UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

GRADO EN INGENIERÍA ELECTRÓNICA Y AUTOMÁTICA INDUSTRIAL

IMPLEMENTACIÓN DE UNA INTERFAZ EN GUI-MATLAB PARA EL ESTUDIO DE LA LOCALIZACIÓN VISUAL EN ROBÓTICA MÓVIL MEDIENTE TRATAMIENTO DE PUNTOS CARACTERÍSTICOS

TRABAJO FIN DE GRADO

Septiembre – 2022

AUTORA: Raquel Arias Montoro

DIRECTOR: David Valiente García

RESUMEN

En este trabajo se presenta una interfaz visual implementada en una Guide de MATLAB, para el estudio de la localización visual a partir de fotografías tomadas con una cámara omnidireccional, mediante el tratamiento de puntos característicos. Se ha desarrollado un algoritmo de localización visual para el testeo con diferentes tipos de puntos característicos. Permitiendo este la obtención de la pose relativa, así como los tiempos computacionales, la precisión de los cálculos y la cantidad de puntos característicos encontrados. Todos ellos han sido analizados posteriormente para obtener unos resultados y comparaciones entre los diferentes tipos de puntos usados.

Se detalla el desarrollo de la Guide implementada en MATLAB como interfaz visual. La cual ha sido generada para facilitar el uso del algoritmo de localización visual y la toma de los datos obtenidos, pudiendo verse estos de forma gráfica.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

1. INTRODUCCIÓN

1.1 JUSTIFICACIÓN DEL PROBLEMA

El siglo XXI llega con grandes avances en la robótica que hacen que la misma se presente como algo cotidiano en las vidas de estas generaciones. A día de hoy es común ver robots industriales realizando cualquier tipo de trabajo mecánico, incluso de precisión. Robots que colaboran en los laboratorios farmacéuticos, en quirófanos o en las actividades diarias para facilitar la vida de las personas. Con la robótica móvil todos estos puntos van mucho más allá, permitiendo su aplicación en tareas como la búsqueda y desactivación de minas, la investigación espacial, e incluso su uso para rescates en lugares de difícil acceso. Todas estas aplicaciones tienen un punto el común, facilitar y mejorar la vida de las personas humanas, así como realizar tareas que a nosotros nos resultarían imposibles [1].

Un robot autónomo móvil es capaz de moverse en el entorno, ya sea conocido o completamente desconocido, y a su misma vez realizar una tarea, todo esto de forma totalmente autónoma. Esta autonomía les da la capacidad de planificar y seguir un camino a través del entorno de manera óptima, evitando obstáculos y calculando su ubicación dentro del mapa [2]. Pero esta autonomía también supone una problemática a la hora de determinar su posición y generar una trayectoria de movimiento en espacios de los que no se tiene conocimiento alguno. Con este objetivo, se han utilizado varios tipos de sensores como GPS, IMS, datos láser 3D y codificadores de rueda (odometría de robot) para adquirir información válida para calcular una estimación [3].

La utilización de cámaras para estas tareas cada vez es más común, debido entre otras a su bajo coste y facilidad de uso. A su misma vez ofrecen información de forma clara y ordenada lo que es esencial en la visión por computador o visión artificial, que requiere de este conjunto de herramientas y algunos métodos para obtener, procesar y después analizar imágenes del mundo real con la finalidad de que puedan ser tratadas por un ordenador. Existen muchos tipos de cámaras utilizadas con estos fines, las cámaras omnidireccionales son una de las más extendidas y usadas en los últimos años,

ya que permiten obtener un amplio campo de visión (CDV) que permite una mejor vista de toda la escena y capturar en una única imagen todo el entorno.

En todo el mundo de la robótica, una de las principales necesidades es obtener una localización muy concreta y exacta. Pero obtener la localización de un robot no es tarea fácil, debido a la distorsión de las imágenes tomadas. La distorsión depende del tipo de cámara y lente utilizada. Una de las lentes más usadas, por su capacidad de capturar un espacio más amplio en una sola imagen, son las lentes de ojo de pez (fisheye), pero estas a su misma vez sufren de una gran distorsión de la realidad capturada. A la hora de realizar cálculos de cualquier tipo con estas imágenes hay que tener en cuenta esa distorsión, para ello es necesario y de suma importancia encontrar calibraciones robustas. La calibración de una cámara permite determinar los parámetros que explican la proyección de un objeto tridimensional sobre el plano de la cámara. Logrando con estas calibraciones caracterizar y eliminar los efectos de la gran distorsión generada en las imágenes de la cámara. Pero por desgracia a día de hoy, es muy complicado obtener una buena calibración y por ello podemos decir que es una de las principales problemáticas que encontramos con las cámaras omnidireccionales.

Con la odometría visual surge la necesidad de crear entornos visuales de manejo sencillo, con capacidad de alojar imágenes y realizar cálculos de localización y mapeo. Facilitando así los estudios y cálculos con imágenes omnidireccionales utilizadas en la visión por computador y la localización de robot móviles. Siendo de gran interés y utilidad su aplicación en la docencia. Ya que permitirían a los estudiantes aprender de forma fácil, clara y muy visual cómo funciona la localización por imágenes. Pudiendo utilizarse este tipo de aplicaciones o interfaces en prácticas de asignaturas del ámbito de la robótica y visión artificial.

Las interfaces visuales cada día son más utilizadas en el entorno educativo ya que ofrecen la facilidad de comprensión unido en numerosas ocasiones a un entretenimiento extra integrado en estas aplicaciones o interfaces. Encontrando ejemplos interfaces visuales educativas para aprender idiomas, conocer el sistema solar en profundidad incluso para el cálculo y aprendizaje de ecuaciones diferenciales o programación.

1.2 OBJETIVOS

El objetivo principal es obtener una interfaz visual para alojar bases de datos de imágenes tomadas con cámaras omnidireccionales para su posterior procesamiento y obtención de la localización y mapeado robótico.

Se escoge la Guide de MATLAB para la realización de esta interface visual, teniendo en cuenta su sencillez y facilidad de comprensión, así como la versatilidad que esta ofrece y permisibilidad de modificación en cualquier momento para añadir mejoras o procesos no implementados. Otro punto positivo es el conocimiento de los estudiantes de este ámbito del entorno de MATLAB y la facilidad que presenta a la hora de la elección por script evitando así la instalación de ninguna otra plataforma o librería.

Objetivos particulares llevados a cabo durante el proyecto:

- Obtener unas bases de datos de imágenes en distintas estancias, para su posterior introducción en la interfaz creada y análisis de ellas.
- Desarrollar un algoritmo de localización visual para testeo con distintos tipos de puntos característicos.
- Desarrollar algoritmo para obtención de resultados temporales, de precisión, distancia y número de puntos, para la comparación y estudio de ellos.
- Integrar todos los desarrollos anteriores sobre la interfaz Guide de MATLAB, permitiendo visualizar de forma práctica todos los puntos mencionados.

1.3 CONTENIDO DEL TRABAJO

En este apartado se presentan los apartados y organización del trabajo, con el fin de dar una idea general sobre el contenido de cada capítulo.

1. Introducción: se presenta la problemática de los robots móviles y sus soluciones.

2. Estado del arte: se desarrolla el origen de este campo y cuál es su situación actual.

3. Calibración: explicación sobre la calibración y su problemática en cámaras omnidireccionales.

4. Obtención de la localización. Odometría visual (VO): se expone la odometría visual y los pasos que se llevan a cabo.

5. Aplicaciones con interfaces visuales: explicación de las interfaces visuales y sus usos.

6. Implementación: se explica el código realizado y la funcionalidad de cada parte de este.

7. Resultados: una vez programada la Guide se extraen los datos de numerosas ejecuciones y se analizan.

8. Conclusiones y líneas futuras: valoración crítica del trabajo donde se exponen las conclusiones extraídas de los datos obtenidos.

9. Bibliografía: referencias bibliográficas.

10. Anexos: documentos adjuntos donde visualizar el código programado.

2. ESTADO DEL ARTE

2.1 ROBÓTICA MÓVIL

Un robot móvil se puede definir como un sistema mecánico capaz de moverse en su entorno de forma autónoma [4]. Para ese propósito, debe estar equipado con:

- Sensores, los cuales se encargan de percibir la información externa.

- Actuadores, son dispositivos capaces de transformar cualquier tipo de energía en energía mecánica, por lo que son los encargados de generar el movimiento.

- Una inteligencia, se considera inteligencia artificial al conjunto de algoritmos que hace posible, recolectando la información de los sensores, mandar una orden a los actuadores para generar el movimiento.

Los robots móviles son capaces de moverse de forma autónoma y segura sin dejar de tener un propósito o tarea asignada.

Aun que pensamos que la robótica móvil es algo muy novedoso, la realidad es que lleva años utilizándose con fines de investigación. William Gray Walter desarrolló el primer AMR a fines de la década de 1940 y principios de la de 1950. Hizo dos prototipos exitosos, Elmer y Elsie [5]. Fueron conocidos con el nombre de las “tortugas” por su forma y movimiento y fueron principalmente destinados a la investigación.

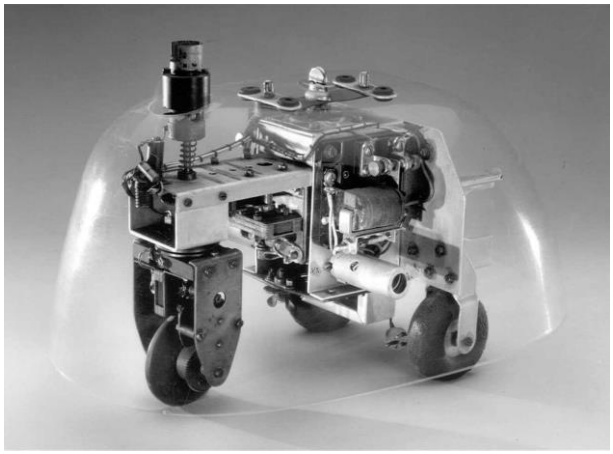


Figura 1: Primer robot móvil creado por Grey Walter en 1948

A principio de los 90 se creó el primer robot comercial, cuya función era desplazarse por un hospital portando diferentes cosas. Aun que fue un gran avance, este robot todavía no contaba con cámaras que hicieran posible su desplazamiento por cualquier lugar completamente desconocido.

Figura 2: Robot Help Mate, primer robot comercial

Pero no fue hasta más tarde cuando se comenzaron a utilizar cámaras como sensores, que permitían al robot tener una imagen de todo el entorno que los rodeaba y por tanto llegar a ser completamente autónomos en cualquier espacio.

2.2 VISIÓN POR COMPUTADOR

La visión por computador, también conocida como visión artificial, es una disciplina que consiste en adquirir, procesar, analizar y comprender imágenes de la realidad con el fin de producir información numérica o simbólica que pueda ser tratada por un ordenador.

Cada día aparecen más aplicaciones donde es usada. Una de las grandes áreas de aplicación es la robótica. Esto se debe a que con esta visión artificial, que pretende emular la capacidad visual del ser humano, podríamos dotar al robot de la capacidad sensorial más necesaria para el reconocimiento de un entorno, la cual les permitiría realizar muchas tareas incluso su movilidad por cualquier espacio.

Son dos partes fundamentales las que lo forman: el Sistema de adquisición de imágenes (*hardware*) y el Sistema de tratamiento de imágenes (*sofware*) [6]. El primero de ellos podemos entenderlo como el proceso de formación y captación de imágenes, mientras que el tratamiento de imágenes podemos dividirlo en 4 etapas:

- Procesamiento: empleando filtros o aumentando el contraste se elimina el ruido de la imagen para mejorar su calidad.
- Cuantificación: se identifica el objeto de estudio dentro de la imagen.
- Muestreo: para la extracción de característica.
- Interpretación: podemos entenderla como la clasificación de los datos obtenidos.

A finales de la década de los 60 y principios de los 70 se crearon los primeros robot con visión artificial, Shakey [7] y Stanford Cart [8].

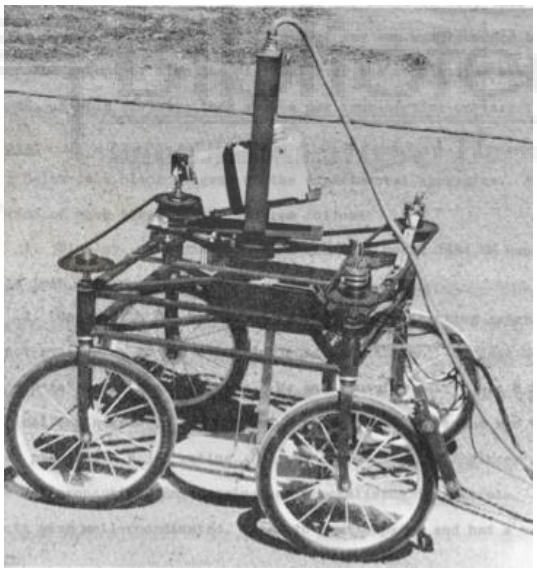


Figura 3: Stanford Cart

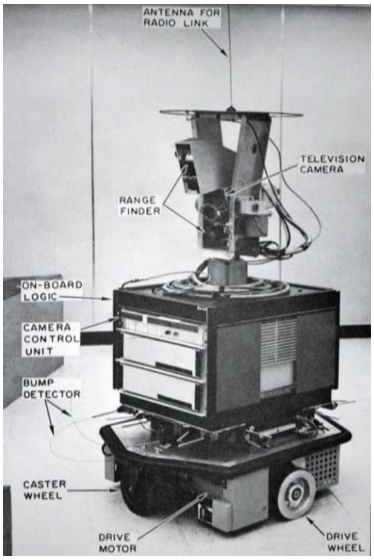


Figura 4: Shakey

2.3 SLAM VISUAL

SLAM (Simultaneous localization and mapping) es una forma de que un robot se localice en un entorno desconocido, mientras construye de forma incremental un mapa de su entorno [9]. SLAM se ha aplicado en numerosos estudios con diferentes tipos de sensores como sensores de infrarrojos o escáneres LASER. Debido el alto coste que implicaba el uso de estos sensores y unido a la gran cantidad de información que proporcionan los sensores de video, ha incrementado el interés en el SLAM basado en imágenes. Este tipo de SLAM es conocido como V-SLAM (Visual simultaneous localization and mapping).

Aunque puedan llegar a confundirse, la VO y el SLAM tienen una importante diferencia. La VO se encarga de estimar de forma incremental la trayectoria de la cámara, pose tras pose. El SLAM sin embargo tiene como objetivo obtener la estimación global, recogiendo la trayectoria y generando el mapa. Por este motivo es necesaria una consistencia global, que se logra cuando el robot reconoce un área, lo que significa que esta ha sido previamente mapeada, generando el cierre del bucle. De esta forma logramos una mejor estimación gráfica.

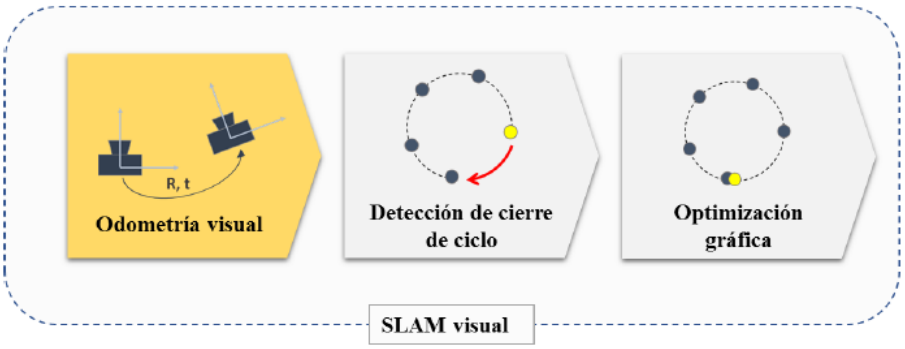


Figura 5: Gráfico de SLAM visual (odometría visual, detección de cierre de ciclo y optimización grafica).

2.4 SENSORES DE VISIÓN

En el mundo de la robótica móvil los sensores son algo esencial e imprescindible para dotar de autonomía a un robot. De esta forma y mediante el tratamiento de la información obtenida por los sensores es posible conocer la posición de un robot y permitir su movimiento sin colisión.

Los sensores se pueden clasificar de numerosas formas, en robótica móvil lo más común es dividirlos en sensores propioceptivos y exteroceptivos. Los sensores propioceptivos funcionan con la percepción del estado interno del dispositivo, por ejemplo la posición, velocidad angular, temperatura o nivel de batería. Estos sensores tienen como desventaja la acumulación de errores, por lo que no son del todo eficientes. Mientras que los sensores exteroceptivos se utilizan para detectar el entorno en que se encuentran, como por ejemplo los sensores de proximidad o de visión. Dentro de este tipo de sensores algunos de los más conocidos y utilizados son el sensor laser, el sónar, el GPS y las cámaras.

2.5 CÁMARAS

Al pensar en visión, lo primero que nos viene a la cabeza es la interpretación de imágenes similares a las tomadas por nuestros ojos. Los sensores de visión se basan en la formación de imágenes al enfocar la luz sobre una superficie sensible a ella, obteniendo así una captura de la realidad similar a la del ojo humano.

La cámara tiene un importante papel en la visión por computador y la fotogrametría, que se encarga de representar escenas 3D en una imagen 2D. Presentan numerosas ventajas en comparación con otros sensores, como el bajo coste, su facilidad de uso y sus dimensiones y peso, así como la cantidad de información que aportan y la alta precisión de la información. Pero también tienen algunas desventajas como la dificultad computacional para procesar las imágenes y extraer datos.

Se pueden obtener diferentes cámaras variando estos tres elementos: (1) la geometría de la superficie, (2) la distribución geométrica y las propiedades ópticas de los fotorreceptores, y (3) la forma en que la luz se recolecta y proyecta sobre la superficie (lentes simples o múltiples, o tubos como en ojos compuestos) [10].

En odometría visual se pueden usar varios tipos de cámara, como estéreo, monocular, estéreo o monocular omnidireccional y cámaras RGB-D [11]

2.5.1 Cámaras estereoscópicas

Una cámara estereoscópica es una cámara binocular, las cuales se caracterizan por tener dos lentes con sensores de imagen separados para cada una de las lentes. Son nombradas cámaras estereoscópicas debido a la visión estereoscópica humana en 3D. Son cámaras capaces de capturar imágenes (fotografías) en tres dimensiones. Gracias a esto permiten obtener información sobre la profundidad, algo que no es posible en cámaras monoculares.

Figura 6: a) Cámara estéreo binocular; b) Cámara estéreo omnidireccional.

La visión binocular humana, produce dos imágenes, una para cada ojo, que posteriormente se mezclan en el cerebro creando así la imagen 3D que tenemos de la realidad. Las cámaras estereoscópicas intentan imitar este comportamiento del ojo humano, utilizando dos objetivos que captan la fotografía en el mismo instante, y como resultado se obtienen las imágenes 3D.

Sin embargo, las cámaras binoculares resultan más complicadas de calibrar que las cámaras monoculares. Esto da lugar a errores de calibración mayores que afectan en el proceso de estimación de la posición o el movimiento. Además, de ser más caras y necesitar de un mayor esfuerzo computacional ya que se deben procesar dos imágenes en lugar de una.

2.5.2 Cámaras monoculares

Las cámaras monoculares son más comunes, estas son usadas a diario en para variadas aplicaciones como los teléfonos móviles. Además de su bajo coste y la facilidad de uso, es menos compleja su calibración. Disminuyendo así el error generado por la calibración.

Por lo contrario, los sistemas de visión monocular sufren de incertidumbre de escala [11]. Lo que significa que si la superficie no es regular, la escala de la imagen fluctuará y será difícil estimar el factor de escala.

Comparando los sistemas monoculares y binoculares, podemos decir que estas últimas son más útiles y eficientes en pequeña escala, donde el espacio entre las cámaras sería mucho más pequeña que la distancia a la escena de la que se toma la imagen. Cuando esto sucede la visión estéreo se vuelve ineficaz, por lo que es recomendable utilizar una cámara monocular.

Figura 7: Cámara monocular.

2.5.3 Cámaras omnidireccionales

La palabra omnidireccional está formada por *omni*, que significa "todo" y *direccional*, haciendo referencia así a que tiene un campo visual de todas las direcciones. Es decir, son cámaras de 360º de distancia focal que permite tomar imágenes omnidireccionales en un ángulo horizontal o en un campo visual. Este campo de visión mejorado puede lograrse utilizando sistemas catadióptricos, obtenidos mediante la combinación oportuna de espejos y cámaras convencionales, o empleando lentes de ojo de pez puramente dióptrico [12]. En los últimos años ha aumentado su uso en diferentes áreas como rastreo, localización, SLAM, odometría visual, etc.

Podemos diferenciar entre dos tipos de cámaras omnidireccionales, centrales y no centrales. Los sistemas de cámaras centrales omnidireccionales son aquellas que satisfacen la propiedad de un solo punto de vista efectivo, es decir que todos los rayos al reflejarse en el espejo se cruzan en un único punto. Se requiere de un grupo de cámaras convencionales apuntando a diferentes direcciones, cubriendo así los 360º. Estas son las llamadas policamaras. Al apuntar a un único punto de vista, permite calcular fácilmente las direcciones de los rayos de luz que ingresan a la cámara [13].

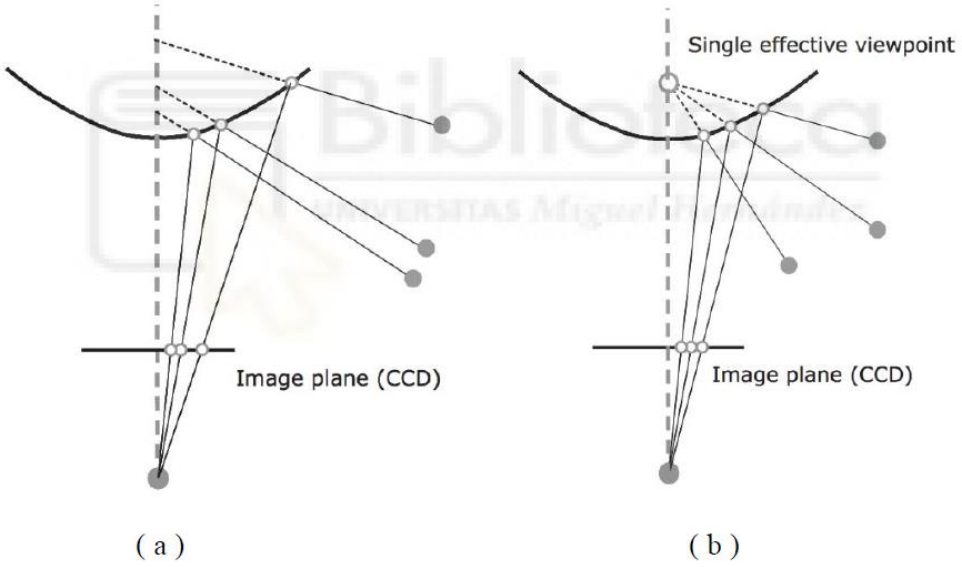


Figura 8: a) Cámara de visión no central, b) Cámara de visión central

Dentro de las no centrales podemos encontrar varios tipos. Un ejemplo de ellas son las cámaras giratorias, que se mueven en una trayectoria circular, tomando así con una única cámara la imagen de 360º. Los sistemas dióptricos son otro tipo de sistemas no centrales. Estos utilizan lentes de gran angular, como lentes de ojo de pez, combinados con cámaras convencionales.

Figura 9: Cámara omnidireccional

Los sistemas omnidireccionales dióptricos más utilizados en visión por computador y robótica son los sistemas hipercatadiópticos, los paracatadiópticos y las lentes de ojo de pez. El primero está compuesto por un espejo hiperbólico y una cámara en perspectiva. Mientras que los paracatadiópticos están compuestos por un espejo parabólico y una cámara ortográfica [13].

2.5.4 Cámara ojo de pez

Las cámaras con lente de ojo de pez son uno de los sistemas que permite capturar un amplio campo de visión. Estas lentes pueden proporcionar un campo de visión superior a 180º. De esta forma, si dotamos una cámara con dos lentes de ojo de pez y combinamos las imágenes podemos obtener un campo de visión de 360º.

El problema principal surge al combinar esas imágenes tomadas con más de una lente de ojo de pez por la variación de las condiciones de iluminación de cada imagen.

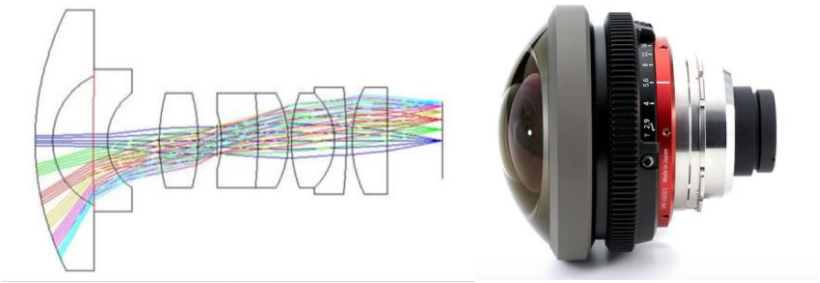


Figura 10: Lente ojo de pez

3. CALIBRACIÓN

El proceso de calibración de una cámara es muy necesario para obtener información en 3D de fotografías tomadas en 2D. En cualquier tipo de cámara es un paso imprescindible si pretendemos obtener medidas de la realidad a partir de una serie de fotografías tomadas [14]. Una buena calibración será determinante a la hora de posteriormente poder tomar medidas o localizar la imagen.

Lo que se pretende con la calibración es obtener los parámetros intrínsecos y extrínsecos para las características de las imágenes y la localización de la cámara en un entorno, respectivamente. Permitiendo, de esta manera, la transformación de información de un sistema de referencia a otro. Lo que es indispensable para la localización ya que así es capaz de encontrar la posición de la cámara dentro de un sistema de referencia global.

En cámaras omnidireccionales el método de calibración es bastante complejo, por la distorsión que las imágenes tienen, debido a que la superficie de la imagen es esférica y no plana como en las cámaras comunes. El modelo de calibración debería tener en cuenta el reflejo operado por el espejo en el caso de una cámara catadióptrica o la refracción causada por la lente en el caso de una cámara de ojo de pez [15].

Uno de los métodos más populares de calibración para las cámaras omnidireccionales es la toma de imágenes de una cuadricula plana en diferentes posiciones y orientaciones, con coordenadas de un objeto conocido tridimensional. Para este tipo de calibración cabe resaltar los métodos de Mei y Rives [16] y de Scaramuzza [16].

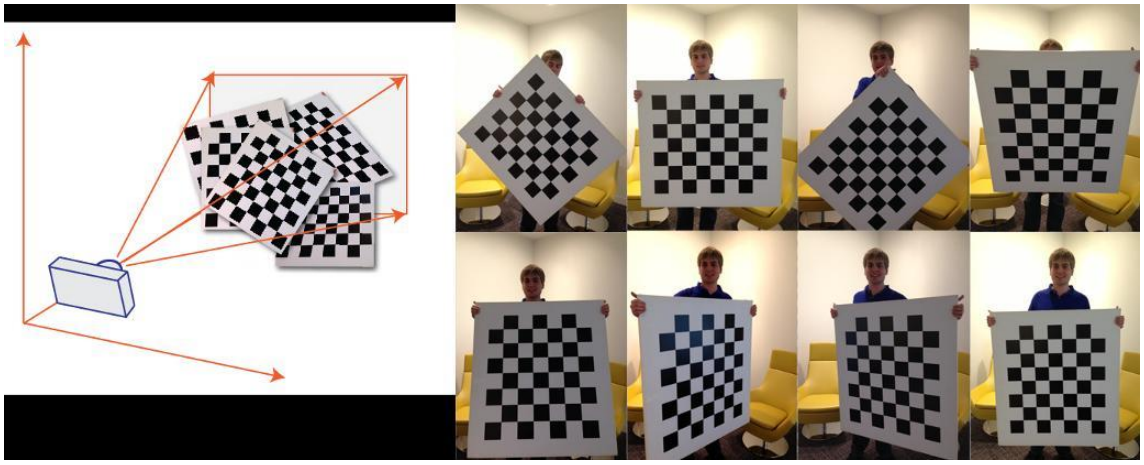


Figura 11: Ejemplo visual de calibración con modelo Mei

4. OBTENCIÓN DE LA LOCALIZACIÓN. ODOMETRÍA VISUAL (VO)

El término "odometría" se originó a partir de dos palabras griegas *hodos,* que significa "viaje" y *metron*, que significa "medida". Esta derivación está relacionada con la estimación del cambio en la pose de un robot (traslación y orientación) a lo largo del tiempo [11]. En concreto podemos decir que el término de odometría visual implica la determinación de la pose actual del robot y la trayectoria recorrida al recuperar una transformación de movimiento entre imágenes, que son captadas en cada pose atravesada por el robot [3]. Así mismo, esto permite generar un mapa de la trayectoria seguida.

VO es el proceso de estimación de pose de un agente (p. Ej., Vehículo, humano y robot) que implica el uso de solo un flujo de imágenes adquiridas de una sola cámara o de varias cámaras conectadas a él [16].

A lo largo de años se ha ido desarrollando esta técnica para la localización de robot a través de imágenes. Moravec introdujo y describió la idea de estimar la pose de un vehículo solo a partir de la información visual a principios de la década de 1980 [17]. Desde entonces hasta el 2000 fue la NASA quien continuó con las investigaciones en la preparación de la Misión a Marte del 2004. Fue entonces en 2004 cuando se definió el término de odometría visual como la estimación del movimiento a partir de la toma de medidas con un sistema de visión, muy similar a la odometría de las ruedas. Desde entonces la odometría visual ha ganado importancia, siendo utilizada para numerosas aplicaciones, algunas de las más comunes y conocidas son los vehículos terrestres o marinos, así como las misiones en otros planes.

Podemos clasificarla en dos grupos, según su método:

- Visión monocular: se utiliza una sola cámara, lo que reduce la complejidad, pero no es posible observar la escala del movimiento.
- Visión estéreo: utiliza más de una cámara, lo cual resuelve el problema de la escala y es capaz de comparar la profundidad entre imágenes.

Sea cual sea el método es importante que las imágenes se tomen bajo unas buenas condiciones de iluminación y similares en todas las imágenes. Siendo

imprescindible que exista un solapamiento entre las imágenes tomadas, para de esta forma poder extraer puntos característicos y compararlos entre imágenes.

La obtención de la localización por odometría visual consta de unas fases correlativas.

1. Obtener secuencia de imágenes.
2. Extracción de puntos característicos de cada imagen.
3. Obtención de los descriptores de los puntos característicos obtenidos.
4. Correspondencia entre los descriptores de dos imágenes.
5. Estimación del movimiento.

4.1 OBTENCIÓN DE LAS SECUENCIAS DE IMÁGENES

Las secuencias de imágenes a estudiar pueden ser tomadas con diferentes tipos de cámaras, como las mencionadas en el capítulo anterior. Con ellas generamos una base de datos a estudiar.

Es importante que todas las imágenes sean tomadas en condiciones similares de iluminación para minimizar el error en los siguientes puntos del proceso de localización mediante la odometría visual. Por este mismo motivo hay que tener en cuenta que cuanto menos móvil sea el escenario menos complejo resultará el proceso y evitara fallos.

4.2 EXTRACCIÓN DE PUNTOS CARACTERÍSTICOS Y OBTENCIÓN DE DESCRIPTORES

Tras la obtención de una secuencia de imágenes, continuamos en esta etapa con la extracción de puntos característicos de ellas. Estos puntos deben cumplir unas determinadas características, dependiendo de la combinación de detectores y descriptores de puntos característicos se encontraran un tipo de puntos.

Seguidamente, tras encontrar los puntos característicos de una imagen, es necesario obtener información relevante del entorno que rodea estos puntos

característicos. Esta información obtenida conviene que sea lo más simple y compacta posible, para posteriormente poder ser comparada fácilmente con los descriptores obtenidos de los descriptores obtenidos en diferentes imágenes.

Para extraer un descriptor se considera un área de la imagen centrada en ese punto característico obtenido. En esta zona o región se realizan diferentes operaciones para obtener unos valores o variables que describan la característica.

4.2.1 Detectores de esquina

Son los detectores de puntos cuyo método trata de encontrar las esquinas dentro de la imagen. Considerando esquina a todos los puntos en los que hay una intersección de dos o más bordes.

Algunos ejemplos son los detectores BRISK, HARRIS o FAST

4.2.2 Detectores de manchas

Entendemos en este caso como mancha a una región dentro de la imagen que se diferencia notoriamente de su entorno más cercano en términos de color, textura e intensidad.

Dentro de este tipo podemos encontrar los detectores SURF y KAZE

4.3 CORRESPONDENCIA ENTRE LOS DESCRIPTORES

La correspondencia entre los descriptores visuales es el proceso mediante el cual se comparan los puntos característicos obtenidos de cada imagen, pudiendo así localizar el mismo punto en diferentes imágenes. De esta manera, se podría realizar un seguimiento de un mismo punto a lo largo de una secuencia de fotografías.

La forma más utilizada para realizar esta tarea consiste en emparejar los puntos característicos comparando las coincidencias entre los vectores descriptores de las características visuales de diferentes fotografías. Es usual realizar esta comparación con la técnica de Machine Learning.

4.4 ESTIMACIÓN DEL MOVIMIENTO

Gracias a la comparación realizada en el paso anterior podemos llegar a estimar el movimiento y rotación de la cámara en cada una de las imágenes. Si suponemos que estas imágenes están siendo tomadas por un robot en movimiento, podríamos ir comparando las imágenes consecutivas y así generar la trayectoria descrita por dicho robot.

5. APLICACIONES CON INTERFACES VISUALES

Las interfaces gráficas de usuario, conocidas como GUI (*Graphical User Interface*), son programas informáticos que se utilizan como interfaces de usuario. Estas usan un conjunto de objetos gráficos e imágenes para representar de forma más sencilla e intuitiva la información y las acciones disponibles en la interfaz. Se usan principalmente para proporcionar un entorno visual sencillo que permite la comunicación del usuario con el sistema operativo del ordenador o máquina.

Surgen como la evolución de las interfaces de línea de comandos que se usaban para operar en los primeros sistemas operativos y computadoras y es una pieza fundamental en un entorno gráfico [18]. El objetivo de esta interfaces gráficas es que todo el código del *backend* de un sistema, necesario para su funcionamiento, quede representado de la forma clara y sencilla para los usuarios, simplificando así las tareas y acercando estos sistemas a toda clase de personas sin necesidad de tener una formación específica [19].

En la actualidad existen diversas aplicaciones con interfaces visuales, para facilitar sus funciones al usuario. Estas aplicaciones son utilizadas diversos campos como la educación, la medicina, la industria y el entretenimiento.

Un ejemplo de aplicación, que usamos casi a diario seria “Calendar”, una aplicación de calendario en la que un usuario puede introducir todo tipo de eventos, con diversos datos, de forma clara y sencilla. Posteriormente podremos ver los eventos introducidos de forma ordenada en la línea de tiempo de nuestro calendario.

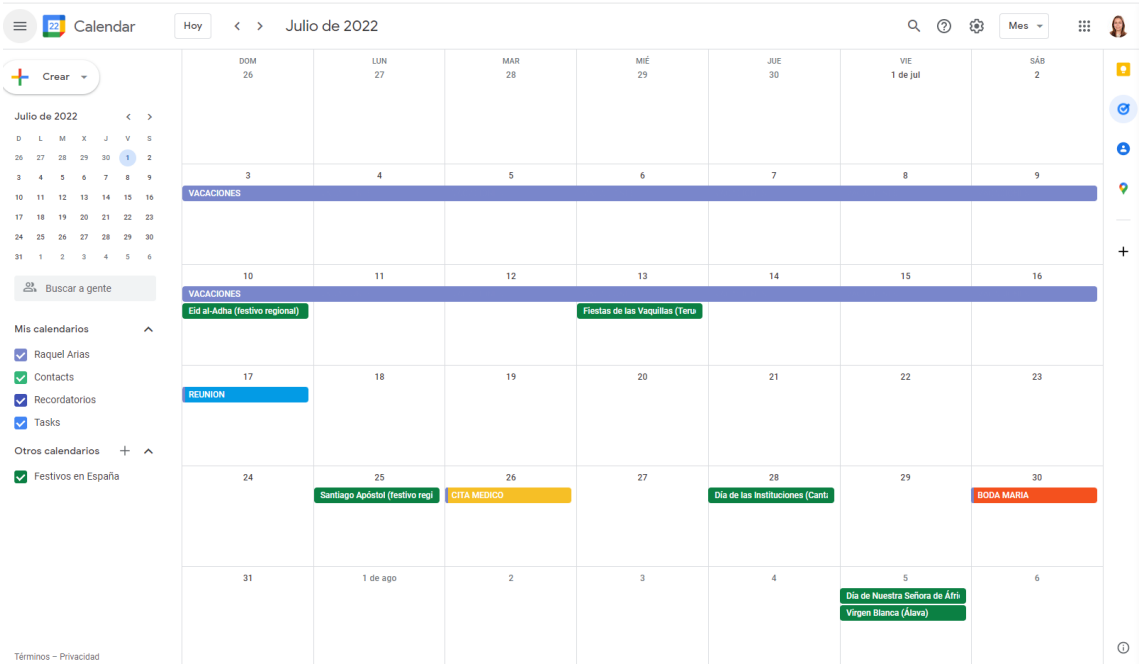


Figura 12: Google Calendar

Como también hemos comentado las interfaces visuales son cada vez más utilizadas en docencia, a diversos niveles. Ya desde una temprana edad se usan para facilitar el aprendizaje. Un ejemplo de ellas es el "Cocodile clips", programa con una interfaz gráfica sencilla para simular circuitos electrónicos y ver su comportamiento. Otra aplicación usada en educación y entretenimiento podría ser Scratch, una aplicación con una robusta interfaz visual, que en forma de juego se utiliza en educación para enseñar a los más pequeños a programar de forma básica.

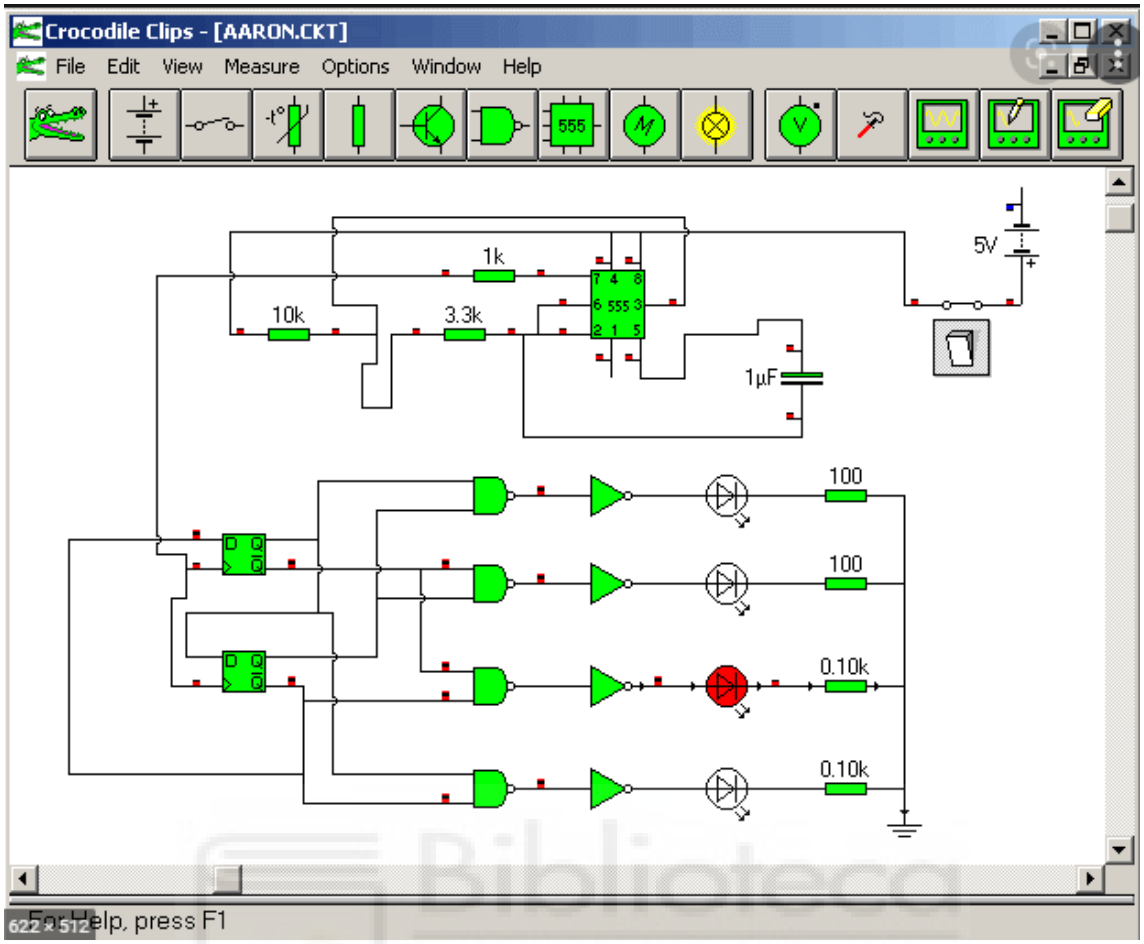


Figura 13: Cocodile Clips

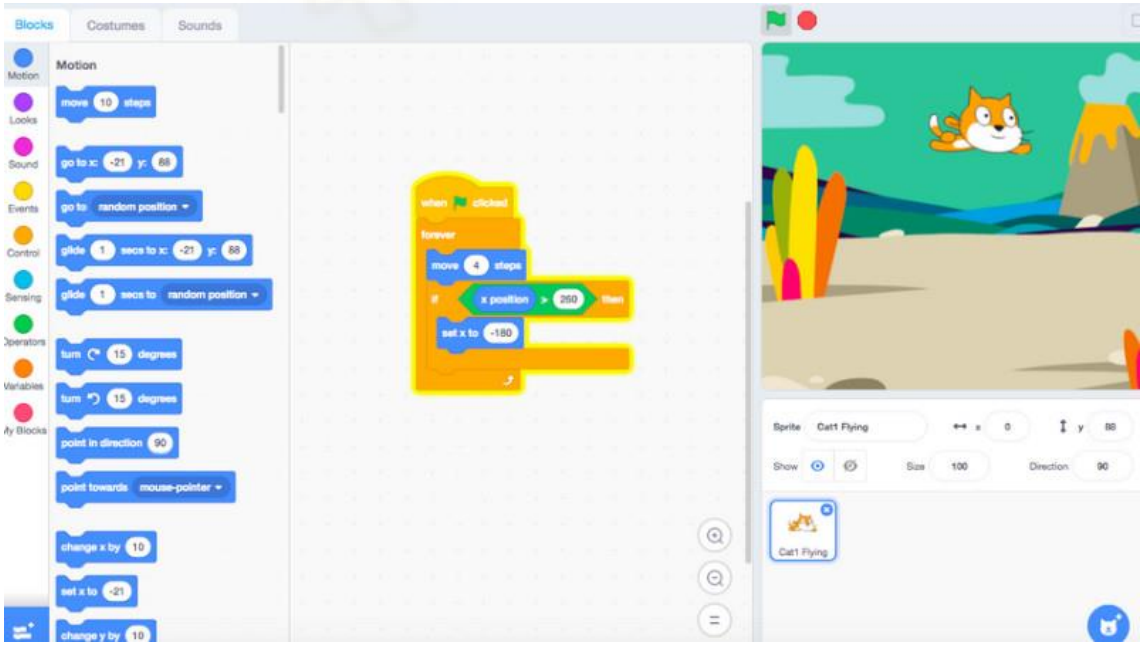


Figura 14: Scratch

A día de hoy uno de los campos en los que más se están desarrollando y mejorándose las interfaces visuales es la realidad virtual, utilizada sobre todo con fines de entretenimiento. Lo que se pretende con esto es proporcionar al usuario la máxima sensación de libertar en la interacción, mediante la simulación de entornas reales, procesos de interacción naturales y respuestas del sistema en tiempo real. Todos estos creados gracias a las interfaces visuales.

Estas interfaces pueden desarrollarse con diversos *frameworks*, programas y lenguajes de programación. Algunos de los más conocidos y utilizados son Java, C++ o PHP, pero existen muchos más. Por ejemplo, MATLAB, que aunque es un sistema de cómputo numérico, ofrece un lenguaje de programación propio, en el que vamos a desarrollar nuestra aplicación.

Un ejemplo ya creado y muy extendido de MATLAB seria Simulink. Este proporciona un editor gráfico y bibliotecas de bloques personalizables para modelar y simular sistemas dinámicos. Está integrado con MATLAB, lo que le permite incorporar algoritmos de MATLAB en modelos y exportar los resultados de la simulación a MATLAB para su posterior análisis [20].

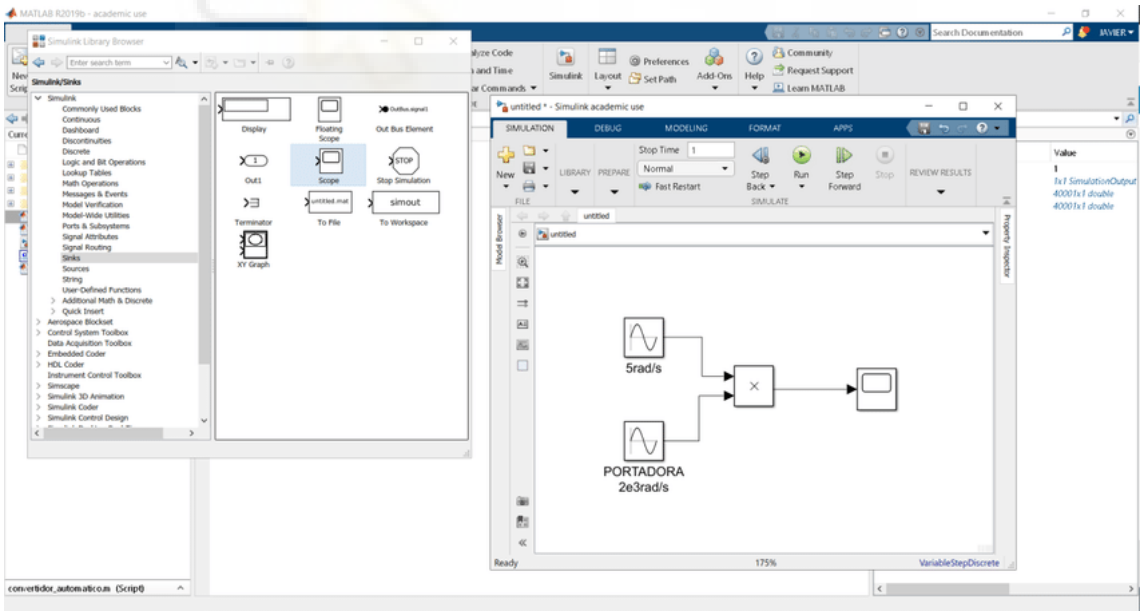


Figura 15: Simulink

6. IMPLEMENTACIÓN

A continuación vamos a explicar los procesos seguidos para la realización del trabajo y su posterior análisis. Comenzaremos haciendo una explicación sobre las imágenes tomadas para generar la base de datos con la que vamos a trabajar y continuaremos con la implementación de la Guide y los cálculos que esta realiza.

6.1 IMÁGENES

Las imágenes con las que se han realizado los estudios han sido tomadas en dos salas del edificio Innova del Campus Universitario Miguel Hernández de Elche, el salón de grados y la biblioteca. En cada una de las estancias se ha tomado una fotografía cada 40 cm siguiendo líneas horizontales y verticales y con una rotación de 0º. Generando así dos bases de datos:

- Biblioteca. Con un total de 176 imágenes, ordenadas teniendo de referencia los ejes X e Y al lado izquierdo de la puerta.
- Salón de grados. En este se han tomado 316 fotografías. Teniendo como referencia X e Y cero la segunda puerta.

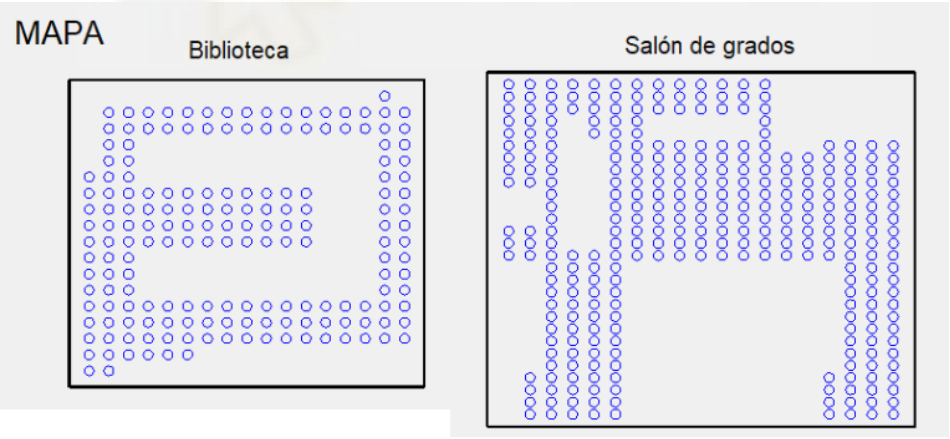


Figura 16: Mapas de las salas con puntos en los que se toman las fotografías.

Estas fotografías se han tomado con una cámara Garmin VIRB 360 [21]. Una cámara 360º compuesta por dos lentes con un campo de visión de 202 grados, pudiendo capturar así una esfera completa de fotos y videos en alta resolución.

Figura 17: Cámara Garmin VIRB 360.

La cámara permite elegir entre distintos modos, generando así diferentes imágenes: delantera, trasera, 360º y RAW. Las imágenes en este caso han sido tomadas con el modo RAW, que toma dos imágenes esféricas, una de la parte delantera y la otra de la posterior, pero estas imágenes no se juntan generando una única panorámica. Permitiendo así tratar cada imagen por separado.

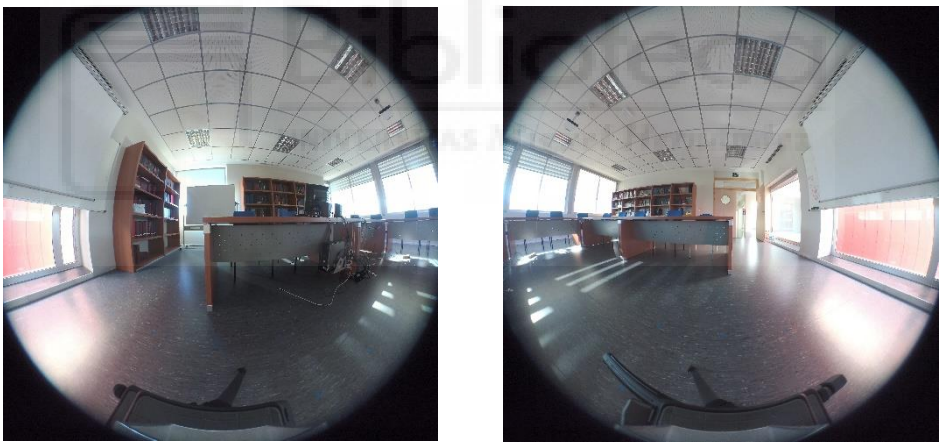


Figura 18: Fotografía de la biblioteca tomada con Garmin VIRD 360

Tras la toma de las fotografías de estas salas, y teniendo en cuenta que durante todo el estudio vamos a trabajar con ellas, hemos nombrado y guardado estas imágenes ordenadamente, facilitando así su uso. Estas imágenes han sido guardadas en un formato que expresa literalmente la sala y posición por nombre, quedando así ordenadas según sus coordenadas.

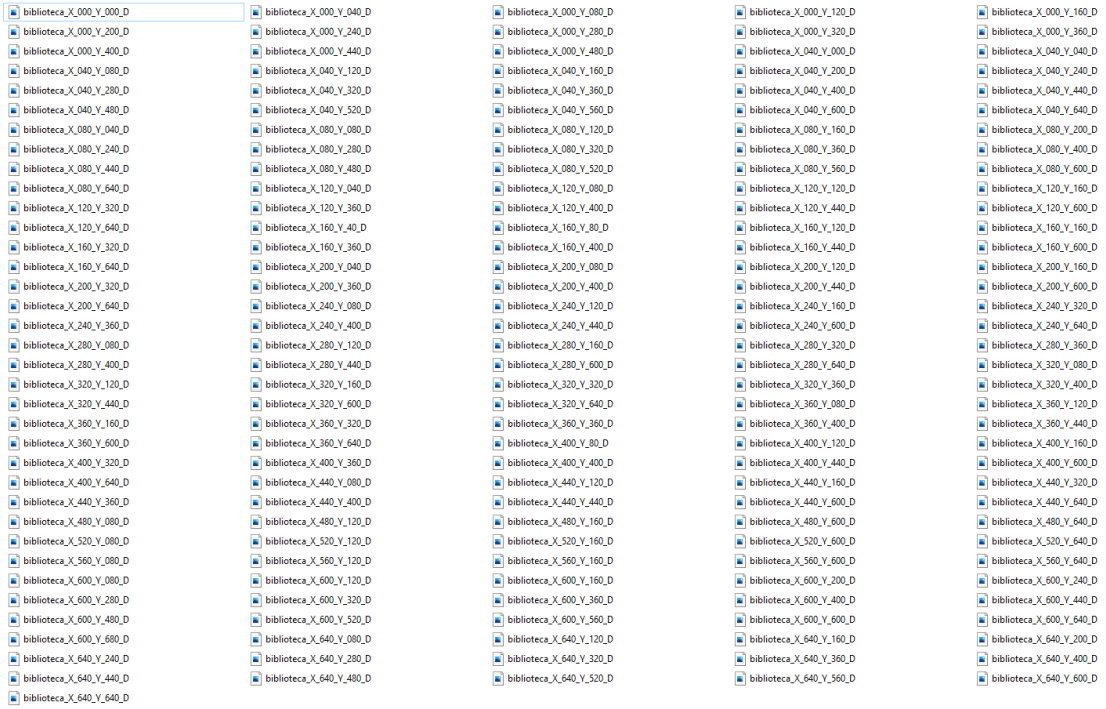


Figura 19: Captura de la carpeta de imágenes tomadas en la biblioteca

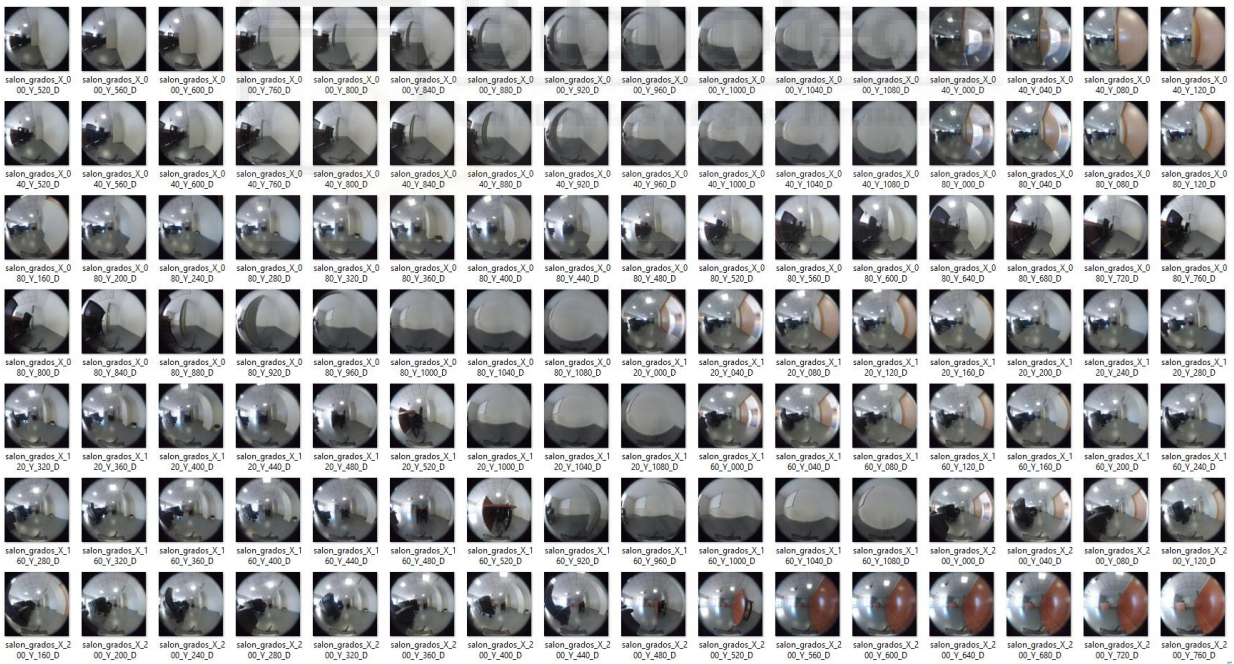


Figura 20: Captura de la carpeta de imágenes tomadas en el salón de grados.

6.2 GENERAR GUIDE

En el siguiente apartado se va a explicar de forma resumida la estructura del código. Se explicará de forma general las funciones, para poder entender cómo funciona la Guide creada y los cálculos que esta realiza. Así como, la cámara utilizada para tomar las fotografías con las que se ha generado la base de datos utilizada en el trabajo para ejecutar este código y comparar los resultados.

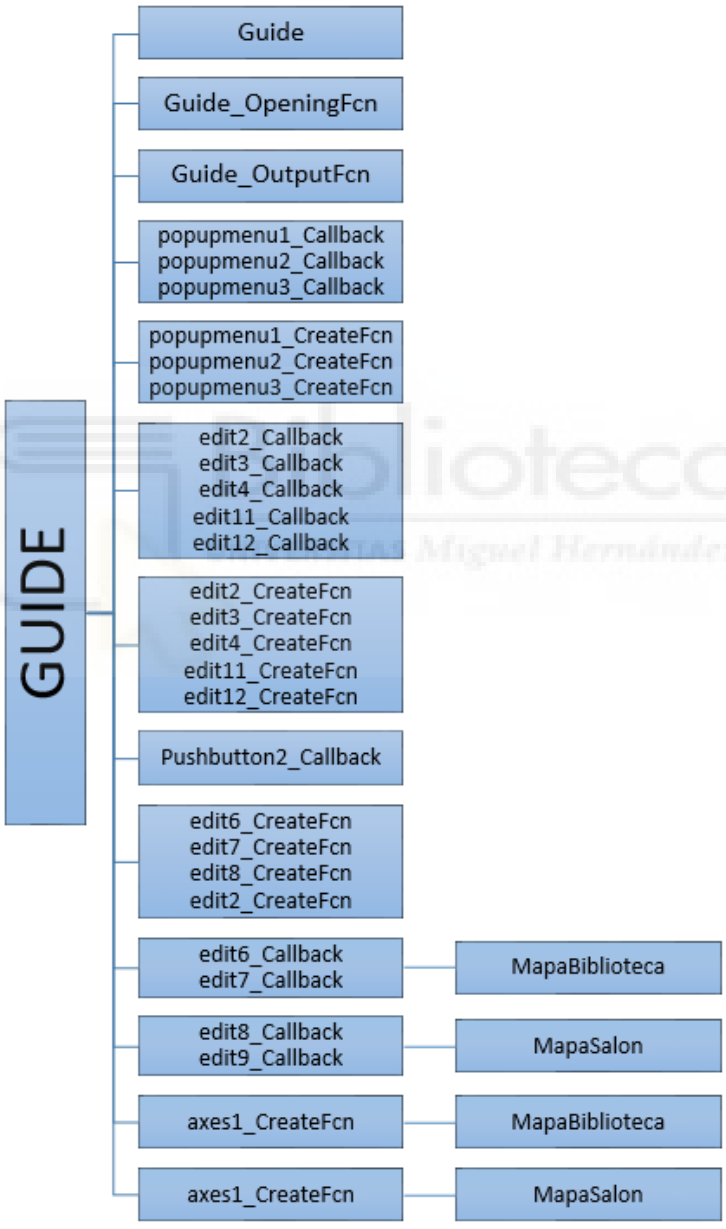


Figura 21: Diagrama esquemático del código

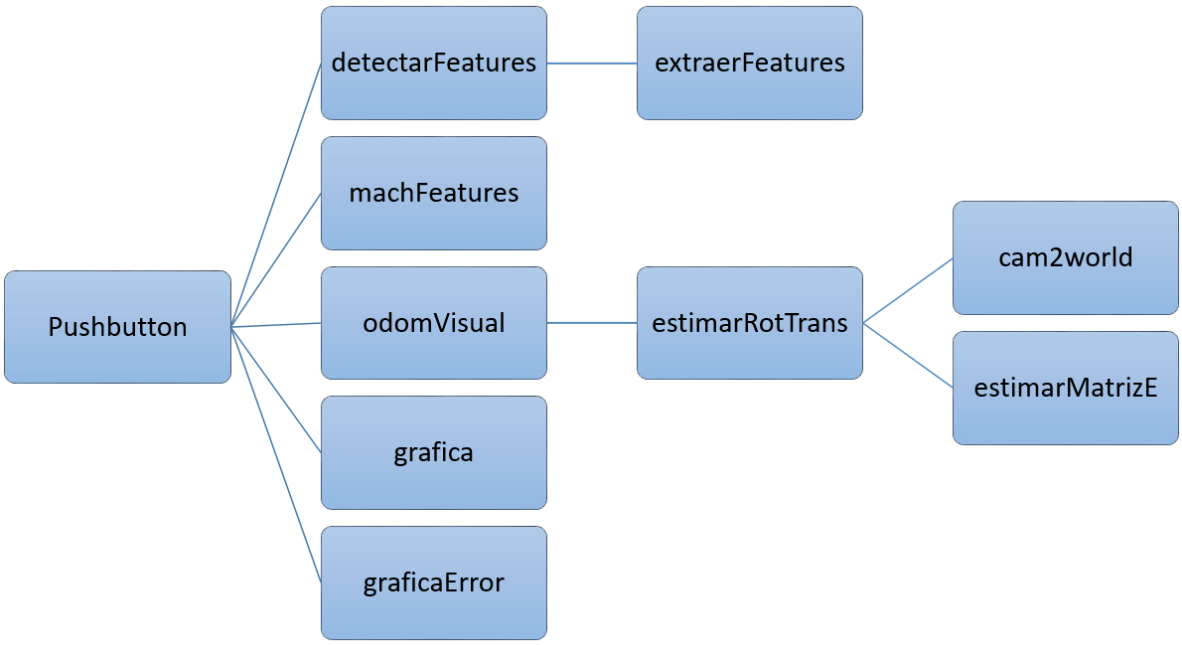


Figura 22: Diagrama esquemático del código implementado dentro de la función Pushbutton.

6.2.1 *Funcion Guide*

En primer lugar tenemos la función "Guide" la que ejecutamos al principio de todo el código como *function varargout = Guide (varargin)*. Esta función se encarga de crear una nueva Guide o en el caso de que esta ya exista, ejecutarla. Y gracias a la variable de salida "varargout", que es una variable de salida en una declaración de definición de función, permite que la función devuelva cualquier número de argumentos de salida.

6.2.2 *Guide_OpeningFcn*

Esta función se ejecuta automáticamente justo en el momento en que la Guide se hace visible. En esta función se cargan las bases de datos de imágenes, así como el *ground truth* y la calibración. También aquí señalamos todos los valores posibles para las variables que más tarde podremos modificar en la Guide y damos a estas un valor concreto con el que se inicializan.

Esta misma función volvemos a utilizarla seguidamente, pero en este caso dado unos valores de salida. Los valores de salida que daremos aquí es simplemente los objetos que queremos hacer visibles en la Guide al iniciar, así como los valores que queremos que muestren los *displays* antes de ejecutar.

6.2.3 *popupmenu*

Para generar todos los menús desplegables de nuestra Guide utilizamos las función *popupmenu_CreateFcn*. En nuestro caso hemos utilizado tres de este tipo de bloques, por lo que tenemos tres funciones similares para crear estos desplegables. En el primero de ellos podemos seleccionar la base de datos de imágenes, seguidamente tenemos otro desplegable en el que elegiremos el tipo de puntos con el que queremos calcular y finalmente podemos elegir entre dos opciones, comparar las dos imágenes que indiquemos o que se coparen todas las imágenes de la base de datos unas con otras.

Figura 23: Desplegables de la Guide

Junto con estas funciones encargadas de crear los desplegables tenemos las funciones *popupmenu_Callback*, encargada de recoger los datos seleccionados y generar los diferentes casos.

6.2.4 *edit*

Al igual que hemos visto en el apartado anterior para las listas desplegables, para los bloques de tipo editables tenemos una función para crearlos y otra en la que generamos el contenido.

Tenemos tres casillas editables que usaremos para introducir la *R*, que es el umbral de relación para rechazar coincidencias ambiguas, y otros dos para indicar las imágenes a comparar en el caso de seleccionar la opción comparar dos imágenes.

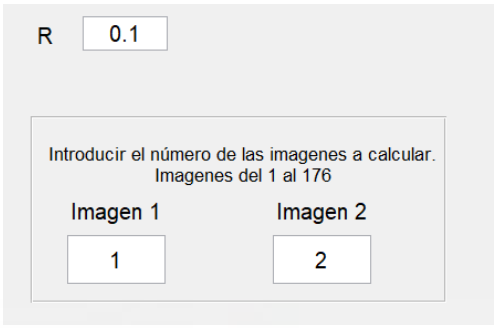


Figura 24: Bloques editables de la Guide

También podemos encontrar en el final del código otras cuatro funciones de *edit*. En ellas se crean las marcas que señalan la posición de las imágenes seleccionadas, en los mapas, que más tarde se mostraran en la función *axes*.

Figura 25: Bloques editables de datos de salida en la Guide

6.2.5 *pushbutton*

Estos serían los botones, en los que simplemente clicando podríamos realizar cualquier acción que indiquemos.

En nuestro caso hemos creado un único botón, este sería el botón “CALCULAR”. Clicando este botón indicamos que se comiencen a realizar los cálculos. Es por eso que todo el código creado para comparar las imágenes y realizar los cálculos se encuentra dentro de la función *Pushbutton_Callback*.

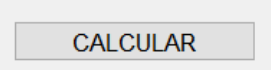


Figura 26: Botón calcular de la Guide

6.2.6 *axes*

Gracias a estos bloques podemos incluir, imágenes, mapas o graficas en la Guide.

Para generar los mapas de las salas en las que se tomaron las imágenes tenemos las funciones "*axes1_CreateFcn*" y "*axes2_CreateFcn*", en las que podemos encontrar las funciones llamadas *MapaBiblioteca* y *MapaSalon*. En estas funciones creamos un mapa de las estancias con los puntos en los que se tomaron las fotografías y una cruz que señala la imagen que hemos elegido en ese momento.

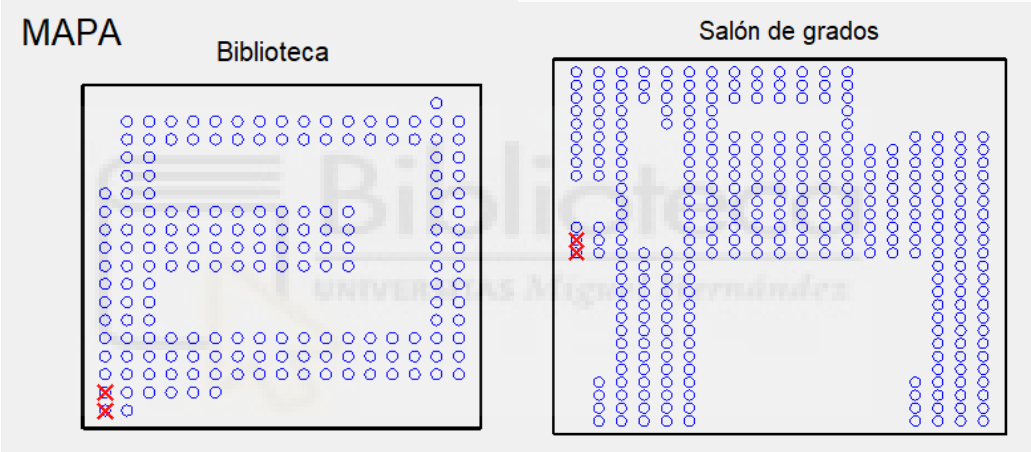


Figura 27: Mapas generados en la Guide

También usamos esta función para generar los histogramas con los datos tomados al comparar todas las imágenes entre ellas.

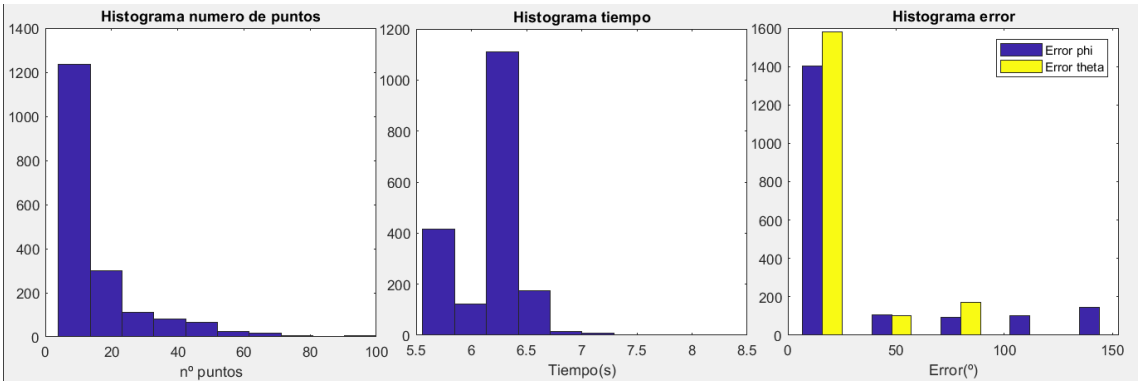


Figura 28: Graficas en la Guide

6.3 CÁLCULOS

Como comentamos en el apartado anterior, toda la parte de cálculo está dentro de la función *pushbutton*. Lo primero que hacemos en esta función es cargar las imágenes, las matrices del *ground truth* y los parámetros de calibración.

A partir de aquí podríamos decir que la función se separa en dos opciones, una de ellas para comparar dos imágenes y la otra que compararía todas entre ellas. Ambas opciones utilizan las mismas funciones, pero en el caso de querer comparar todas las imágenes estas funciones se repetirían en bucle.

Ahora si comenzaríamos el algoritmo de odometría visual. Comenzamos trasformando las imágenes a color a una escala de grises con la función *rgb2gray(readimage(handles.img,handles.num1)).*

6.3.1 Detectar puntos.

6.3.1.1 detectarFeatures

La siguiente función que tenemos es *[descriptores,validPoints]=detectarFeatures (I,Feature).* En esta función extraemos los puntos característicos que pasaremos a la función *extratFeatures*, para posteriormente devolver las características de estos puntos.

En función del tipo de puntos característicos seleccionado, se llama a una de las funciones incluidas en *Computer Vision Toolbox*.

Como argumentos de entrada tenemos: *I*, que son las imágenes de entrada en escala de grises y *Feature*, que es el tipo de punto característico que se pretende detectar

Los argumentos de son los descriptores en una matriz de MxN, donde *M* es el número de puntos y *N* la longitud de cada descriptor y la variable *validPoints*, que serán los puntos de los cuales se ha podido obtener su descriptor.

6.3.1.2 extratFeatures

Como hemos comentado en apartado anterior esta función se ejecuta dentro de *detectarFeatures* y se encarga de encontrar las características de cada punto. Teniendo como argumentos de entrada la imagen de la que se han extraído los puntos y estos puntos; y como salida una matriz de MxN con los descriptores.

6.3.2 Buscar correspondencias entre las dos imágenes.

6.3.2.1 machtFeatures

Esta función *indexPairs = matchFeatures(prevFeatures, actFeatures, 'MaxRatio', handles.R,'Unique',true* busca características coincidentes entre los descriptores de la función anterior.

La función toma como entrada las dos matrices de los descriptores anteriormente calculadas, el "*MaxRatio*" y el escalar lógico "*Unique*". El *MaxRatio*, llamado *R* en la Guide, también será un parámetro variable que nos permite especificar un umbral de relación para rechazar coincidencias ambiguas. "*Unique*", es un escalar lógico que siempre será verdadero para devolver solo coincidencias únicas entre los descriptores.

Esta función devuelve una matriz que contiene índices de las características con mayor probabilidad de correspondencia entre las dos matrices de características de entrada.

6.3.3 Estimar la pose relativa y calcular el error cometido.

Para poder realizar la estimación de la pose relativa es necesario que en el la función anterior al menos se hayan encontrado cuatro puntos coincidentes. De lo contrario el método no consigue estimar una matriz esencial, la cual es necesaria para obtener la pose relativa.

6.3.3.1 odomVisual

La funcion *[pose,error_phi,error_theta,distancia] = odomVisual (m1,m2, num1, num2)* va a calcular la pose de la imagen para posteriormente calcular el error cometido entre la estimación de la posición y la posición real.

Esta función tiene como entrada lo puntos coincidentes encontrados, así como el índice de la imagen que se ha comparado. La salida seria el error de ϕ y θ y la distancia entre las dos imágenes comparadas.

Para poder realizar estos cálculos es necesario que se hayan encontrado al menos cuatro puntos en el paso anterior. De lo contrario el programa no entrara en la función y por lo contrario nos mostrara un mensaje de alerta indicándonos el motivo por el que no se ha podido realizar el cálculo, ya sea porque no se ha encontrado ningún punto o estos no sean suficientes.

6.3.3.2 estimarRotTrans

Esta función la utilizamos dentro de la *odomVisual*. Se encarga de calcular la matriz esencial y posteriormente la descompone obteniendo así cuatro posibles soluciones de la pose relativa.

6.3.3.3 estimarMatrizE

Llamamos a esta función desde *estimarRotTrans*. Es aquí donde calculamos la matriz esencial, quedando la rotación en el eje Y y la translación en X y Z. Para general tal matriz utilizamos los puntos en el sistema de coordenadas de la cámara, como esfera unidad, calculados con la función *cam2world*.

6.3.3 Tratamiento de datos

Una vez finalizadas todas las funciones anteriores, tendríamos como resultados el error en ϕ y θ, así como la cantidad de números encontrados y el tiempo invertido en la comparación de cada imagen. Estos resultados vamos a mostrarlos sin tratamiento alguno, en el caso de comparar solo dos imágenes. En este caso también mostraríamos

ambas imágenes marcando los puntos encontrados y sus correspondencias. Sin embargo, si hemos comparado todas las imágenes entre ellas tendremos una matriz para cada uno de estos datos, por lo que calcularemos la media de estos y será el dato que mostremos. También vamos a realizar unas gráficas, para poder visualizar de forma rápida, clara y ordenada los resultados.

6.3.3.1 grafica

En esta función se tratan las matrices número de puntos y tiempo. Generando así una matriz 2xN en la que ponemos en común el tiempo con el número de puntos encontrados y se desprecian los datos para los cuales no se pueden encontrar puntos.

Esta función las usamos para sacar las dos primeras gráficas, un histograma del número de puntos y otro con los tiempos.

6.3.3.2 graficaError

La función *graficaError*, es similar a la función gráfica, comentada en el apartado anterior. En esta los valores de entrada serian tanto el error de phi como el error theta. Tratando estos datos llegamos a una matriz 2xN en la que cada columna es uno de los errores. Posteriormente utilizamos esta matriz para crear la tercera de las gráficas que podemos ver en la Guide, una gráfica con los histogramas de ambos errores.

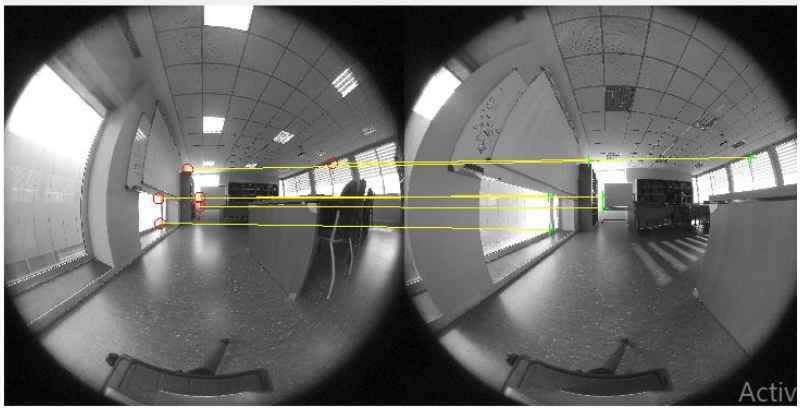
7. RESULTADOS

En este apartado se ha realizado un análisis de los resultados obtenidos e interpretados tras la simulación de diferentes casos, variando los parámetros que son posibles modificar.

7.1 MAXRATIO (R)

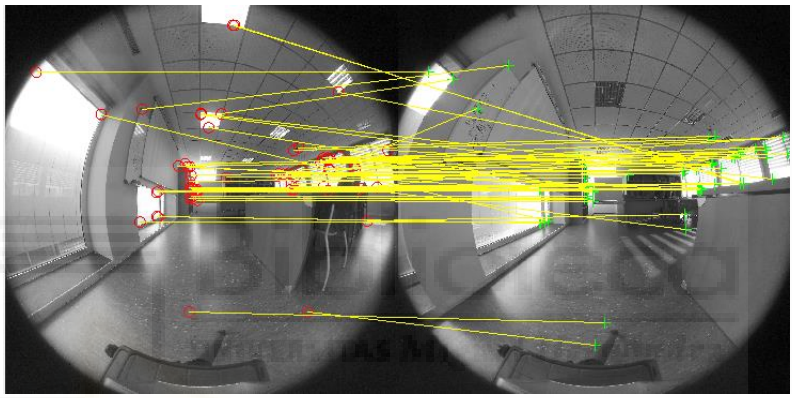
En primer lugar, vamos a realizar un estudio del *MaxRatio*, parámetro que hemos llamado *R*, el cual se introduce en la función *matchFearutes.* Este parámetro indica el porcentaje de relación entre el punto encontrado como correspondiente y el segundo vecino más cercano, rechazando así las coincidencias que puedan ser más ambiguas. Los valores que le daremos están entre 0 y 1, al aumentar este valor se obtendrán más coincidencias, pero aumentara la posibilidad de que estas coincidencias no sean exactas. Sin embargo, si lo bajamos cerca de 0 el punto elegido como correspondiente será muy distinto al siguiente candidato, con lo cual este es más fiable.

Comparando dos imágenes muy visualmente se puede comprobar como aumenta la cantidad de puntos encontrados, pero también se observa a simple vista como muchos de los puntos señalados como coincidentes no lo son. En las figuras 29 y 30 se puede ver el aumento de puntos ligado al aumento de *R*, pero también como muchos de estos puntos característicos no son coincidentes en las dos imágenes, esto conlleva un aumento del error en el cálculo.



Error phi 3.44064 Error theta 1.9298 Tiempo 2.325 s Nº Puntos 6

Figura 29: Puntos característicos encontrados SURF con R=0,2 y resultados.



Error phi 79.7473 Error theta 1.05573 Tiempo 2.1629 s Nº Puntos 65

Figura 30: Puntos característicos encontrados SURF con R=0,8 y resultados.

En las Figuras 31 y 32 vemos al comparar todas las imágenes con todas, como con el mismo tipo de puntos y mismas imágenes a comparar, el número de puntos coincidentes aumenta notablemente al aumentar la R. Pasando la media de puntos encontrados de 4,6 a 104,6 unos 100 puntos más. La problemática es que al subir esta R los puntos característicos en muchas ocasiones no son reales, comete un error mucho mayor. Esto se puede apreciar mucho más en el error φ, el cual en el ejemplo usado aumenta de un 2,65 hasta el 93,45 al incrementar la R de 0,2 a 0,8.

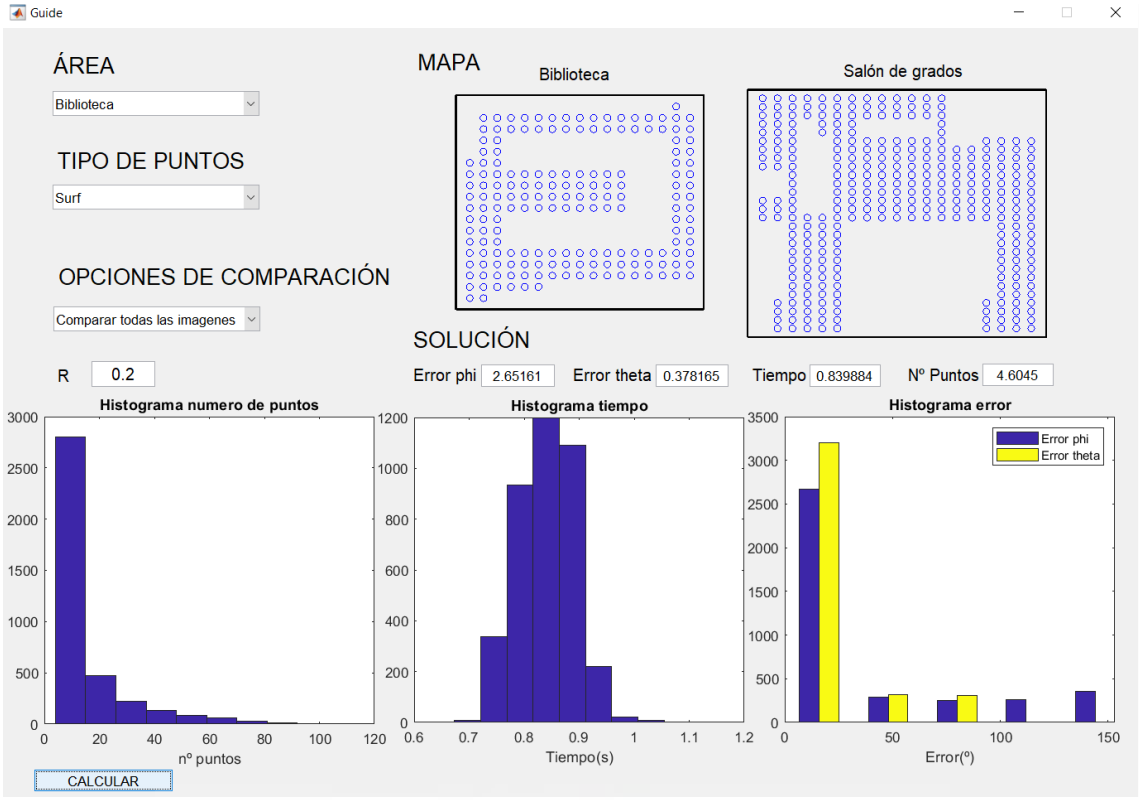


Figura 31: Guide comparando puntos SURF con R=0,2.

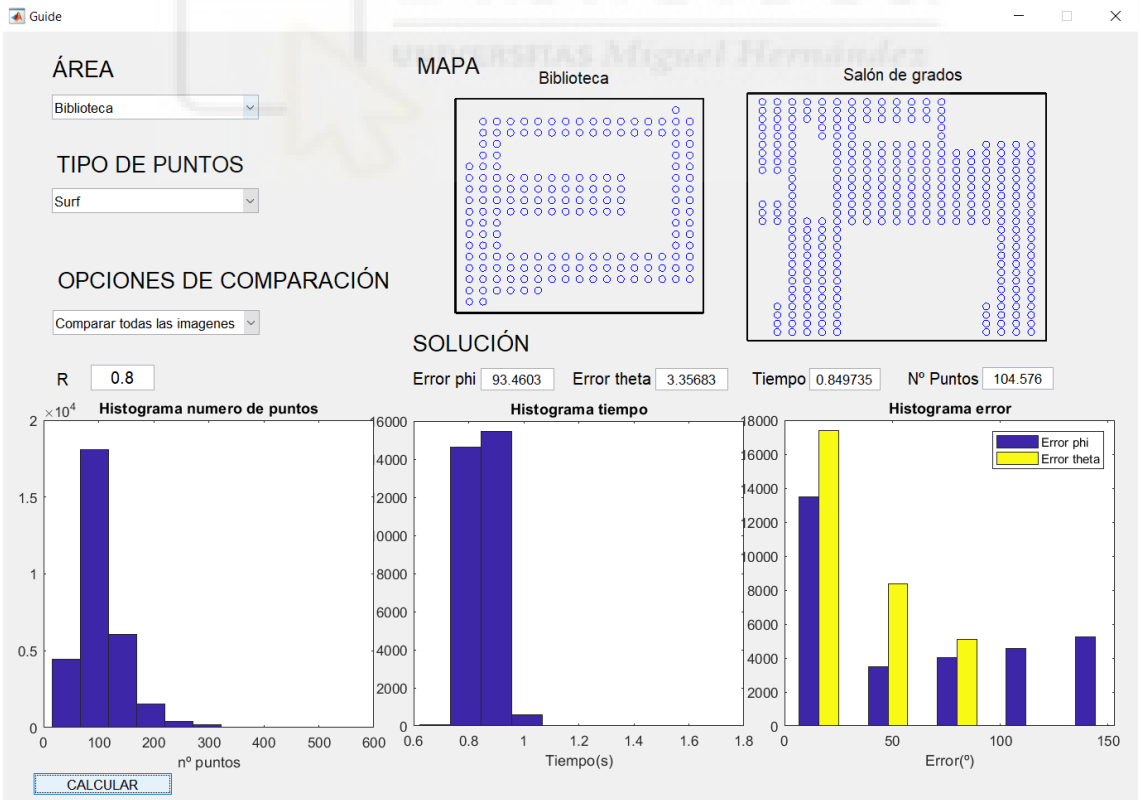


Figura 32: Guide comparando puntos SURF con R=0,8

7.2 TIPOS DE DETECTORES, TIEMPO Y NÚMERO DE PUNTOS

En el siguiente apartado vamos a realizar un estudio de los tipos de detectores usados para observar el número de puntos característicos medio detectados y el tiempo computacional medio que se emplea. Para la toma de tiempos se ha tenido en cuenta únicamente la parte de detectar puntos característicos, el algoritmo completo.

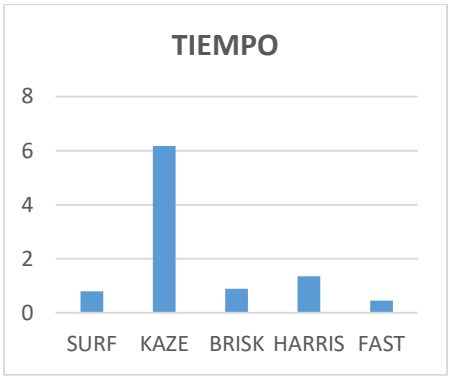


Figura 33: Tiempo computacional en segundos por cada tipo de descriptor implementado.

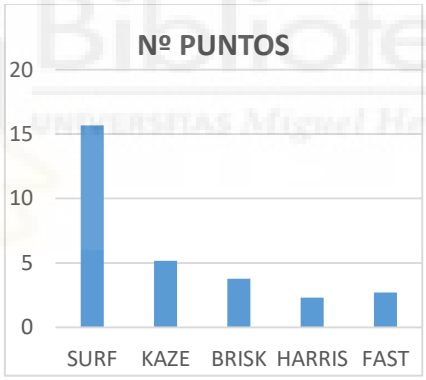
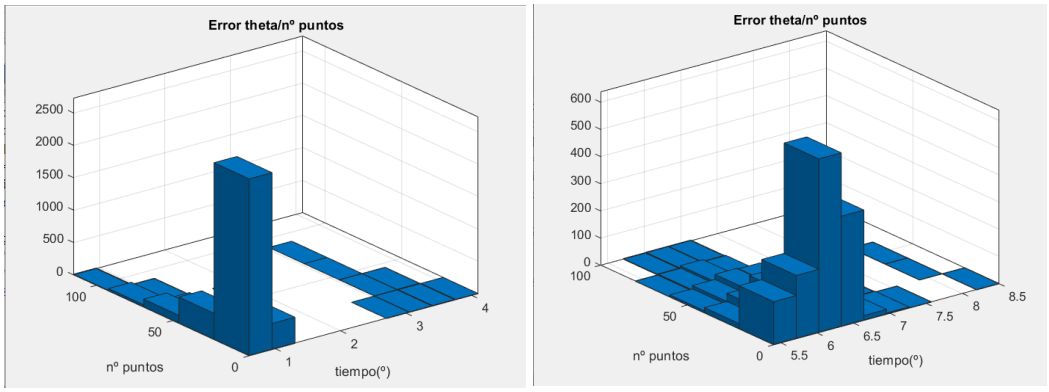


Figura 34: Número de puntos característicos detectados de cada tipo.

Tras observar los resultados obtenidos, apreciamos que el detector de puntos característicos que requiere un mayor tiempo computacional es el detector KAZE. La diferencia de tiempo con el resto de métodos de detección utilizados es significativo, ya que triplicaría los tiempos computacionales del resto de tipo de detectores. Sin embargo como apreciamos en la Figura 31, el detector de características KAZE es el segundo que más puntos detecta, solo superado por el detector SURF. Por otro lado apreciamos que el detector que menos puntos característicos ha encontrado sería HARRIS, sin embargo este sería el segundo tipo que más tiempo computacional requiere.

A continuación, vamos a comparar los tiempos de ejecución directamente con el número de puntos característicos encontrados, para ver la relación que hay entre ellos.



(a) (b)

Figura 35: Histograma tiempo computacional por cantidad de puntos característicos. (a) Puntos SURF y R=0,2. (b) Puntos KAZE y R=0,2

En el histograma podemos ver cómo el tiempo computacional no está directamente relacionado con el número de puntos encontrados. Se esperaba observar cómo el tiempo computacional aumentaba al aumentar el número de puntos detectados, pero esto no ha sucedido. Los tiempos se mantienen pese a que el número de puntos aumente.

7.3 TIPOS DE DETECTORES – ERROR

A continuación vamos a estudiar los errores dependiendo de cada tipo de detector de puntos utilizado.

Figura 36: Error medio cometido por cada tipo de puntos. (a) Error de phi. (b) Error theta

Observando las gráficas obtenidas de la media de errores de todas las imágenes comparadas en una ejecución, vamos a resaltar el tipo de detectores KAZE, ya que comete un error en ϕ bastante bajo, pero sin embargo en θ es con diferencia el tipo que mayor error comete. En general podemos ver que el descriptor de puntos SURF es el que tiene un mayor error en el cálculo de la posición, pese a ser el que un mayor de puntos característicos encuentra, como hemos comentado anteriormente. Siendo HARRIS el tipo de puntos con menor error en el ambos casos.

Estos errores tan altos podemos intuir que se deben a la problemática que encontramos en las cámaras omnidireccionales a la hora de extraer el ángulo concreto. Posteriormente han surgido diversas ambigüedades con el ángulo obtenido, ya que en algunas ocasiones como resultado tenemos un ángulo complementario o suplementario al que estábamos buscando. Esta problemática se ha intentado solucionar, pero cabe indicar que en algunos casos no ha sido del todo resuelto, aumentando así la media de los errores obtenidos.

7.4 DISTANCIA – NÚMERO DE PUNTOS – ERROR

Todas las gráficas que se muestran en el siguiente apartado han sido realizadas con los datos tomados de una ejecución con el tipo de puntos característicos SURF y una R=0,2. Los resultados obtenidos, que se comentan a continuación, no varían significativamente al utilizar otro tipo de puntos o modificar el valor de R. Por lo que tomaremos estos datos como ejemplo visual del estudio.

Comenzamos estudiando el número de puntos característicos detectados frente a la distancia a la que se han tomado las fotografías en la realidad. Para ello hemos generado un histograma, Figura 37, en el que apreciamos como al incrementarse la distancia entre las imágenes disminuye los puntos característicos encontrados. Esto es un comportamiento esperado, ya que a mayor distancia más varía el fondo de las imágenes y más complicado es encontrar puntos comunes.

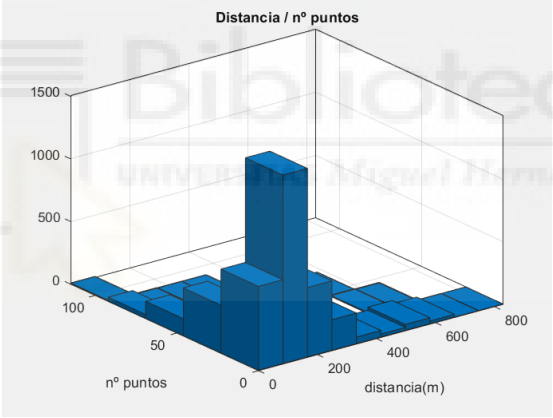


Figura 37: Histograma distancia frente a número de puntos característicos.

A continuación comparamos el número de puntos característicos con el error cometido, para estudiar su relación.

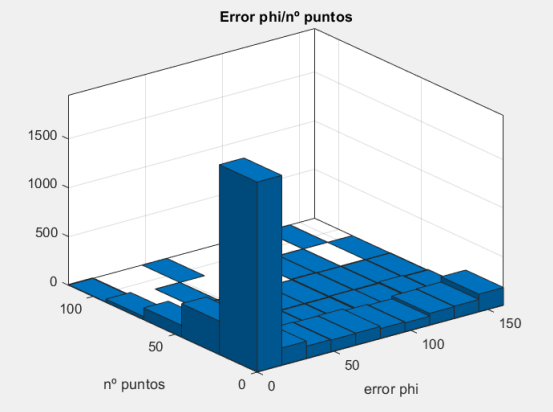


Figura 38: Histograma número de puntos característicos frente a error en φ

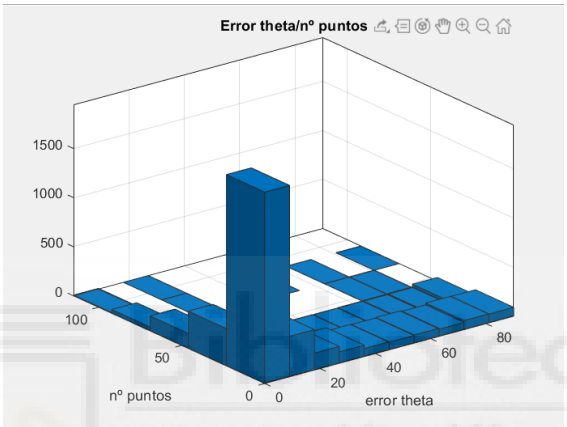


Figura 39: Histograma número de puntos característicos frente a error en ϑ.

Como podemos apreciar en los histogramas de las Figuras 38 y 39, no hay una clara relación entre el número de puntos característicos encontrados y el error cometido, tanto en ϕ como en θ. Esto se debe a que realmente el error está en el método y principalmente en los errores en las correspondencias.

Vamos a comprobar finalmente si esa relación que no existe entre el número de puntos característicos detectados y el error existiera con la distancia a la que se han tomado las fotografías.

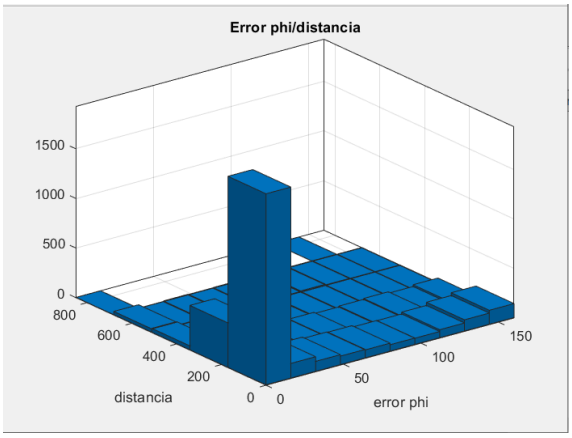


Figura 40: Histograma distancia frente a error en φ.

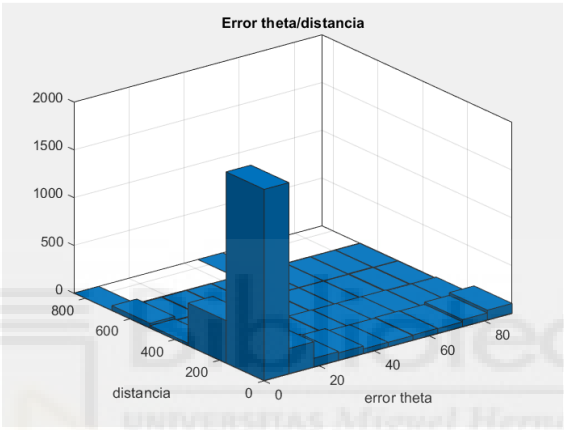


Figura 41: Histograma distancia frente a error en ϑ.

En estos histogramas tampoco podemos ver una relación directa, pese a los resultados esperados, no se cumple que a una mayor distancia haya un mayor error. Lo que nos hace cerciorarnos de que el mayor error está en las correspondencias erróneas.

8. CONCLUSIONES Y LÍNEAS FUTURAS

Este trabajo ha consistido en la realización de una interfaz gráfica en MATLAB para estimar la pose relativa entre dos imágenes de una base de datos de imágenes, todas ellas tomadas con una campara omnidireccional de 360 grados, más concretamente la cámara Garmin VIRB 360.

El primer objetivo del trabajo era obtener una base de datos de imágenes tomadas a unas distancias concretas unas de otras, para realizar todo el posterior estudio. Estas imágenes fueron tomadas en dos salas de edificio INNOVA a unos 40 cm en línea recta unas de otras. Este objetivo se logró satisfactoriamente, consiguiendo indexar las imágenes con un formato que expresa literalmente su posición por nombre.

Posteriormente se procede a desarrollar un algoritmo de localización visual, para obtener la pose relativa de las imágenes, con diferentes tipos de puntos. Para la etapa del algoritmo que correspondería con la detección de puntos característicos de las imágenes, se han utilizado puntos tipo SURF, KAZE, BRISK, HARRIS, FAST y MinEigen. En los resultados obtenidos se ha podido comprobar que el tipo de puntos KAZE requiere de un elevado tiempo computacional, notablemente mayor que cualquiera de los otros, por tanto consideramos que este tipo de detector de puntos no sería apropiado para llevar un trabajo a tiempo real.

Por otro lado observando las gráficas comprobamos el elevado valor de los errores medios cometidos al estimar la pose en casi todos los tipos de detectores. Tras analizar las gráficas consideramos que el tipo de puntos BRISK es el que menos error comete y por lo tanto seria en más apropiado a la hora de realizar un trabajo de precisión. El error en θ es en todos los casos mucho menor que el error cometido en ϕ, siendo ambos valores demasiado elevados. Esto se debe a que la cámara utilizada para la toma de imágenes no es lineal y por lo tanto ocasiona algunos inconvenientes en la calibración.

Finalmente se ha desarrollado una interfaz visual en MATLAB, que nos permite cambiar algunos valores e imágenes para su posterior cálculo de la pose relativa. Permitiendo así de una forma más sencilla realizar los cálculos de la pose relativa y

generar una salida de datos para visualizar los valores medios de tiempo, error y número de puntos, así como las gráficas de estos para realizar un estudio. Siendo esto muy beneficioso, por ejemplo, a la hora de realizar prácticas en los últimos cursos de ingeniería.

En cuanto a líneas futuras para la mejora de este trabajo podemos encontrar diversas. Desde la mejora del material para la toma de las fotografías con las que generar la base de datos, hasta el diseño de la interfaz visual Guide generada en MATLAB, pasando por la mejora para la precisión del algoritmo que calcula la pose relativa. Estas mejoras no se han podido incluir en el trabajo por la falta de tiempo y en algunos casos de medios computacionales.

En concreto, una de las mejoras que realizaría es el uso de las dos fotografías toma la cámara en un mismo instante. Pudiendo así comparar las dos fotografías de la cámara delantera y las dos de la trasera y calcular una pose común; o crear una imagen panorámica con las fotografías tomadas por ambas cámaras.

También sería importante dedicar tiempo a hacer un estudio exhaustivo de los ángulos extraídos, ya que durante este trabajo han surgido ambigüedades con los ángulos complementarios y suplementarios.

Dado que el trabajo consistía en el estudio de la localización y la orientación de distintas poses de una cámara para capturar imágenes, con el fin de completar el trabajo sería interesante instalar la cámara en un robot móvil para poder así estimar su trayectoria y llegar a generar mapas visuales del entorno.

9. BILIOGRAFÍA

[1] H. Cruz, *Monografia robot movil,* 2008.

[2] O. R. M. B. M. J. y. L. P. Arturo Gil, *Estimation of Visual Maps with a Robot Network Equipped with Vision Sensors,* 2010.

[3] L. F. R. A. G. A. L. P. C. O. R. G. David Valiente García, «Visual Odometry through Appearance- and Feature-Based Method with Omnidirectional Images,» *Journal of Robotics,* vol. 2012, nº 797063, p. 13, 2012.

[4] L. Jaulin, Mobile Robotics, WILEY, 2019.

[5] D. Goodwin, «The Evolution of Autonomous Mobile Robots,» 2020.

[6] J. A. S. Sánchez, Avances en Robótica y visión por computador, Cuenca, 2002.

[7] N. Nilsson, «Shakey the Robot,» 1984.

[8] H. P. Moravec, «The Stanford cart and the CMU rover,» *Proceedings of the IEEE,* vol. 71, nº 7, pp. 872-884, 1983.

[9] A. B.-H. y. R. H. Khalid Yousif, «An Overview to Visual Odometry and Visual SLAM: Applications to Mobile Robotics,» *Intelligent Industrial Systems,* vol. 1, pp. 289-311, 2015.

[10] C. F. y. Y. A. J. Neumann, «Eyes from eyes: new cameras for structure from motion,» Copenhagen, 2002.

[11] M. H. M. M. I. S. &. N. B. I. Mohammad O. A. Aqel, «Review of visual odometry: types, approaches, challenges, and applications,» *SpringerPlus 5,* nº 1897, 2016.

[12] A. M. y. R. S. D. Scaramuzza, «A Flexible Technique for Accurate Omnidirectional Camera Calibration and Structure from Motion,» de *Fourth IEEE International Conference on Computer Vision Systems*, Nueva York, 2006.

[13] J. B. P. S. y. J. G. Luis Puig, «Calibration of omnidirectional cameras in practice: A comparison of methods,» de *Computer Vision and Image Understanding*, 2012, pp. 120-137.

[14] C. R. V. A. J. S. Salmerón, «Procedimiento completo para el calibrado de camaras usando una plantilla plana,» *Revista Iberoamericana de Automática e Informática Industrial RIAI,* vol. 5, nº 1, pp. 93-101, 2008.

[15] D. Scaramuzza, «https://www.researchgate.net/publication/280671103_Omnidirectional_Camera,» Enero 2014. [En línea].

[16] C. M. y. P. Rives, «Single View Point Omnidirectional Camera Calibration from,» 2004.

[17] A. M. ,. R. S. D. Scaramuzza, «A Flexible Technique for Accurate Omnidirectional Camera Calibration and Structure from Motion,» 2006.

[18] M. H. M. M. I. S. y. N. B. I. Mohammad O.A Aqel, «Review of visual odometry: types, approaches, challenges, and applications,» *SpringerPlus,* vol. 5, nº 1897, 2016.

[19] H. Moravec, «Obstacle avoidance and navigation in the real world by a seeing robot rover.,» Stanford Univ., Stanfor, 1980.

[20] «WIKIPEDIA - Interfacez grafica de usuario,» [En línea]. Available: https://es.wikipedia.org/wiki/Interfaz_gr%C3%A1fica_de_usuario#:~:text=La%20interfaz%20gr%C3%A1fica%20de%20usuario,acciones%20disponibles%20en%20la%20interfaz..

[21] D. G. IONOS. [En línea]. Available: https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/que-es-una-gui/.

[22] «Simulink Documentation,» [En línea]. Available: https://es.mathworks.com/help/simulink/.

[23] «Especificaciones de la cámara Garmin Virb 360.,» [En línea]. Available: www8.garmin.com/automotive/pdfs/VIRB360-specs.pdf.

10. ANEXOS

10.1 ANEXO II: CÓDIGO IMPLEMENTADO EN MATLAB PARA GUIDE DE CÁLCULO DE POSICIÓN Y ERROR DE IMÁGENES

```
function varargout = Guide(varargin)
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @Guide_OpeningFcn, ...
                   'gui_OutputFcn',  @Guide_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
   gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
   [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
   gui_mainfcn(gui_State, varargin{:});
end



% --- Executes just before Guide is made visible.
function Guide_OpeningFcn(hObject, eventdata, handles, varargin)

global intrinsics GT GT1 GT2 num1 num2
%% %% Añadir directorios
addpath('Funciones')
addpath('Mapas')
addpath('Graficas')
addpath(fullfile(pwd,'Calibracion')) % Carpeta en la que se encuentra los parámetros de calibración
addpath(fullfile(pwd,'Ground truth')) % Carpeta en la que se encuentra las poses reales de la camara

%% Cargar imágenes, ground truth y parámetros de calibración
% Imágenes
handles.biblio = imageDatastore(fullfile(pwd,'img1'));
handles.salon = imageDatastore(fullfile(pwd,'img2'));

%Ground truth
load('GT1.mat');
load('GT2.mat');

handles.img = handles.biblio;
handles.sala = 1

%Numero de imagen1
handles.num1 = 1;
```

```
handles.num2 = 2;

%%% 1) EXTRAER PUNTOS CARACTERÍSTICOS Y SUS RESPECTIVOS DESCRIPTORES
%Tipo de puntos
handles.surf = 'SURF';
handles.kaze = 'KAZE';
handles.brisk = 'BRISK';
handles.harris = 'HARRIS';
handles.fast = 'FAST';
handles.mineigen= 'MINEIGEN';

handles.type = 'SURF';
handles.opcion = 1;
handles.R=0.1

% Choose default command line output for Guide
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);



% --- Outputs from this function are returned to the command line.
function varargout = Guide_OutputFcn(hObject, eventdata, handles)
varargout{1} = handles.output;
set(handles.text11,'visible','on')
set(handles.text12,'visible','off')
set(handles.axes4,'visible','off')
set(handles.edit6,'string',1)
set(handles.edit7,'string',2)
set(handles.edit8,'visible','off')
set(handles.edit9,'visible','off')
set(handles.edit11,'string',0.1)
set(handles.axes8,'visible','off')
set(handles.axes6,'visible','off')
set(handles.axes7,'visible','off')



% --- Executes on selection change in popupmenu1.
function popupmenu1_Callback(hObject, eventdata, handles)
str = cellstr(get(hObject,'String')); %Coge todos los datos
val = get(hObject,'Value') %coge el valor
switch str{val}
   case 'Biblioteca'
      handles.img = handles.biblio;
      handles.sala=1;
      set(handles.edit6,'visible','on')
      set(handles.edit7,'visible','on')
```

```
        set(handles.edit8,'visible','off')
        set(handles.edit9,'visible','off')
        set(handles.text11,'visible','on')
        set(handles.text12,'visible','off')
    case 'Salón de grados'
        handles.img = handles.salon;
        handles.sala=2;
        set(handles.edit8,'visible','on')
        set(handles.edit9,'visible','on')
        set(handles.edit8,'string',1)
        set(handles.edit9,'string',2)
        set(handles.edit6,'visible','off')
        set(handles.edit7,'visible','off')
        set(handles.text11,'visible','off')
        set(handles.text12,'visible','on')
end
guidata(hObject,handles)



% --- Executes during object creation, after setting all properties.
function popupmenu1_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



% --- Executes on selection change in popupmenu2.
function popupmenu2_Callback(hObject, eventdata, handles)
str = cellstr(get(hObject,'String')); %Coge todos los datos
val = get(hObject,'Value') %coge el valor
switch str{val}
    case 'Surf'
        handles.type = handles.surf;
    case 'Kaze'
        handles.type = handles.kaze;
    case 'Brisk'
        handles.type = handles.brisk;
    case 'Harris'
        handles.type = handles.harris;
    case 'Fast'
        handles.type = handles.fast;
    case 'MinEigen'
        handles.type = handles.mineigen;
end
guidata(hObject,handles)
```

```
% --- Executes during object creation, after setting all properties.
function popupmenu2_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
   set(hObject,'BackgroundColor','white');
end



% --- Executes on selection change in popupmenu3.
function popupmenu3_Callback(hObject, eventdata, handles)
str = cellstr(get(hObject,'String')); %Coge todos los datos
val = get(hObject,'Value') %coge el valor
switch str{val}
   case 'Comparar 2 imagenes'
      handles.opcion = 1;
      cla(handles.axes8,'reset')
      cla(handles.axes6,'reset')
      cla(handles.axes7,'reset')
      set(handles.uipanel3,'visible','on')
      set(handles.axes8,'visible','off')
      set(handles.axes6,'visible','off')
      set(handles.axes7,'visible','off')
      set(handles.axes4,'visible','on')

   case 'Comparar todas las imagenes'
      handles.opcion = 2;
      set(handles.uipanel3,'visible','off')
      axes(handles.axes1)
      MapaBiblioteca(' ',' ')
      axes(handles.axes2)
      MapaSalon(' ',' ')
      set(handles.axes4,'visible','off')
      cla(handles.axes4)
      set(handles.axes8,'visible','on')
      set(handles.axes6,'visible','on')
      set(handles.axes7,'visible','on')
end
guidata(hObject,handles)



% --- Executes during object creation, after setting all properties.
function popupmenu3_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
   set(hObject,'BackgroundColor','white');
end
```

```
function edit2_Callback(hObject, eventdata, handles)

% --- Executes during object creation, after setting all properties.
function edit2_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
   set(hObject,'BackgroundColor','white');
end



function edit3_Callback(hObject, eventdata, handles)

% --- Executes during object creation, after setting all properties.
function edit3_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
   set(hObject,'BackgroundColor','white');
end



function edit4_Callback(hObject, eventdata, handles)

% --- Executes during object creation, after setting all properties.
function edit4_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
   set(hObject,'BackgroundColor','white');
end



function edit11_Callback(hObject, eventdata, handles)
handles.R = str2double(get(hObject,'String'));

guidata(hObject,handles)



% --- Executes during object creation, after setting all properties.
function edit11_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
   set(hObject,'BackgroundColor','white');
end



function edit12_Callback(hObject, eventdata, handles)

% --- Executes during object creation, after setting all properties.
function edit12_CreateFcn(hObject, eventdata, handles)
```

```
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
   set(hObject,'BackgroundColor','white');
end



% --- Executes on button press in pushbutton2.
function pushbutton2_Callback(hObject, eventdata, handles)

global intrinsics GT GT2 GT1 num1 num2 %m_t m_npuntos

%% Cargar imágenes, ground truth, parámetros de calibración y tipo de puntos
% Carpetas Imágenes
handles.biblio = imageDatastore(fullfile(pwd,'img1'));
handles.salon = imageDatastore(fullfile(pwd,'img2'));

%Ground thuth
if(handles.sala == 1)
  GT = GT1;
  iteraciones=176; %176
else
  GT = GT2;
  iteraciones=316; %316
end

% Parámetros de calibración
load('cali.mat')

%Porcentaje de separacion
R= handles.R;
comparar= handles.opcion;


%%% COMPARAR 2 IMAGENES%%%
if comparar == 1
h = waitbar(0,sprintf('Buscando puntos',1));
%% Algoritmo odometría visual
%  Leer las imágenes y convertirlas a escala de grises
%  PRIMERA IMAGEN
I= rgb2gray(readimage(handles.img,handles.num1));
waitbar(0.1,h);
%  SEGUNDA IMAGEN
J= rgb2gray(readimage(handles.img,handles.num2));
waitbar(0.2,h);

%%% 1) EXTRAER PUNTOS CARACTERÍSTICOS Y SUS RESPECTIVOS DESCRIPTORES
%  PRIMERA IMAGEN
tic
[prevFeatures,prevalidPoints]=detectarFeatures(I,handles.type);
waitbar(0.6,h);
```

```
%  SEGUNDA IMAGEN
[actFeatures,actvalidPoints]=detectarFeatures(J,handles.type);
waitbar(1,h);

%%% 2) BUSCAR CORRESPONDENCIAS ENTRE LAS DOS IMÁGENES
indexPairs = matchFeatures(prevFeatures,actFeatures,'MaxRatio',handles.R,'Unique',true);

if (indexPairs ~= 0)
   m1 = prevalidPoints(indexPairs(:,1),:);
   m2 = actvalidPoints(indexPairs(:,2),:);
   close(h);

% Mostrar imagenes y correspondencia de puntos
   axes(handles.axes4)
   showMatchedFeatures(I,J,m1,m2,'Montage');


%%% 3) ESTIMAR LA POSE RELATIVA Y CALCULAR EL ERROR COMETIDO
   if (size(m1,1) >= 4)
      num1=handles.num1;
      num2=handles.num2;
      [pose,error_phi,error_theta,distancia]=odomVisual(m1,m2,num1,num2);
      set(handles.edit2,'string',error_phi);
      set(handles.edit3,'string',error_theta);
      set(handles.edit12,'string',size(m1,1));
      t=toc;
      tmp= num2str(t);
      segundos = ' s';
      tiempo = strcat(tmp,segundos);
      set(handles.edit4,'string',tiempo);
   else
      warndlg('No se han encontrado suficientes puntos para calcular la posición','MENSAJE');
      disp('No se han encontrado puntos');
      set(handles.edit2,'string','');
      set(handles.edit3,'string','');
      set(handles.edit12,'string',size(m1,1));
      set(handles.axes4,'visible','off');
   end
else
   close(h);
   warndlg('No se han encontrado puntos','MENSAJE');
   disp('No se han encontrado puntos');
   set(handles.edit2,'string','');
   set(handles.edit3,'string','');
   set(handles.edit12,'string','');
   set(handles.edit4,'string','');
   cla(handles.axes4);
   set(handles.axes4,'visible','off');
end
```

```
%%% COMPARAR TODAS LAS IMAGENES %%%
else
   cont=0;
   cont_puntos=0;
   media_error_phi = zeros(iteraciones);
   media_error_theta = zeros(iteraciones);
   media_tiempo= zeros(iteraciones);
   m_t = zeros(iteraciones);
   h = waitbar(0,sprintf('Buscando puntos',1));
   for i=1:iteraciones
      %Primera imagen
      I= rgb2gray(readimage(handles.img,i));
      [prevFeatures,prevalidPoints]=detectarFeatures(I,handles.type);
      for j=1:iteraciones
         if i ~= j
         tic
         %Segunda imagen
         J= rgb2gray(readimage(handles.img,j));
         [actFeatures,actvalidPoints]=detectarFeatures(J,handles.type);

         %Compara las dos imagenes
         indexPairs =
matchFeatures(prevFeatures,actFeatures,'MaxRatio',handles.R,'Unique',true,'MatchThreshold',10);
         if (indexPairs ~= 0)
            m1 = prevalidPoints(indexPairs(:,1),:);
            m2 = actvalidPoints(indexPairs(:,2),:);
            m_npuntos(i,j)=size(m1,1)
            cont_puntos =cont_puntos+1 %numero de veces que se detectan puntos

        % 3) ESTIMAR LA POSE RELATIVA Y CALCULAR EL ERROR COMETIDO
            if (size(m1,1) >= 4)
               [~,error_phi,error_theta,distancia]=odomVisual(m1,m2,i,j);
               %Guarda error
               m_error_phi(i,j) = error_phi;
               m_error_theta(i,j) = error_theta;
               m_distancia(i,j) = distancia;
               %Guarda tiempo
               t=toc;
               m_t(i,j)=t
               cont=cont+1
            end
         end
         end
      end
      waitbar((i/iteraciones),h);
   end
   waitbar(1,h);

  %%GRAFICAS
   % Numero puntos
```

```
   [aux]=grafica(m_npuntos,m_t)
   hist(handles.axes8,aux(:,1),10)
   title(handles.axes8,"Histograma numero de puntos")
   xlabel(handles.axes8,"nº puntos")

   %Tiempo
   hist(handles.axes6,aux(:,2),10)
   title (handles.axes6,"Histograma tiempo")
   xlabel(handles.axes6,"Tiempo(s)")

   %Error
   [aux]=graficaError(m_error_phi,m_error_theta)
   hist(handles.axes7,aux,5)
   title (handles.axes7,"Histograma error")
   legend(handles.axes7,{"Error phi","Error theta"})
   xlabel(handles.axes7,"Error(º)")

  %%CALCULOS
   %Error medio
   media_error_phi = sum(sum(m_error_phi))/cont
   media_error_theta = sum(sum(m_error_theta))/cont
   set(handles.edit2,'string',error_phi);
   set(handles.edit3,'string',error_theta);
   %Tiempo medio
   media_tiempo = sum(sum(m_t))/cont
   set(handles.edit4,'string',media_tiempo);
   media_npuntos = sum(sum(m_npuntos))/cont_puntos
   set(handles.edit12,'string',media_npuntos);

   close(h);
end



% --- Executes during object creation, after setting all properties.
function axes4_CreateFcn(hObject, eventdata, handles)



function edit6_Callback(hObject, eventdata, handles)
num1 = str2double(get(hObject,'String'));
if num1 >176
    warndlg('La imagenes introducida no existen en la base de datos.Se sustituira por la imagen numero
1','ALERTA');
    set(handles.edit6,'string',1);
end
handles.num1 = num1;
axes(handles.axes1)
MapaBiblioteca(num1,handles.num2)
guidata(hObject,handles)
```

```
% --- Executes during object creation, after setting all properties.
function edit6_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
   set(hObject,'BackgroundColor','white');
end



function edit7_Callback(hObject, eventdata, handles)
num2 = str2double(get(hObject,'String'));
if num2 >176
    warndlg('La imagen introducida no existen en la base de datos. Se sustituira por la imagen numero
1','ALERTA');
    set(handles.edit7,'string',1);
end
handles.num2 = num2;
axes(handles.axes1)
MapaBiblioteca(handles.num1,num2)
guidata(hObject,handles)



% --- Executes during object creation, after setting all properties.
function edit7_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
   set(hObject,'BackgroundColor','white');
end



function edit8_Callback(hObject, eventdata, handles)
num1 = str2double(get(hObject,'String'));
if num1 >316
    warndlg('La imagenes introducida no existen en la base de datos.Se sustituira por la imagen numero
1','ALERTA');
    set(handles.edit6,'string',1);
end
handles.num1 = num1;
axes(handles.axes2)
MapaSalon(num1,handles.num2)
guidata(hObject,handles)



% --- Executes during object creation, after setting all properties.
function edit8_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
```

```
   set(hObject,'BackgroundColor','white');
end


function edit9_Callback(hObject, eventdata, handles)
num2 = str2double(get(hObject,'String'));
if num2 >316
    warndlg('La imagen introducida no existen en la base de datos. Se sustituira por la imagen numero
1','ALERTA');
    set(handles.edit7,'string',1);
end
handles.num2 = num2;
axes(handles.axes2)
MapaSalon(handles.num1,num2)
guidata(hObject,handles)



% --- Executes during object creation, after setting all properties.
function edit9_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
   set(hObject,'BackgroundColor','white');
end



% --- Executes during object creation, after setting all properties.
function axes1_CreateFcn(hObject, eventdata, handles)
global num1 num2 GT
addpath('MAPAS')
num1= 1;
num2= 2;
MapaBiblioteca(num1,num2)



function axes2_CreateFcn(hObject, eventdata, handles)
addpath('MAPAS')
num1= 1;
num2= 2;
MapaSalon(num1,num2)



function axes6_CreateFcn(hObject, eventdata, ~)



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
function [descriptores,validPoints]=detectarFeatures(I,Feature)
%%%
% En función del tipo de puntos característicos seleccionado, se llama a
% una de las funciones incluídas en Computer Vision Toolbox.
% Argumentos de entrada:
%   I - imagen de entrada en escala de grises.
%   Feature - tipo de punto característico que se pretende detectar:
% Argumentos de salida:
%   descriptores - matriz de MxN, donde M es el número de puntos y N la
%       longitud de cada descriptor.
%   validPoints - son los puntos de los cuales se ha podido obtener su
%       descriptor.
%%%

% 1)    Extraer características locales
switch Feature

   case 'SURF' % Detectar puntos SURF
   value=2000.0; % Al aumentar el valor se obtiene menos puntos pero más fuertes.
   points = detectSURFFeatures(I,'MetricThreshold',value);

   case 'KAZE' % Detectar puntos KAZE
   value=0.005; % Más restrictivo cuanto más aumenta. Value>0 .  Default: 0.0001
   points = detectKAZEFeatures(I,'Threshold',value);

   case 'BRISK' % Detectar puntos BRISK.
   value=0.2; %Mas restrictivo cuanto mas aumenta. 0<value<1. Default: 0.2
   points=detectBRISKFeatures(I,'MinContrast',value);

   case 'HARRIS' % Detectar esquinas HARRIS
   points = detectHarrisFeatures(I);

   case 'FAST' % Detectar esquinas FAST
   points = detectFASTFeatures (I);

   case 'MINEIGEN' % Detectar esquinas MinEigen
   points = detectMinEigenFeatures(I);

   otherwise
   disp('No es válido el valor introducido')
end

% 2)    Obtener los descriptores de cada punto extraído
[descriptores,validPoints] = extractFeatures(I,points);
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



function [pose,error_phi,error_theta,distancia]=odomVisual(m1,m2,num1,num2)
global GT

pose=estimarRotTrans(m1,m2);

% Ángulos de obtenidos en positivo
if pose(1)<0
    pose(1)=pose(1)+360;
end

if pose(2)<0
    pose(2)=pose(2)+360;
end

%% CALCULAR EL ERROR COMETIDO EN LA ESTIMACIÓN
%%% POSE REAL: GROUND TRUTH
% Coodenadas
   x1= GT(num1,1);
   x2= GT(num2,1);
   th1=GT(num1,3);
   th2=GT(num2,3);
   y1= GT(num1,2);
   y2= GT(num2,2);

% Angulo de ROTACIÓN
phi_r= rad2deg(atan2((y2-y1),(x2-x1)));

%en positivo
if phi_r<0
   phi_r=phi_r+360;
end

%Distancia entre posiciones (imágenes)
distancia = sqrt((x2-x1)^2+(y2-y1)^2);

%Theta relativo
theta_r=abs(th2-th1);
%%% CÁLCULO DEL ERROR COMETIDO
error_theta=abs(pose(1)-theta_r);
error_phi=abs(pose(2)-phi_r);
```

```
% Si el error es mayor a 180º, coger el ángulo conjugado
if error_phi>160
%    error_phi=abs(360-error_phi);
error_phi=atan(sin(error_phi)/cos(error_phi));
% else if error_phi>180 && error_phi<220
%        error_phi=error_phi-180;
end

if error_phi<0
   error_phi=-error_phi;
end
if error_theta>180
   error_theta=360-error_theta;
end

end
```

%%%


```
function [Solucion,th1,th2,phi1,phi2]=estimarRotTrans(m1,m2)
%%%
% Calcula la matriz esencial y posteriormente la descompone obteniendo las
% cuatro posibles soluciones de pose relativa
%%%

global intrinsics

p1=m1.Location;
p2=m2.Location;


if size(m2,1)<4
   th1=[]; phi1=[]; th2=[]; phi2=[];
   return
end

% Obtener los puntos en el sistema de coordenadas de la cámara (esfera
% unidad)
P1=cam2world(p1',intrinsics);
P2=cam2world(p2',intrinsics);

% Matriz esencial
```

```
E = estimarMatrizE(P1,P2);

% Descomponer la matriz esencial para obtener las cuatro posibles
% soluciones
[U,~,V]=svd(E);

W=[0 -1 0;1 0 0;0 0 1];
Z = [0 1 0; -1 0 0; 0 0 0];

% Posibles matrices de rotación
R1 = U * W * V';
R2 = U * W' * V';

if det(R1) < 0
   R1 = -R1;
end

if det(R2) < 0
   R2 = -R2;
end

% Posibles vectores de traslación
Tx = U * Z * U';
t1=[Tx(3, 2), Tx(1, 3), -Tx(2, 1)];
t2=-t1;

% Cuatro posibles soluciones:
% Rotación
% 1) R1
ang = rad2deg(rotm2eul(R1));
th1=ang(2);
% 2) R2
ang = rad2deg(rotm2eul(R2));
th2=ang(2);
% Traslación
% 1) t1
phi1=atan2d(t1(3),t1(1));
% 2) t2
phi2=atan2d(t2(3),t2(1));


%Teniendo en cuenta las soluciones desfasadas 180º, habrá 4 combinaciones
%posibles. Calculamos si el rayo reproyectado tiene coordenadas positivas,
%es decir, está delante de las dos cámaras. La solución elegida será
%aquella que obtenga más rayos con coordenadas positivas
X=[P1;P2];
c1=[];
c2=[];
c3=[];
c4=[];
```

```
for i=1:length(p1)
   c1 = [c1 test_solution(X(1:3,i),X(4:6,i), R1, t1)];
   c2 = [c2 test_solution(X(1:3,i),X(4:6,i), R1, t2)];
   c3 = [c4 test_solution(X(1:3,i),X(4:6,i), R2, t1)];
   c4 = [c4 test_solution(X(1:3,i),X(4:6,i), R2, t2)];
end
c1 = sum(c1);
c2 = sum(c2);
c3 = sum(c3);
c4 = sum(c4);
[c1 c2 c3 c4];
[m ind]=max([c1 c2 c3 c4]);
%%%%%%%
%default case
Solucion=[0 0];

switch ind
   case 1
      Solucion=[th1 phi1];
   case 2
      Solucion=[th1 phi2];
   case 3
      Solucion=[th2 phi1];
   case 4
      Solucion=[th2 phi2];
end
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%




function M=cam2world(m, ocam_model)

n_points = size(m,2);

ss = ocam_model.ss;
xc = ocam_model.xc;
yc = ocam_model.yc;
width = ocam_model.width;
height = ocam_model.height;
c = ocam_model.c;
d = ocam_model.d;
e = ocam_model.e;

A = [c,d;
```

```
    e,1];
T = [xc;yc]*ones(1,n_points);

m = A^-1*(m-T);
M = getpoint(ss,m);
M = normc(M); %normalizes coordinates so that they have unit length (projection onto the unit sphere)

function w=getpoint(ss,m)

% Given an image point it returns the 3D coordinates of its correspondent optical
% ray

w = [m(1,:) ; m(2,:) ; polyval(ss(end:-1:1),sqrt (m(1,:).^2+m(2,:).^2)) ];
```

%%%

```
function Es = estimarMatrizE(P1,P2)
P1=P1';
P2=P2';

% Rotación eje Y translación en X y Z

% Si no se ha obtenido un mínimo de cuatro correspondencias no se puede
% calcular la matriz esencial

Q = [P1(:,2).*P2(:,1),...
P1(:,1).*P2(:,2),...
P1(:,3).*P2(:,2),...
P1(:,2).*P2(:,3),...
];

[~, ~, V] = svd(Q);
Ep=[  0      V(2,4)      0;
   V(1,4)    0      V(4,4);
    0      V(3,4)      0  ];


% RESTRICCIONES

% a) Debe ser singular
Ep = Ep / norm(Ep);

% b) Tiene dos valores singulares iguales
[U,S,V]=svd(Ep);
```

```
c=(S(1,1)+S(2,2))/2;

Sprima=[c   0   0 ;
     0   c   0 ;
     0   0   0 ];

Es = U * Sprima * V';

end
```

%%

```
function [aux]=grafica(m_npuntos,m_t)
v_npuntos = [m_npuntos(:)']; %convertir en vector
v_tiempo = [m_t(:)'];
matriz=[v_npuntos;v_tiempo]; %une dos vectores en matriz
matriz = matriz'; % la transpongo para verla mejor en columnas.
[a b]=find(matriz(:,2))  % encontramos los valores distintos de 0.
aux=matriz(a,:)  % los ponemos en una matriz auxiliar
end
```


%%

```
function [aux]=grafica_error(m_error_phi,m_error_theta)
v_error_phi= [m_error_phi(:)']; %convertir en vector
v_error_theta = [m_error_theta(:)'];
matriz=[v_error_phi;v_error_theta]; %une dos vectores en matriz
matriz = matriz'; % la transpongo para verla mejor en columnas.
[a b]=find(matriz(:,2))  % encontramos los valores distintos de 0.
aux=matriz(a,:)  % los ponemos en una matriz auxiliar
histograma= hist(aux,5);
end
```

%%

```
function MapaSalon(num1,num2)
load('Ground truth/GT2.mat')
cla %limpia

%Salon. Toda la matriz
colum1=GT2(:,1);
colum2=GT2(:,2);

%Imagenes seleccionadas
x_img1=GT2(num1,1);
y_img1=GT2(num1,2);
x_img2=GT2(num2,1);
y_img2=GT2(num2,2);

hold on
plot(0,0)
plot(colum1,colum2,'bo','MarkerSize',5);
if (num1~= ' ' & num2~= ' ')
plot(x_img1,y_img1,'rx','MarkerSize',10,'linewidth',1.5)
plot(x_img2,y_img2,'rx','MarkerSize',10,'linewidth',1.5)
end

x_marco=[(min(colum1)-40);(min(colum1)-40);(max(colum1)+40);(max(colum1)+40); (min(colum1)-40)];
y_marco=[(min(colum2)-40);(max(colum2)+40);(max(colum2)+40);(min(colum2)-40);(min(colum2)-40)];
line(x_marco,y_marco, 'Color','k','LineWidth',1.5)

axis off
end



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



function MapaBiblioteca (num1,num2)
load('Ground truth/GT1.mat')
cla %limpia

%Biblioteca. Toda la matriz
colum1=GT1(:,1);
colum2=GT1(:,2);

%Imagenes seleccionadas
x_img1=GT1(num1,1);
```

```
y_img1=GT1(num1,2);
x_img2=GT1(num2,1);
y_img2=GT1(num2,2);

hold on
plot(0,0)
plot(colum1,colum2,'bo','MarkerSize',5);
if (num1~= ' ' & num2~= ' ')
plot(x_img1,y_img1,'rx','MarkerSize',10,'linewidth',1.5)
plot(x_img2,y_img2,'rx','MarkerSize',10,'linewidth',1.5)
end

x_marco=[(min(colum1)-40);(min(colum1)-40);(max(colum1)+40);(max(colum1)+40); (min(colum1)-40)];
y_marco=[(min(colum2)-40);(max(colum2)+40);(max(colum2)+40);(min(colum2)-40);(min(colum2)-40)];
line(x_marco,y_marco, 'Color','k','LineWidth',1.5)

axis off
end
```