llamados pilas o en inglés stacks) que implementan la funcionalidades tales como localización y mapeo simultáneo, planificación, percepción, simulación, etc.

Escribir software para robots es difícil, en parte debido al continuo crecimiento del campo de la robótica. Los robots además pueden tener una amplia variedad de hardware lo que complica aún más el código. Además el código requerido para trabajar en robótica tiende a ser muy extenso, debe contener una amplia pila desde el software de los drivers hasta los niveles más altos de abstracción. El nivel de experiencia y conocimientos necesarios para desarrollar un proyecto de robótica son inabarcables para un solo desarrollador, por lo que las arquitecturas de software deben soportar la integración de distintos sistemas.

Durante años los equipos de desarrolladores han creado una amplia variedad de complejos sistemas para manejar más fácilmente el prototipado y la experimentación con robots, dando lugar a multitud de software usado tanto en industria como en ámbitos académicos. Cada uno de estos sistemas fue diseñado para un propósito en particular, en respuesta las debilidades de los otros sistemas existentes o para enfatizar aspectos que ninguno de los sistemas anteriores enfatizaba. De la necesidad de simplificar el desarrollo de robótica, y gracias a todos los sistemas que se crearon anteriormente se creó ROS.

La idea de ROS es tener una red para programación de robots ampliamente usada y que puedas usar partes de código de otros robots de modo que realizando pequeñas variaciones sea válido.

3 CÓDIGO

3.1 Introducción

En esta sección se analizará todo el código del programa línea a línea, explicando la razón de cada una y su objetivo. En líneas generales el programa, a partir de una serie de imágenes procedentes de una fuente externa, las analizará y determinará cuáles de ellas son relevantes. Todas las que se consideren relevantes o válidas serán almacenadas en una base de datos junto con información complementaria. Se considerarán relevantes las imágenes que tengan un grado de no similitud considerable, con respecto a las imágenes de la base de datos. Todo esto se conseguirá haciendo uso de ROS y OpenCV.

3.2 Explicación del código

3.2.1 Librerías

```
#include "ros/ros.h"
#include "std_msgs/String.h"
```

Primero tenemos la inclusión de todas las librerías necesarias. Primero tenemos las librerías generales de ROS que se incluyen en *ros/ros.h,* junto con las de mensajes de ROS, *std_msgs/String.h* que maneja los tipos de datos y estructuras de mensaje más básicas.

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
```

Luego tenemos algunas librerías generales de C, *stdlib.h* que contiene los prototipos de funciones de C para gestión de memoria dinámica, control de procesos y otras, y *stdio.h* que maneja las entradas y salidas, y por último *time.h* que provee de herramientas para medir el tiempo en el programa y así estudiar su rendimiento.

```
#include <sstream>
#include <iostream>
#include <vector>
```

Más adelante nos encontramos con librerías de C++ que se encargan de diversas tareas, la primera permite que el programa se comunique con otros, mande o envíe información. El segundo controla el flujo de entrada y salida. Y el tercero que está pensada para trabajar con estructuras unidimensionales de datos.

```
#include "opencv2/core/core.hpp"
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/features2d/features2d.hpp"
#include <opencv2/nonfree/features2d.hpp>
#include <opencv2/legacy/legacy.hpp>
```

Después tenemos múltiples librerías de OpenCV con distintas utilidades, desde el manejo básico de estructuras

hasta funciones para procesar imágenes. Todas estas librerías son necesarias para poder utilizar muchas de las funciones necesarias para la realización del proyecto. Estas librerías son muy comunes y son utilizadas en la mayoría de programas en los que se trabaja sobre imágenes. Hay que concretar que la librería "`legacy.hpp`" es un tanto particular, su objetivo no es proveer de nuevas funciones con las que trabajar, sino que compatibiliza las funciones antiguas con las nuevas. Esto es muy útil, ya que como estamos utilizando una plataforma en constante cambio muchas veces la documentación y guías contienen llamadas a funciones desactualizadas. Con esta librería se eliminan muchos problemas tanto presentes como futuros a la hora tanto de buscar información como de mantener el programa actualizado.

```
#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/image_encodings.h>
#include <sensor_msgs/CameraInfo.h>
```

Estas son librerías de ROS que se encargan principalmente de la comunicación del programa con el exterior. El primero contiene todas las funciones que se encargan de gestionar el transporte de imágenes, en este caso de recibirlas, es una librería básica para programas con arquitecturas de tipo cliente-servidor, en este caso seriamos el cliente. La segunda es la librería que permite pasar de imágenes de ROS a imágenes de OpenCV y viceversa, ya que las dos plataformas usan sistemas distintos para trabajar con las imágenes. Las dos últimas pertenecen al paquete "`sensor_msgs`" que contiene todas las funciones y la información necesaria para codificar o decodificar la información recibida del exterior o a enviar. En concreto tenemos la librería que permiten usar los tipos de codificación de imágenes más habituales junto con la librería que permite trabajar con la información proporcionada por cámaras de diverso tipo.

```
using namespace std;
using namespace cv;
```

Estas estructuras ayudan a simplificar la programación, ya que definen tipos de funciones que se van a usar. Tanto "`std`" como "`cv`" son las clases a la que pertenecen la mayoría de la funciones usadas. En C++ hay que escribir la ruta completa a la hora de llamar a una función, por ejemplo: cv::función, usando estas estructuras simplificarían las llamadas a funciones reduciéndolas a, por ejemplo: función, lo que agiliza la elaboración del programa.

3.2.2 Estructuras y variables globales

```
struct imagen
{
    int id;
    Mat img_original;
    Mat img_red;
    Mat img_descriptor;
    struct imagen * img_sig;
};
```

Primero tenemos la estructura básica sobre la que vamos a trabajar, en ella contendremos todos los datos que hemos considerados fundamentales a la hora de identificar a cada imagen. Primero tenemos un entero que se corresponde al identificador de la imagen, este indica el orden en el que se aceptó como imagen válida. Este campo solo se rellenará una vez pasado un filtro que indica la validez de la imagen. Este dato es útil para el post-procesamiento que se haga fuera del programa, sobre la base de datos creada. Después tenemos para almacenar la imagen original según como es recibida. Luego tenemos la imagen con la resolución inferior, con la que se obtiene el descriptor. Esto nos lleva al descriptor de la imagen. Por último tenemos un puntero a estructura que

servirá para crear la base de datos en forma de lista. Esta estructura formará la unidad básica de la base de datos que se creara para almacenar las imágenes, esta base de datos es una parte fundamental del funcionamiento del programa. Además esta base de datos puede ser muy útil para otros programas.

```
typedef struct imagen *Tlista;
```

Después definimos un puntero a la estructura anterior, este es necesario para trasladar la información de las estructuras ya que en C++ no se puede usar una estructura como parámetro de una función, hay que usar un puntero.

```
Tlista principio;
```

Esta es una variable global que apuntará al principio de la base de datos. Es útil porque permite separar la base de datos, que no debe verse afectado excepto a la hora de definirlo inicialmente, ya que en este programa no se contempla el borrado de imágenes. Este elemento sirve como referencia a la hora de trabajar con la base de datos.

3.2.3 Función inserta final

Esta es una sencilla función que añade un nuevo elemento a la base de datos al final de ésta

```
void insertarFinal(Tlista &lista, int valor , Mat original , Mat red , Mat descriptor)
```

Primero tenemos los parámetros que se le pasan a la función. Se pasa el principio de la lista junto con todos los valores para los campos necesarios en cada elemento de la base de datos.

```
{
    Tlista t, q = new(struct imagen);
```

Necesitamos crear estos dos elementos, uno para que sea el nuevo nodo de la lista y el otro para recorrer la lista hasta encontrar el final. No se utiliza directamente el puntero pasado como parámetro, ya que no queremos que éste se modifique en el proceso padre.

```
    q->id  = valor;
    original.copyTo(q->img_original);
    red.copyTo(q->img_red);
    descriptor.copyTo(q->img_descriptor);
    q->img_sig = NULL;
```

Aquí se rellenan los valores correspondientes al nuevo nodo, estos valores se han recibido como parámetro.

```
  if(lista==NULL)
       {
       principio = q;
       lista = q;
       }
```

Si lista es igual a NULL esto implica que la base de datos está vacía, si esto pasa el nodo que tenemos se corresponde con el primero así que se situaría al principio de la base de datos.

```
 else
     {
     t = lista;
     while(t->img_sig != NULL)
         {
         t = t->img_sig
         }
     t->img_sig = q;
     }
}
```

Por último si no es el primero se recorre con un while hasta encontrar el final, acto seguido se añade.

3.2.4 Función busca coincidencia

Esta función es básica en el funcionamiento del programa, ya que determina, a partir de la información recibida como parámetros determinar si una imagen es válida o no, lo que implica saber si estamos en una nueva localización.

```
float busca_conincidencia (Tlista &lista , Mat descriptor , Mat imagen , int id)
```

Primero tenemos en la declaración los parámetros que se pasan. Tenemos por un lado el acceso a la base de datos, junto con descriptor de la imagen que se quiere introducir, que es elemento que se usa para comparar las imágenes entre sí. Además se incluyen la imagen original y la ID que se le ha asignado por motivos de post-procesado y de ajuste del programa.

```
{
float aux = 0;
float dist_hamm_min = 500;
float dist_hamm = 0;
int id_match = 0;
Mat img_aux;
Mat negro = imread("/home/victor/Descargas/extra_image.jpg");
Tlista t = new(struct imagen);
```

Luego tenemos las declaraciones de los elementos a utilizar. Primero destacamos “aux” que será lo que devuelva la función. Luego tenemos “dist_hamm_min” y “dist_hamm” que son las variables en las que se guardan las distancias de Hamming, una para la mínima, que es la más relevante, y otra para la que se acaba de calcular, además para evitar problemas con los algoritmos se inicializa “dist_hamm_min” a 500, un valor muy alto. “img_aux” y “negro” se utilizan como imágenes para mostrar y hacer una comparación visual. Por ultimo “t” se utiliza para recorrer la base de datos.

```
 if(lista==NULL)
     {
     aux = 0;
     }
```

Primero comprobamos si la lista está vacía, si es así se considerara automáticamente que la imagen es válida ya

que no hay con que compararla.

```
else
    {
    t = lista;
    while(t != NULL)
        {
        dist_hamm = norm(t->img_descriptor,descriptor,NORM_HAMMING);
        if (dist_hamm < dist_hamm_min)
            {
            dist_hamm_min = dist_hamm;
            id_match = t->id;
            img_aux = t->img_original;
            }
        t = t->img_sig;
        }
```

En este fragmento se comparan los descriptores, el de la imagen actual con todos los de la base de datos. La línea principal es “dist_hamm = norm(t->img_descriptor,descriptor,NORM_HAMMING);” aquí se obtiene la distancia de Hamming entre dos descriptores, este resultado se compara con el menor obtenido anteriormente. Recordemos que cuanto menor sea la distancia de Hamming mayor será la similitud entre dos imágenes, por lo tanto para saber si tenemos una imagen nueva debemos obtener primero la imagen a la que más se parezca. Si hemos obtenido una nueva distancia menor que la anterior se actualizarán los datos. Este proceso se repetirá hasta que se haya recorrido entera la base de datos.

```
if (dist_hamm_min > 90)
   {
   aux = 0;
   imshow("view", imagen);
   imshow("bbdd", negro);
   }
else
   {
   aux = id_match;
   imshow("view", imagen);
     imshow("bbdd", img_aux);
     }
```

En este fragmento se aceptan o descartan las imágenes. Mediante pruebas experimentales se ha determinado que el nivel de corte más adecuado es 90. De modo que si la distancia mínima es menor que 90 se considerará la imagen como no válida. Además se muestran las imágenes, en el caso de que la imagen sea aceptada la imagen aceptada junto con otra en negro, y en caso que no sea aceptada se mostrara la imagen no aceptada junto con la imagen que se ha considerado similar a ella de forma que se puede observar el proceso de descarte lo que ayuda al proceso de depuración y al perfeccionamiento del programa.

```
}
return aux;
}
```

Por último se devuelve el valor "aux" que será 0 en caso de una imagen valida y la identificación correspondiente del elemento de la base de datos al que se haya encontrado mayor similitud, si la imagen no es válida.

3.2.5 ImageCallback

Esta función es el núcleo del programa, es un tipo de función de ROS que se utiliza para recibir información de los sensores, en nuestro caso de una cámara de forma que se pueda procesar en tiempo real. En concreto cada vez que recibimos una imagen por el canal de la cámara se ejecutara la función.

```
void imageCallback(const sensor_msgs::ImageConstPtr& msg)
```

Primero tenemos la declaración estándar de la función. Su argumento se corresponde con el canal por donde vienen las imágenes.

```
{
static int count=1;
static Tlista lista = NULL;
static double dist_hamm; //mirar para quitar
static float aux = 1;
static Mat lista_desc(1000,32,CV_8U); //mirar para quitar
imagen actual;
cv_bridge::CvImagePtr img_ptr;
```

Después tenemos las variables que son necesarias. Primero tenemos "count" que se corresponderá con el índice que contará cuantas imágenes se van añadiendo, además se utiliza para rellenar el atributo "id" de cada imagen. Luego "lista" se utilizará para almacenar la base de datos. En "aux" se almacenarán el valor devuelto por "busca_conincidencia". En actual se guardarán los datos de la imagen nueva correspondiente a cada ciclo. Por ultimo "img_ptr" es necesario para pasar la imagen recibida que estará codificada por ROS a un formato adecuado para OpenCV. Algunos de los elementos anteriores han sido declarados como estáticos ya que esta función se ejecuta con cada nueva imagen y no es deseable que se declaren de nuevo cada vez principalmente porque se borraría su valor anterior.

```
try
  {
  img_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::BGR8);
  img_ptr->image.copyTo(actual.img_original);
  resize(actual.img_original , actual.img_red ,   Size(actual.img_original.cols/4
, actual.img_original.rows/4));
  actual.id=count;
  std::vector<cv::KeyPoint> kpts1;
  cv::BriefDescriptorExtractor extractor(32);
  kpts1.push_back(cv::KeyPoint(40, 40, 30));
  Mat descriptor1;
  extractor.compute(actual.img_original, kpts1, descriptor1);
  actual.img_descriptor = descriptor1;
```

Este fragmento se encarga de procesar la imagen recibida y rellenar todos los campos para su posible almacenamiento. Primero se transforma la imagen en algo que OpenCV pueda manejar y posteriormente se copia en "actual", que es el elemento a rellenar. Luego obtenemos la versión reducida de la imagen original y la almacenamos, esto se hace para reducir la información de la imagen a lo más esencial lo que aumenta enormemente la eficacia del método. Posteriormente se le asigna a "id" el valor de "count". Luego se crean los elementos necesarios para obtener el descriptor de la imagen reducida. Para ello se crea un vector de puntos que determinará los puntos clave sobre los que aplicar el algoritmo y se crea también un extractor de 32 bits del tipo Brief, este es el tamaño que se ha determinado adecuado. Por último se computa el algoritmo y se guarda su resultado.

```
if (count == 0)
    {
    insertarFinal(lista  ,  count  ,  actual.img_original  ,  actual.img_red   ,
actual.img_descriptor);
    count++;
    }
```

Este fragmento se ejecutará en el primer ciclo ya que "count" no se reseteará durante la ejecución del programa. Simplemente añade el nodo a la base de datos, que será el primer elemento, porque es el primer ciclo gracias a lo cual no hay que hacer comprobaciones extra.

```
else
    {
    aux = busca_conincidencia(lista , actual.img_descriptor , actual.img_original
, count);
    if (aux == 0)
       {
       count++;
       insertarFinal(lista  , count   , actual.img_original  , actual.img_red  ,
actual.img_descriptor);
       }
   }
```

Este es el fragmento que se ejecuta normalmente. Ejecuta la función "`busca_conincidencia`" y dependiendo del resultado añade, o no, el nodo a la base de datos.

```
  }
  catch (cv_bridge::Exception& e)
  {
    ROS_ERROR("Could not convert from '%s' to 'mono8'.", msg->encoding.c_str());
  }
}
```

Este último fragmento de la función se ejecuta como una excepción en caso de que haya algún error en el traspaso de imágenes.

3.2.6 Main()

Por último tenemos la función “main” del programa que consiste básicamente en mantener la búsqueda de imágenes por el canal establecido.

```
int main(int argc, char **argv)
```

Primero tenemos la declaración estándar, que es común a la de C.

```
    {
    init(argc, argv, "listener");
    NodeHandle nh;
```

Esta primera parte se utiliza para iniciar el nodo de ROS, esto es necesario para poder comunicarse con otros sistemas que utilicen ROS. En nuestro caso nos comunicaríamos con el sistema que proporciona las imágenes.

```
    namedWindow("view");
    namedWindow("bbdd");
    startWindowThread();
```

Este fragmento se encarga de crear las ventanas donde se muestran las imágenes.

```
    image_transport::ImageTransport it(nh);
    image_transport::Subscriber sub = it.subscribe("/euroc5/cam0/image_raw", 1,
imageCallback);
```

Aquí se establece la conexión con la cámara que proporciona las imágenes. El programa que transmite las imágenes ha sido proporcionado por el tutor de este proyecto, el profesor Fernando Caballero.

```
    ros::spin();
```

Esta línea sirve para que el programa se ejecute constantemente.

```
    destroyWindow("view");
    destroyWindow("bbdd");
    }
```

Por último estas líneas cierran las ventanas de las imágenes usadas.

3.2.7 Otros

Hay otros fragmentos de código que se han utilizado, especialmente para depuración del programa, pero que no se han colocado en el código final.

```
cout << cualquier_variable;
cout << "\n";
```

Este fragmento se ha utilizado para mostrar por pantalla el valor de las variables, para así poder depurar más fácilmente el programa.

```
clock_t start, end;
double elapsed;
start = clock();
…………………………………………………….
end = clock();
elapsed = ((double) (end - start)) / CLOCKS_PER_SEC;
```

Este código se utiliza para medir el tiempo de la ejecución del código. Esto es importante no solo por la eficacia del programa, sino también porque al encontrarnos en una aplicación en tiempo real los tiempos de ejecución importan y pueden estar limitados por la situación en la que se vaya a usar el código.

4 RESULTADOS

4.1 Introducción

En esta sección procederemos a la realización de una prueba del programa. Para ello disponemos de un grupo de imágenes, proporcionadas por el tutor de este proyecto, el profesor Fernando Caballero. Todas estas imágenes se corresponden a una situación real de un dron en espacio interior. Se analizarán las imágenes por movimientos del robot y se justificará la inclusión, o no de nuevas imágenes. Tras el análisis se mostrará que el programa detecta satisfactoriamente las zonas en las que ha estado previamente, y que además presenta una considerable robustez frente a perturbaciones en el movimiento como cabeceos o alabeos. Para quien no esté familiarizado con el vocabulario aeronáutico, estos son distintos tipo de movimiento que explicaré a continuación.

Alabeo es un movimiento en el eje longitudinal, que es un eje imaginario que se extiende desde la parte frontal o morro hasta la parte posterior, la cola en caso de un avión.

Ilustración 4: Alabeo

Cabeceo es un movimiento en el eje transversal, que es un eje imaginario que, en un avión, iría desde el extremo de un ala al extremo de la otra ala. En un dron sería un eje equivalente.

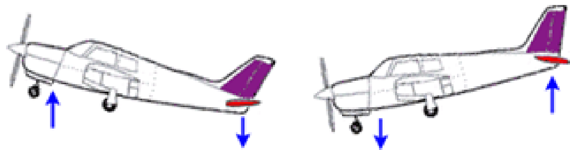

Ilustración 5: Cabeceo

Guiñada es un movimiento en el eje vertical, este es un eje imaginario situado en el centro de gravedad y transversal a los ajes transversal y longitudinal.

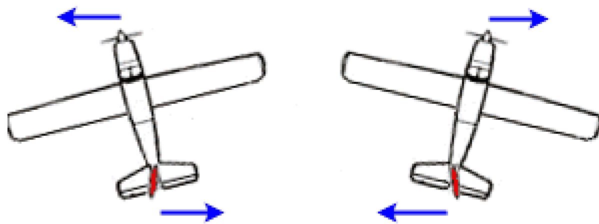

Ilustración 6: Guiñada

4.2 Análisis

4.2.1 1ª Sección

El primer segmento de video es simplemente en dron quieto, así que solamente se añade una imagen, esto también sirve de prueba inicial para comprobar el funcionamiento del programa ante imágenes estáticas, para las cuales se demuestra que funciona correctamente. Este estado dura desde el segundo 0 hasta el segundo 40.5.

4.2.2 2ª Sección

Desde el segundo 40.5 hasta el segundo 56 se procede al despegue. En este movimiento se generan muchas imágenes, esto es debido a dos factores. El primer factor se debe a la realización de un movimiento algo brusco, lo que perturba mucho la imagen y que puede ocasionar imágenes distorsionadas que suelen ser consideradas como falsos positivos. Por otro lado es el primer movimiento, lo que implica que casi no hay imágenes con las que comparar por lo que es natural obtener muchos positivos.

4.2.3 3ª Sección (Ilustración 5)

Este es uno de los movimientos principales ya que se mueve en línea recta sobre el eje central donde se realizarán el resto de movimientos y junto con la anterior será la principal fuente de imágenes a comparar. A partir de ahora acompañaremos cada sección con un esbozo del recorrido realizado. La flecha indica la dirección de la cámara, que siempre será la misma aunque sufrirá efectos de alabeo, cabeceo y guiñada durante el recorrido, que como se verá más adelante no serán muy relevantes.

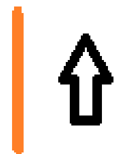

Ilustración 7

4.2.4 4ª Sección (Ilustración 6)

En esta parte el dron retrocede y en esta ocasión solo se añade una imagen ya que el recorrido realizado es muy similar al hecho en el segmento anterior por lo que se empieza a ver el funcionamiento del programa. Este segmento esta entre los segundos 63.5 y 67, y hay que prestar especial atención al intervalo entre los segundos 66 y 67 ya que se produce el primer cierre de bucle del programa, y se ve que el funcionamiento es el esperado, es decir no se producen nuevas imágenes en este segmento.

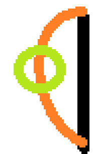

Ilustración 8

4.2.5 5ª Sección (Ilustración 7)

Aquí tenemos un movimiento circular hacia delante. Progresivamente se observa que se reduce el número de nuevas imágenes, es este caso se han añadido 2. Esto se debe a que nuestra base de datos es cada vez mayor, por lo tanto se reconocen los lugares próximos a donde ya hemos estado. Además para que se observe mejor el funcionamiento del programa se ha añadido las zonas que dan lugar a las nuevas imágenes. Este movimiento sucede en el intervalo de 67-70 segundos

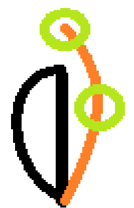

Ilustración 9

4.2.6 6ª Sección (Ilustración 8)

En este trayecto hacia atrás se añaden dos imágenes, a partir de aquí ya se tiene una base de datos lo suficientemente amplia de la zona en la que se está moviendo el dron por lo que el número de imágenes nuevas bajará drásticamente. Este movimiento dura desde el segundo 70 al 75.5. Durante el recorrido en la primera parte se produce un descenso algo brusco y en el trayecto final un alabeo algo brusco también, lo que parece que produce las nuevas imágenes.

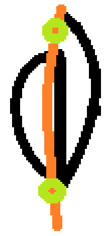

Ilustración 10

4.2.7 7ª Sección (Ilustración 9)

Entre los segundos 75.5 y 77 realiza un movimiento de cabeceo y vuelve a la posición anterior, en este caso se añade una imagen.

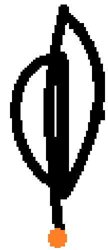

Ilustración 11

4.2.8 8ª Sección (Ilustración 10)

Entre los segundos 77 y 78.5 realiza un movimiento de alabeo tanto a la derecha como a la izquierda. En este caso el algoritmo aguanta la perturbación y no añade ninguna imagen, con lo que se empieza a demostrar la robustez del sistema.

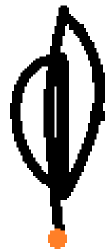

Ilustración 12

4.2.9 9ª Sección (Ilustración 11)

Aquí se mueve un poco a la derecha junto con cabeceo hacia abajo. Aquí no se generan imágenes nuevas ya que ya se había sometido anteriormente a un movimiento de cabeceo y ya ha añadido imágenes para esta posición. Este movimiento dura de los segundos 78.5 a 79.5.

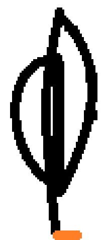

Ilustración 13

4.2.10 10ª Seccion (Ilustración 12)

De los segundos 79.5 a 81.5 realiza un movimiento hacia delante recuperándose del cabeceo anterior. No se generan nuevas imágenes. A partir de ahora no se generan casi imágenes por lo que la mayoría del recorrido está preparado para demostrar la robustez del sistema.

Ilustración 14

4.2.11 11ª Sección (Ilustración 13)

Desde el segundo 81.5 al segundo 83.5 se realiza un movimiento hacia delante, no se añade ninguna imagen.

Ilustración 15

4.2.12 12ª Sección (Ilustración 14)

Entre el segundo 83.5 y el segundo 89 se realiza un amplio movimiento hacia atrás con curva, en el cual se añade una imagen.

Ilustración 16

4.2.13 13ª Sección (Ilustración 15)

Del segundo 89 hasta el 92 se hace un movimiento en diagonal hacia atrás en el que no se añade ninguna imagen.

Ilustración 17

4.2.14 14ª Sección (Ilustración 16)

Desde el segundo 92 hasta el 95 se realiza un movimiento en diagonal hacia delante sin añadir ninguna imagen.

Ilustración 18

4.2.15 15ª Sección (Ilustración 17)

Del segundo 95 hasta el 97 se realiza otro movimiento en diagonal hacia delante. Ya se puede obsevar perfectamente el funcionamiento del sistema ya que no introduce casi imágenes nuevas ya que nos estamos moviendo en la misma zona. En este segmeto tampoco se añade ninguna imagen.

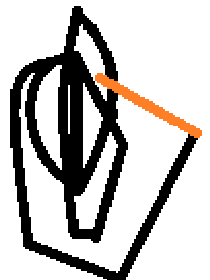

Ilustración 19

4.2.16 16ª Sección (Ilustración 18)

Ahora se realiza un movimiento hacia adelante con cabeceo hacia abajo, durante este cabeceo se produce una nueva imagen. Este segmento dura desde el segundo 97 al 100.5.

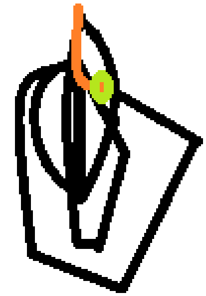

Ilustración 20

4.2.17 17ª Sección (Ilustración 19)

Se realiza un movimiento hacia la derecha ascendente, con un poco de cabeceo provocado por la maniobra, durante los segundos de 100.5 a 104.5. Se produce una imagen debido al cabeceo.

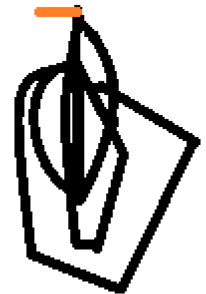

Ilustración 21

4.2.18 18ª Sección (Ilustración 20)

Del segundo 104.5 al 107.5 se realiza un movimiento hacia atrás en el que no se producen nuevas imágenes.

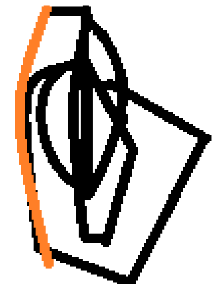

Ilustración 22

4.2.19 19ª Sección (Ilustración 21)

Desde el segundo 107.5 hasta el 114.5 se produce un movimiento hacia delante con ascenso en la parte final. Se añade un imagen pero es debido a una perturbación en el movimiento que provoca un cambio brusco de iluminación, esto hace que surja un falso positivo.

Ilustración 23

4.2.20 20ª Sección (Ilustración 22)

Del segundo 114.5 hasta el 118 se produce un movimiento de curva hacia atrás con descenso y cabeceo hacia abajo lo que provoca una nueva imagen debido al fuerte cabeceo.

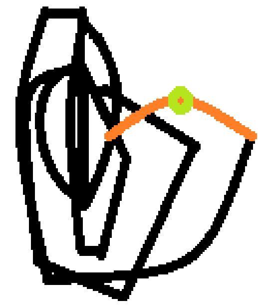

Ilustración 24

4.2.21 21ª Sección (Ilustración 23)

Movimiento en curva un poco hacia delante y luego hacia atrás, sin ninguna imagen nueva, desde el segundo 118 hasta el 125.5.

Ilustración 25

4.2.22 22ª Sección (Ilustración 24)

Desde el segundo 125.5 hasta el 130 se realiza un movimiento hacia delante diagonal a la izquierda con una importate guiñada hacia la izquierda que dado que es un cambio bastante importante en las imágenes produce 3 nuevas imágenes debido principalmente a la fuerte guiñada.

Ilustración 26

4.2.23 23ª Sección (Ilustración 25)

Desde el segundo 130 hasta el 134.5 se produce un movimiento hacia atrás con efectos de guiñada y alabeo moderados, pero el programa es capaz de soportarlo y no se añade ninguna imagen.

Ilustración 27

4.2.24 24ª Sección (Ilustración 26)

Del segundo 124.5 hasta el 143 se produce un movimiento en diagonal hacia delante con guiñada muy similar al de la sección 22, pero en este caso no se han añadido nuevas imágenes. Este caso es otro buen ejemplo del buen funcionamiento del programa ya que al repetir un movimiento similar a uno anterior no se han añadido imágenes nuevas.

Ilustración 28

4.2.25 25ª Sección (Ilustración 27)

Se produce un pequeño movimiento hacia delante con un considerable cabeceo que no provoca nuevas imágenes, desde el segundo 143 al segundo 147.

Ilustración 29

4.2.26 26ª Sección (Ilustración 28)

Desde el segundo 147 hasta el segundo 149.5 se produce otro pequeño movimiento hacia delante con un considerable alabeo y cabeceo pero no lo suficiente como para provocar nuevas imágenes.

Ilustración 30

4.2.27 27ª Sección (Ilustración 29)

Del segundo 149.5 hasta el 155.5 se produce un movimiento hacia atrás con algo de guiñada hacia la derecha y fuertes alabeos, que junto con encontrarnos en una zona algo más diferente produce una nueva imagen.

Ilustración 31

4.2.28 28ª Sección (Ilustración 30)

El dron se coloca en posición para aterrizar entre los segundos 155.5 y 159, y no se produce ninguna nueva imagen.

Ilustración 32

4.2.29 29ª Sección (Ilustración 31)

El dron aterriza en el mismo lugar de donde despegó, entre los segundos 159 y 166.5, no se producen nuevas imágenes demostrando una vez más la robustez del programa que incluso en un movimiento brusco como puede ser aterrizar no añade nuevas imágenes innecesarias.

Ilustración 33

4.2.30 30ª Sección

Por último el dron se mantiene quieto hasta la finalización de la grabación en el segundo 197, evidentemente sin producir nuevas imágenes.

4.3 Otras medidas

Adicionalmente vamos a realizar una medida del tiempo de ejecución del programa. Para ello creamos y usamos las variables de medida de tiempo, que se explicaron en el apartado de código. Primero medimos el máximo tiempo de procesamiento de un bucle, en nuestro video del dron, lo que nos daría el margen superior de tiempo, y el más limitante. Habrá que considerar este tiempo a la hora de establecer la velocidad de transmisión de imágenes para que no perdamos ninguna. En nuestro caso el tiempo máximo es de 0.036618 segundos. Además hemos medido el tiempo mínimo y es 0.0067 segundos. A pesar de los valores anteriores hay que considerar que los tiempos cambios en cada ejecución y que este programa ha sido ejecutado en un sistema operativo (Ubuntu) emulado con el programa VirtualBox.

4.4 Reflexión

A modo de conclusión sobre este apartado podemos decir que el programa es capaz de trabajar en un entorno "real" demostrando un comportamiento muy sólido frente a perturbaciones en el movimiento. Además es tremendamente eficaz en los puntos críticos del trayecto (cierres de bucle).

5 CONCLUSIONES

5.1 Conclusiones finales

Con todos los procesos realizados durante el documento podemos decir que hemos conseguido crear un sistema autónomo que a partir de unas imágenes proporcionadas en tiempo real es capaz de determinar si se ha estado allí antes. Esto es un importante paso para el desarrollo de futuras aplicaciones basadas en vehículos aéreos no tripulados, ya que sistemas como este son básicos para perfeccionar su funcionamiento. Pero no solo hemos conseguido realizar la detección de lugares anteriormente visitados sino que hemos usado novedosos sistemas tanto de detección como de procesamiento de las imágenes, consiguiendo un programa robusto y eficaz, capaz de soportar severas perturbaciones en las condiciones de uso. Además hemos almacenado exitosamente un trayecto o zona completa en una pequeña base de datos, tremendamente compacta y con suficiente información como para ayudar a montones de aplicaciones que requieran información de recorrido realizado.

Con esto podemos decir que este proyecto ha sido un éxito, sobre el que se podrán sostener desarrollos de futuras aplicaciones en el campo de la visión por computador para vehículos aéreos no tripulados. También se ha expuesto el uso y funcionamiento de un descriptor de imagen novedoso, el BRIEF. Demostrando además, su eficacia, que colaborara con la extensión de su uso, que podría llegar a sustituir a los más utilizados actualmente.

5.2 Perspectivas futuras

Este documento provee de una base sobre la que realizar montones de futuras aplicaciones.

Para empezar la ampliación del proyecto en sí iría principalmente orientada a la mejora de la detección de nuevas posiciones junto con la optimización del código. Para lo primero se podrían tomar muchas rutas siendo las más naturales las de aumentar el nivel de las imágenes, en el sentido de ampliar la información recibida. Esto se podría conseguir, por ejemplo, añadiendo nuevas cámaras que proporcionaran nuevas formas de ver una misma posición, esto pese a poder mejorar mucho la calidad de las detecciones podría complicar mucho los algoritmos y con toda seguridad daría lugar a un programa más lento, que en algunas aplicaciones será inadmisible. Otra opción es cambiar la cámara estándar por otra que proporcione más información u otra información distinta. Además se podrían añadir otro tipo de sensores, como sensores de proximidad, que podrían mejorar la calidad de las detecciones. Otra opción más sencilla seria simplemente aumentar la cantidad de bits del descriptor así como la resolución de las imágenes, esto aumentaría la calidad de las detecciones de una forma bastante sencilla. Aun así es difícil mejorar el programa por estos métodos sin perder velocidad por lo que habría que establecer el uso de la aplicación antes de empezar a desarrollarla por este camino.

Otra opción para mejorarlo es optimizar el código. Para ello habría que mirar los principales puntos donde se concentra el tiempo de procesamiento, como por ejemplo el bucle donde se busca si alguna imagen coincide. Este tipo de soluciones mejorarían el tiempo de procesamiento del sistema sin afectar a su calidad. Otra opción sería perder algo de calidad de detección, reduciendo por ejemplo el número de bits del descriptor BRIEF. Estas soluciones solo serían recomendables en el caso de que el tiempo de procesamiento fuera muy determinante.

REFERENCIAS

[1] K. Konolige, G. Grisetti, R. Kümmerle, W. Burgard, B. Limketkai, and R. Vincent, "Efficient sparse pose adjustment for 2D mapping," in *IEEE/RSJ 2010 International Conference on Intelligent Robots and Systems, IROS 2010 - Conference Proceedings*, 2010, pp. 22–29.

[2] M. Kaess, A. Ranganathan, and F. Dellaert, "iSAM: Incremental smoothing and mapping," *IEEE Trans. Robot.*, vol. 24, no. 6, pp. 1365–1378, 2008.

[3] M. Kaess, H. Johannsson, R. Roberts, V. Ila, J. Leonard, and F. Dellaert, "ISAM2: Incremental smoothing and mapping with fluid relinearization and incremental variable reordering," in *Proceedings - IEEE International Conference on Robotics and Automation*, 2011, pp. 3281–3288.

[4] R. Kümmerle, G. Grisetti, H. Strasdat, K. Konolige, and W. Burgard, "G2o: A general framework for graph optimization," in *Proceedings - IEEE International Conference on Robotics and Automation*, 2011, pp. 3607–3613.

[5] J. Sivic and A. Zisserman, "Video Google: a text retrieval approach to object matching in videos," *Proc. Ninth IEEE Int. Conf. Comput. Vis.*, no. Iccv, pp. 2–9, 2003.

[6] D. Nistér and H. Stewénius, "Scalable recognition with a vocabulary tree," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2006, vol. 2, pp. 2161–2168.

[7] G. Schindler, M. Brown, and R. Szeliski, "City-scale location recognition," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2007.

[8] M. Cummins and P. Newman, "FAB-MAP: Probabilistic Localization and Mapping in the Space of Appearance," *Int. J. Rob. Res.*, vol. 27, pp. 647–665, 2008.

[9] H. Bay, T. Tuytelaars, and L. Van Gool, "SURF: Speeded up robust features," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2006, vol. 3951 LNCS, pp. 404–417.

[10] C. Cadena, D. Gálvez-López, J. D. Tardós, and J. Neira, "Robust place recognition with stereo sequences," in *IEEE Transactions on Robotics*, 2012, vol. 28, no. 4, pp. 871–885.

[11] W. Maddern, M. Milford, and G. Wyeth, "Continuous appearance-based trajectory SLAM," in *Proceedings - IEEE International Conference on Robotics and Automation*, 2011, pp. 3595–3600.

[12] A. Oliva and A. Torralba, "Chapter 2 Building the gist of a scene: the role of global image features in recognition," *Progress in Brain Research*, vol. 155 B. pp. 23–36, 2006.

[13] A. Oliva and A. Torralba, "Modeling the shape of the scene: A holistic representation of the spatial envelope," *Int. J. Comput. Vis.*, vol. 42, no. 3, pp. 145–175, 2001.

[14] A. C. Murillo and J. Kosecka, "Experiments in place recognition using gist panoramas," in *2009 IEEE 12th International Conference on Computer Vision Workshops, ICCV Workshops 2009*, 2009, pp. 2196–2203.

[15] M. Calonder, V. Lepetit, C. Strecha, and P. Fua, "BRIEF: Binary robust independent elementary features," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2010, vol. 6314 LNCS, no. PART 4, pp. 778–792.

[16] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *Int. J. Comput. Vis.*, vol. 60, no. 2, pp. 91–110, 2004.

[17] R. W. Hamming, "Error Detecting and Error Correcting Codes," *Bell Syst. Tech. J.*, vol. 29, no. 2, pp. 147–160, 1950.

[18] C. D. Pilcher, J. K. Wong, and S. K. Pillai, "Inferring HIV transmission dynamics from phylogenetic

sequence relationships," *PLoS Medicine*, vol. 5, no. 3. pp. 0350–0352, 2008.

[19] K. Mikolajczyk and C. Schmid, "Performance evaluation of local descriptors.," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 27, no. 10, pp. 1615–1630, 2005.

[20] G. Hua, M. Brown, and S. Winder, "Discriminant embedding for local image descriptors," in *Proceedings of the IEEE International Conference on Computer Vision*, 2007.

[21] G. Shakhnarovich and T. J. Darrell, "Learning Task-Specific Similarity by," *Electr. Eng.*, vol. 146, no. 1, pp. 70–76, 2005.

[22] M. Ozuysal, M. Calonder, V. Lepetit, and P. Fua, "Fast Keypoint Recognition Using Random Ferns," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2010.

[23] V. Lepetit and P. Fua, "Keypoint recognition using randomized trees," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 28, no. 9, pp. 1465–1479, 2006.

[24] T. Tuytelaars and C. Schmid, "Vector quantizing feature space with a regular lattice," in *Proceedings of the IEEE International Conference on Computer Vision*, 2007.

[25] A. Gionis, P. Indyk, and R. Motwani, "Similarity Search in High Dimensions via Hashing," in *VLDB '99 Proceedings of the 25th International Conference on Very Large Data Bases*, 1999, vol. 99, no. 1, pp. 518–529.

[26] A. Torralba, R. Fergus, and W. T. Freeman, "80 million tiny images: A large data set for nonparametric object and scene recognition," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 30, no. 11, pp. 1958–1970, 2008.

[27] R. Salakhutdinov and G. Hinton, "Learning a nonlinear embedding by preserving class neighbourhood structure," *AI Stat.*, vol. 3, no. 1, pp. 412–419, 2007.

[28] B. Stroustrup, *The C++ Programming Language*. 1997.

[29] G. Bradski and A. Kaehler, *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Inc., 2008.

[30] "OpenCV | OpenCV." [Online]. Available: http://opencv.org/. [Accessed: 06-Feb-2016].

[31] "OpenCV." [Online]. Available: http://itseez.com/OpenCV/. [Accessed: 06-Feb-2016].

[32] M. Quigley, E. Berger, and A. Y. Ng, "Stair: Hardware and software architecture," in *AAAI 2007, Robotics Workshop*, 2007, pp. 31–37.

[33] A. Martinez and E. Fernández, *Learning ROS for Robotics Programming*. 2013.

[34] M. Quigley, K. Conley, B. Gerkey, J. FAust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Mg, "ROS: an open-source Robot Operating System," *Icra*, vol. 3, no. Figure 1, p. 5, 2009.

Anexo A: Glosario

BRIEF: Binary Robust Independent Elementary Features	xi
SURF: Speeded Up Robust Features	2
ROS: Robot Operating System	xi
OpenCV: Open Source Computer Vision	xi
SIFT: Scale-Invariant Feature Transform	2
3D: 3 Dimensions	5
GIST: librería en C	2
CPU: Central Processing Unit	6

Anexo B: Código completo

```
#include "ros/ros.h"
#include "std_msgs/String.h"

#include <stdlib.h>
#include <stdio.h>
#include <time.h>

#include <sstream>
#include <iostream>
#include <vector>

#include "opencv2/core/core.hpp"
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/features2d/features2d.hpp"
#include <opencv2/nonfree/features2d.hpp>
#include <opencv2/legacy/legacy.hpp>

#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/image_encodings.h>
#include <sensor_msgs/CameraInfo.h>


using namespace std;
using namespace cv;
```

```
struct imagen
{
    int id;
    Mat img_original;
    Mat img_red;
    Mat img_descriptor;
    struct imagen * img_sig;
};

typedef struct imagen *Tlista;
Tlista principio;



void insertarFinal(Tlista &lista, int valor , Mat original , Mat red , Mat
descriptor)
    {
    Tlista t, q = new(struct imagen);
    q->id  = valor;
    original.copyTo(q->img_original);
    red.copyTo(q->img_red);
    descriptor.copyTo(q->img_descriptor);
    q->img_sig = NULL;
    if(lista==NULL)
        {
        principio = q;
        lista = q;
        }
    else
        {
        t = lista;
        while(t->img_sig != NULL)
            {
            t = t->img_sig;
            }
        t->img_sig = q;
        }
    }
```

```
float busca_conincidencia (Tlista &lista , Mat descriptor , Mat imagen , int id)
    {
    float aux = 0;
    float dist_hamm_min = 500;
    float dist_hamm = 0;
    int id_match = 0;
    Mat img_aux;
    Mat negro = imread("/home/victor/Descargas/extra_image.jpg");
    Tlista t = new(struct imagen);
    if(lista==NULL)
        {
        aux = 0;
        }
    else
        {
        t = lista;
        while(t != NULL)
        {
        dist_hamm = norm(t->img_descriptor,descriptor,NORM_HAMMING);
        if (dist_hamm < dist_hamm_min)
            {
            dist_hamm_min = dist_hamm;
            id_match = t->id;
            img_aux = t->img_original;
            }
        t = t->img_sig;
        }
        if (dist_hamm_min > 90)
            {
            aux = 0;
            imshow("view", imagen);
            imshow("bbdd", negro);
            }
        else
            {
            aux = id_match;
            imshow("view", imagen);
            imshow("bbdd", img_aux);
            }
        }
        return aux;
```

```
}



void imageCallback(const sensor_msgs::ImageConstPtr& msg)
    {
    static int count=0;
    static Tlista lista = NULL;
    static float aux = 1;
    imagen actual;
    cv_bridge::CvImagePtr img_ptr;
    try
        {
        img_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::BGR8);
        img_ptr->image.copyTo(actual.img_original);
        resize( actual.img_original , actual.img_red , cv::Size(
actual.img_original.cols/4 , actual.img_original.rows/4 ) );
        actual.id=count;
        vector<cv::KeyPoint> kpts1;
        BriefDescriptorExtractor extractor(32);
        kpts1.push_back(cv::KeyPoint(40, 40, 30));
        Mat descriptor1;
        extractor.compute(actual.img_original, kpts1, descriptor1);
        actual.img_descriptor = descriptor1;
        if (count == 0)
            {
            count++;
            insertarFinal( lista , count , actual.img_original ,
actual.img_red,actual.img_descriptor );
            }
        else
            {
            aux   =   busca_conincidencia(lista   ,   actual.img_descriptor   ,
actual.img_original , count);
            if (aux == 0)
                {
                count++;
                insertarFinal( lista , count , actual.img_original ,
actual.img_red , actual.img_descriptor );
                }
            }
        }
```

```
  catch (cv_bridge::Exception& e)
    {
    ROS_ERROR("Could not convert from '%s' to 'mono8'.", msg->encoding.c_str());
    }
}

int main(int argc, char **argv)
    {
    ros::init(argc, argv, "listener");
    ros::NodeHandle nh;
    namedWindow("view");
    namedWindow("bbdd");
    startWindowThread();
    image_transport::ImageTransport it(nh);
    image_transport::Subscriber sub = it.subscribe("/euroc5/cam0/image_raw", 1,
imageCallback);
    ros::spin();
    destroyWindow("view");
    destroyWindow("bbdd"); //se ha añadido
    }
```