TESIS DOCTORAL

Técnicas para la localización visual robusta de robots en tiempo real con y sin mapas

Autor:

EDUARDO PERDICES GARCÍA

Director:

JOSÉ MARÍA CAÑAS PLAZA

Programa de Doctorado en
Sistemas de Hardware y Software Avanzados

2017

Autorizo la defensa de la tesis doctoral *Técnicas para la localización visual robusta de robots en tiempo real con y sin mapas* cuyo autor es D. Eduardo Perdices García.

José María Cañas Plaza

Madrid, 20 de Abril de 2017

Agradecimientos

Han pasado muchos años desde que entré en el grupo de robótica de la URJC, coincidiendo en este tiempo con muchos estudiantes, doctorandos, profesores e investigadores. He podido compartir con ellos buenos momentos tanto fuera como dentro de la Universidad, tanto con mis compañeros de despacho (Julio, Gonzalo, Alejandro, Borja) como con los que se pasaban el día en el laboratorio haciendo sus proyectos (Sara, Rubén, Luismi, Fran, Roca, Darío, Pablo y otros muchos). A todos ellos les quiero agradecer su ayuda y compañía en todo este tiempo.

También han sido parte fundamental de esta tesis los miembros del equipo de la RoboCup de la URJC, ya que han colaborado directamente en parte del *software* de este trabajo. Además, hemos vivido juntos grandes momentos, como el primer partido del Mediterranean Open que nunca olvidaremos.

Por supuesto, tengo que agradecer a mi tutor, José María, todos estos años de dedicación en los que nos hemos reunido casi semanalmente, trabajando juntos en el proyecto de fin de carrera, el trabajo de fin de máster, en otros proyectos y ahora en esta tesis. Gracias a él he descubierto este apasionante mundo de la robótica y la visión artificial, que probablemente no hubiese conocido de otra forma.

Por último, quisiera dar las gracias a todos aquellos que me han estado apoyando consciente o inconscientemente a lo largo de estos años, a mis padres, mis hermanos, mis mejores amigos (Diego, Marina, Carolina) y en especial a Estefanía, quienes han sido testigos de todo el esfuerzo necesario para realizar este trabajo.

A todos, ¡muchas gracias!

Resumen

Los robots son cada vez más utilizados en todo tipo de tareas. Para que los robots sean útiles, deben ser capaces de obtener información de su alrededor para interactuar con el medio que les rodea. Esta información se puede recibir mediante múltiples sensores, destacando entre ellos las cámaras ya que permiten obtener muchos datos y su precio es bajo. Para que un robot pueda interactuar con su entorno es fundamental que conozca su localización, de la que dependerá en gran medida su comportamiento.

Durante muchos años, uno de los principales campos de investigación de la robótica ha sido la localización de robots en entornos conocidos, en los que se dispone de un mapa o se conoce la posición de ciertos elementos reconocibles (balizas). Existen numerosas alternativas para abordar este problema: la localización con sensores específicos, la localización mediante balizas y geometría o la localización probabilística (filtros de Kalman, modelos de Markov, filtros de partículas, etc). La principal ventaja de estos algoritmos es su robustez; sin embargo, presentan ciertas limitaciones a la hora de tratar con mapas que contengan simetrías, ya sea porque no sean capaces de mantener de forma simultánea varias posibles soluciones o porque su tiempo de cómputo sea demasiado elevado.

Por otro lado, la localización de robots o cámaras independientes en entornos desconocidos ha tenido un gran avance en los últimos años. A partir del año 2003, se empezaron a desarrollar los algoritmos conocidos como MonoSLAM. Estos algoritmos lograban a partir de una sola cámara localizarse en 3D y crear un mapa del entorno de forma simultánea en tiempo real, utilizando para ello filtros de Kalman. Desde el 2007, se intentó resolver este mismo problema empleando algoritmos de optimización que funcionasen en tiempo real, con los que se mejoró la precisión y los mapas obtenidos.

Los algoritmos de localización con mapas permiten situar a los robots en entornos de grandes dimensiones, pero su localización no suele ser muy precisa (centímetros). Por otra parte, los algoritmos de localización en entornos desconocidos tienen una gran precisión (milímetros), pero se pierden fácilmente en determinadas circunstancias. Por ello, en esta tesis se explora la forma de aportar conocimiento en estos dos tipos de

algoritmos de localización. Primero, se propone un nuevo método de localización con mapas conocidos enfocado a entornos que cuenten con simetrías, utilizando una sola cámaras y funcionando en tiempo real. El segundo aporte consiste en un algoritmo de autolocalización visual sin mapas computacionalmente ligero, para que funcione incluso en procesadores modestos, típicos de drones o teléfonos móviles. Ambas propuestas han sido validadas experimentalmente en condiciones reales.

Abstract

Robots are increasingly becoming more important in many fields. To make robots useful, they must obtain information from their environment to interact with their surroundings. This information can be received from numerous sensors, particularly cameras as many data can be obtained from them and they are relatively inexpensive. Robot self-localization is an essential factor so that a robot may properly interact with its surroundings, since the right behavior may depend heavily on its localization.

For many years, robot self-localization in known environments has been one of the main research fields of robotics, using maps or only certain recognizable elements (beacons). There are many techniques to address this problem: localization with specific sensors, localization using beacons and geometry or probabilistic localization (Kalman filters, Markov models, particle filters, etc.). The main advantage of these algorithms is their robustness; however, they suffer in environments with symmetries, either because they can not handle several possible solutions simultaneously or because their execution time is too high. Map localization algorithms allow robots to be located in large environments, but their localization is often not very accurate (centimeters).

On the other hand, robot localization in unknown environments has achieved a huge breakthrough in the last years. Starting in 2003, so-called MonoSLAM approaches have been developed, which use a single camera to simultaneously locate a robot in 3D and create a map of its surroundings in real-time using Kalman filters. Since 2007, new approaches emerged to solve this problem using optimization algorithms operating in real-time, improving accuracy and maps density. Localization algorithms in unknown environments are very accurate (millimeters), but they get lost easily in certain circumstances.

This thesis explores how to contribute in these two localization algorithms types. First, a new map localization approach is proposed focused on maps with symmetries, using one single camera and real-time operation. The second contribution is an efficient self-localization algorithm for unknown environments, which works even on modest processors. Both approaches have been validated experimentally in real conditions.

Índice general

Índice de figuras

Capítulo 1

Introducción

La presente tesis doctoral resume el trabajo de investigación realizado en el área de la autolocalización de robots tanto en entornos conocidos como en desconocidos utilizando una cámara como sensor principal. Este problema ha sido estudiado en profundidad en las últimas décadas a medida que se han ido mejorando tanto la capacidad de cómputo de los procesadores como la precisión y la robustez de los algoritmos desarrollados.

En este capítulo se presenta el contexto de esta tesis y se resalta la utilidad del problema de investigación elegido, que sigue siendo un problema abierto. Primero se describirán los campos de investigación en los que se encuadra, la robótica y la visión artificial, y después se presentarán los objetivos concretos de la tesis y el problema abordado.

1.1. Robótica

Un robot es un sistema electromecánico que utiliza una serie de elementos *hardware* (actuadores, sensores y procesadores) y cuyo comportamiento viene controlado por un *software* programable que le da la inteligencia. La robótica se puede ver como la ciencia y la tecnología de los robots, donde se combinan varias disciplinas como la mecánica, la informática, la electrónica y la ingeniería artificial, que hacen posible el diseño *hardware* y *software* del robot.

En 1961 se construyó el primer robot programable; este robot, conocido como *Unimate*, servía para levantar piezas industriales a altas temperaturas. Sin embargo, no sería hasta la década de los 70 cuando se comenzase a desarrollar del todo esta tecnología, creándose en 1973 el primer robot con 6 ejes electromecánicos (nombrado *Famulus*) y en 1975 el primer brazo mecánico programable (*PUMA*).

A partir de los años 80 se comenzaron a utilizar en masa este tipo de robots, ya que proporcionaban alta rapidez y precisión. Se utilizaron sobre todo para la fabricación de coches, el empaquetamiento de comida y otros bienes y la producción de placas de circuitos impresos.

Desde entonces se han realizado grandes avances en la robótica, existiendo actualmente robots para todo tipo de propósitos. Son principalmente utilizados en entornos industriales puesto que realizan el trabajo de una forma más precisa y barata que los humanos (Figura 1.1(a)). También se utilizan en otros campos como la medicina, la educación, el rescate de personas, la ayuda en las tareas del hogar e incluso la exploración espacial.

Para esta tesis, los robots más interesantes son los robots móviles, es decir, aquellos especializados en la movilidad sobre un terreno (ya sea conocido o desconocido). Para ello, los robots hacen uso de distintos sensores que les permiten captar aquella información del exterior necesaria para conocer su posición en el entorno o para navegar a un destino solicitado.

Uno de los mejores ejemplos de este tipo de robots son los coches autónomos que han desarrollado numerosas empresas, como Google, Tesla o Uber. Así, dotando a un coche de diversos sensores, como cámaras, sensores láser, GPS, etc, éste es capaz de moverse de forma totalmente autónoma entre el tráfico de la ciudad. Para ello, el *software* del coche debe ser capaz de detectar señales de tráfico (semáforos, peatones y otros objetos) y de interactuar con estos elementos tal y como lo haría un humano (Figura 1.1(b)).

(a) (b)

Figura 1.1: Brazos robóticos industriales fabricando coches (a). Software de conducción autónoma *Autopilot* de Tesla (b).

Otro ejemplo destacable son los vehículos aéreos no tripulados, también conocidos como drones o UAV (*Unmanned Aerial Vehicle*), que inicialmente se utilizaron con fines militares

y que, poco a poco, van aplicándose a otros mercados civiles como la vigilancia, la inspección o el envío de paquetería. Así, empresas como Amazon ya cuentan con prototipos que realizan envíos de paquetes y lo depositan sobre un logotipo impreso que el cliente sitúa libremente.

Existen otros muchos robots móviles, como la aspiradora *Roomba*, utilizada en el ámbito doméstico para limpiar de forma automática, o los robots humanoides, que emulan la forma de andar de los humanos e interactúan con éstos utilizando multitud de sensores.

1.2. Visión artificial

Muchos de los robots anteriores utilizan la visión artificial como sensor principal, siendo éste el segundo campo de investigación que da contexto a esta tesis. La visión artificial es un campo de la inteligencia artificial cuyo objetivo es la extracción de información sobre el mundo real empleando una o varias imágenes. La información relevante que se puede obtener a partir de las imágenes permite por ejemplo reconocer objetos, recrear en 3D la escena que se observa o realizar el seguimiento de una persona u objeto.

El inicio de la visión artificial se puede situar en 1961 por parte de Larry Roberts, quien creó un programa que podía ver una estructura de bloques, analizar su contenido y reproducirla desde otra perspectiva, utilizando para ello una cámara y procesando la imagen desde un computador. Sin embargo, para obtener este resultado, las condiciones de la prueba estaban muy controladas y simplificadas; otros muchos científicos trataron de solucionar el problema de conectar una cámara a un computador y hacer que éste describiese lo que veía en condiciones más realistas.

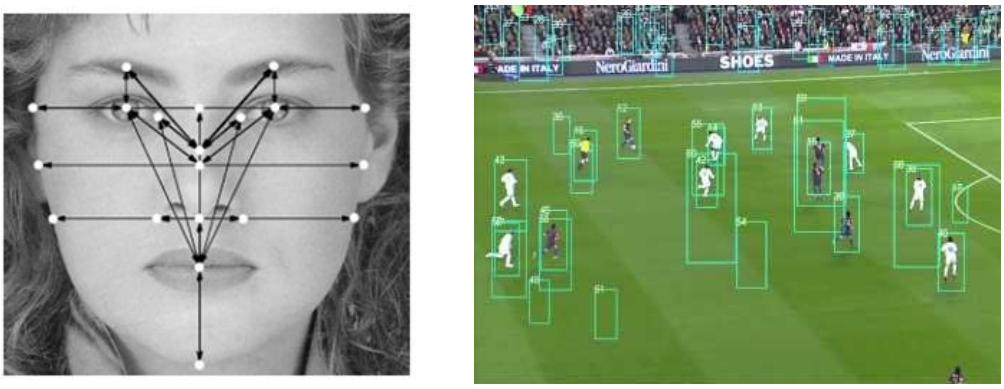
Los investigadores se dieron cuenta de que la tarea no era sencilla ya que, aunque a los seres humanos nos resulte natural percibir objetos con nuestra visión, aún no comprendemos en detalle el mecanismo biológico que nos permite hacerlo, por lo que no resulta fácil plasmarlo en un algoritmo. Esto ha dado pie a un amplio campo de investigación que tomó el nombre de visión artificial, en el que se han realizado grandes avances.

Si bien el uso de cámaras como fuente de información implica un alto coste computacional, si se consiguen analizar correctamente las imágenes es posible extraer mucha más información que con otro tipo de sensores.

A comienzos de los años 90 comenzaron a aparecer ordenadores capaces de procesar lo suficientemente rápido imágenes, por lo que comenzaron a tomar forma y a dividirse los

posibles problemas de la visión artificial en otros más específicos. Actualmente, la visión artificial se utiliza en muchos procesos científicos, militares o industriales, por ejemplo para el reconocimiento de objetos o en el seguimiento de éstos:

- Reconocimiento de objetos: Se trata de buscar unas propiedades concretas de un determinado objeto (forma, color o cualquier otro patrón) en una imagen para determinar si un objeto se encuentra o no en ella. Por ejemplo mediante la obtención de píxeles característicos que destaquen en la imagen (Figura 1.2(a)) o utilizándose técnicas de *Deep Learning* con redes neuronales.
- Seguimiento de objetos: Tras ser detectado, se pueden realizar tareas de seguimiento de un objeto. Podrá efectuarse dicho seguimiento teniendo en cuenta sus propiedades (texturas, bordes, etc) o analizando su desplazamiento respecto a imágenes anteriores (Figura 1.2(b)).



(a) (b)

Figura 1.2: Reconocimiento de cara (a). Seguimiento de jugadores en tiempo real (b).

Analizando las imágenes en busca de primitivas básicas se podrán identificar elementos más complejos. Así por ejemplo, las técnicas conocidas como OCR (*Optical Character Recognition*) permiten identificar símbolos o letras en imágenes que pueden ser después procesadas para buscar un significado, como por ejemplo para obtener la matrícula de un coche a la entrada de un aparcamiento o incluso para traducir textos a otros idiomas.

Otras técnicas como la *segmentación semántica* pueden ser utilizadas para identificar objetos en la imagen que permitan identificar la escena global en la que se encuentran.

1.2.1. Áreas de aplicación

A medida que avanza la investigación en el campo de la visión por computador el número de aplicaciones aumenta de forma exponencial. No es fácil establecer unos límites en las áreas de aplicación, puesto que hoy en día se ha ganado mucho en robustez y se utilizan cada vez más en entornos que eran inimaginables hace unos años. Por ello, todo hace pensar que la mayoría de aplicaciones están por llegar y no se conocen todavía. Aún así, a día de hoy estas son varias áreas en las que la visión artificial destaca:

- Manufacturas: Es una de los áreas más importantes y con más años de experiencia en el uso de este tipo de técnicas. La visión artificial puede utilizarse tanto para la selección y clasificación de objetos como para la identificación de elementos defectuosos en sistemas de control de calidad.

- Medicina: Mediante visión por computador puede extraerse información de las imágenes médicas, como rayos X o tomografías, para facilitar el diagnóstico de los pacientes detectando tumores u otras malformaciones.

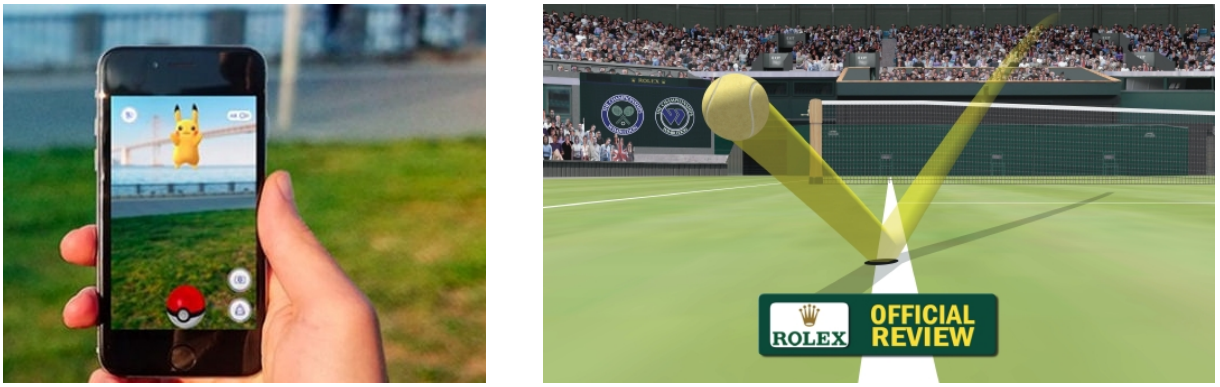
- Defensa: En el ámbito militar está ampliamente extendido el uso de visión por computador. Empleando cámaras y otros sensores se obtiene y se procesa la información para identificar objetivos e incluso se pueden guiar misiles de forma precisa en tiempo real.

- Videojuegos: Cada vez son más los videojuegos en los que se interactúa con el entorno mediante cámaras, permitiendo experiencias de realidad aumentada con teléfonos móviles u otros dispositivos de uso doméstico (Figura 1.3(a)).

- Deportes: Con el avance de la tecnología, cada vez son más los deportes en los que las cámaras juegan un papel fundamental. En deportes como el fútbol americano se utilizan algoritmos de reconstrucción por visión para ver las jugadas en 3 dimensiones y poder tomar decisiones. En otros deportes como el tenis se utiliza el llamado *Ojo de Halcón* para determinar si una pelota ha caído dentro o fuera del campo (Figura 1.3(b)).

- Vigilancia: El seguimiento de personas permite determinar de forma automática si se están dando situaciones de alerta. Por ejemplo, es posible detectar intrusos en casas o establecimientos, o saber si alguien está invadiendo las vías antes de que venga un tren.

- Otros: Existen otras muchas aplicaciones que están hoy en día en desarrollo pero que en un futuro serán una realidad, como por ejemplo la compra en centros comerciales sin necesidad de pasar por caja gracias al seguimiento por visión de los clientes (supermercados *Amazon Go*[1]).



(a) (b)

Figura 1.3: Realidad aumentada en videojuegos (a). *Ojo de Halcón* en partido de tenis (b).

1.2.2. Cámaras RGBD

Las cámaras clásicas proporcionan únicamente información de color para cada uno de sus píxeles, ya sea en escala de grises o en varios canales. Sin embargo, en los últimos años ha aparecido un nuevo tipo de sensor, las llamadas cámaras RGBD, que además de entregar imágenes como el resto de cámaras, también proporcionan la distancia a la que se encuentran los objetos en cada uno de los píxeles de la imagen.

Figura 1.4: Dispositivo *Kinect* con cámara RGBD.

El auge de este sensor comenzó en el año 2010 con el lanzamiento al público de *Kinect* (Figura 1.4), un dispositivo vendido para su utilización con la videoconsola *Xbox 360*. Aunque inicialmente fue pensado para la industria del videojuego, su bajo precio y su

[1] `https://www.amazon.com/go`

potencial uso hicieron que otros fabricantes sacasen al mercado dispositivos similares y que hayan sido desde entonces ampliamente utilizados en múltiples campos de investigación.

La primera versión de *Kinect* determinaba la distancia emitiendo un patrón conocido de luz en infrarrojos y viendo cómo aparecía ese patrón en la imagen de infrarrojos. La segunda generación de *Kinect*, lanzada en 2014, utiliza la tecnología de tiempo de vuelo (TOF) para calcular la distancia a cada objeto. En ambos casos, su funcionamiento hace que estos sensores no puedan utilizarse a distancias mayores de 7-9 metros ni en escenarios donde la luminosidad sea muy alta (o haya luz solar directa), por lo que su funcionamiento solo es óptimo en interiores.

Conociendo sus limitaciones, este tipo de dispositivos son cada vez más utilizados en aplicaciones de visión por computador, principalmente para reconstrucción de escenas en 3D, seguimiento de personas o detección de movimiento. Además, es muy posible que en un futuro más dispositivos cuenten con estos sensores gracias a proyectos como *Tango* de Google[2], que pretende incluir dispositivos RGBD en teléfonos móviles y tabletas, o la compra de Apple de PrimeSense, empresa encargada del desarrollo de la primera versión de *Kinect*.

1.3. Visión artificial en robótica

Para que un robot tenga información del entorno que le rodea pueden utilizarse diversas fuentes de información: sensores de distancia (ultrasonido, láser), sensores de posicionamiento (inclinómetros) u otros sensores como las cámaras; de estas últimas se puede obtener cierta información del mundo utilizando visión artificial.

A pesar de que obtener información mediante visión artificial tiene una gran complejidad, las cámaras son el sensor más utilizado en los robots para detectar lo que les rodea. Esto es debido a que es un sensor muy barato comparado con el resto y, además, la información que se puede obtener a partir de ellas es mucha.

Muchos de los robots de la sección anterior utilizan las cámaras como sensor principal. Éste es el caso de los coches autónomos, que necesitan la información que les proporcionan sus cámaras para poder ver las señales de tráfico o evitar a otros vehículos.

También existen robots comerciales basados en localización visual, como las aspiradoras de última generación de Dyson o de iRobot (*Roomba 980*). Estas aspiradoras utilizan

[2] `https://get.google.com/tango/`

localización visual para situarse dentro de una casa y poder determinar así qué recorrido sistemático realizar para limpiar la vivienda en el menor tiempo posible (Figura 1.5).

(a) (b)

Figura 1.5: Aspiradora con localización visual de la marca Dyson (a). Mapa del entorno generado por la aspiradora *Roomba 980* (b).

1.3.1. Áreas de aplicación

Hasta hace pocos años el uso de técnicas de visión artificial estaba prácticamente limitado a escenarios industriales o a entornos controlados en los que las condiciones de trabajo fuesen conocidas de antemano. Sin embargo, cada vez son más las aplicaciones en el mundo real donde los robots interactúan con su entorno usando visión de forma similar a como lo harían los humanos.

A continuación se muestran algunos ejemplos de aplicaciones reales que utilizan visión artificial en robots:

- Percepción de estímulos u objetos: Los robots pueden percibir mediante cámaras objetos de su entorno que les llevarán a tomar decisiones o a interactuar con ellos. En la Figura 1.6(a) se muestra uno de los retos de la *Darpa Robotics Challenge*, en el que un robot tenía que ser capaz de detectar una puerta y abrirla.

- Control visual: Según los datos obtenidos de sus cámaras los robots modelarán su comportamiento. Por ejemplo, así sucede cuando se utilizan brazos robotizados guiados por visión.

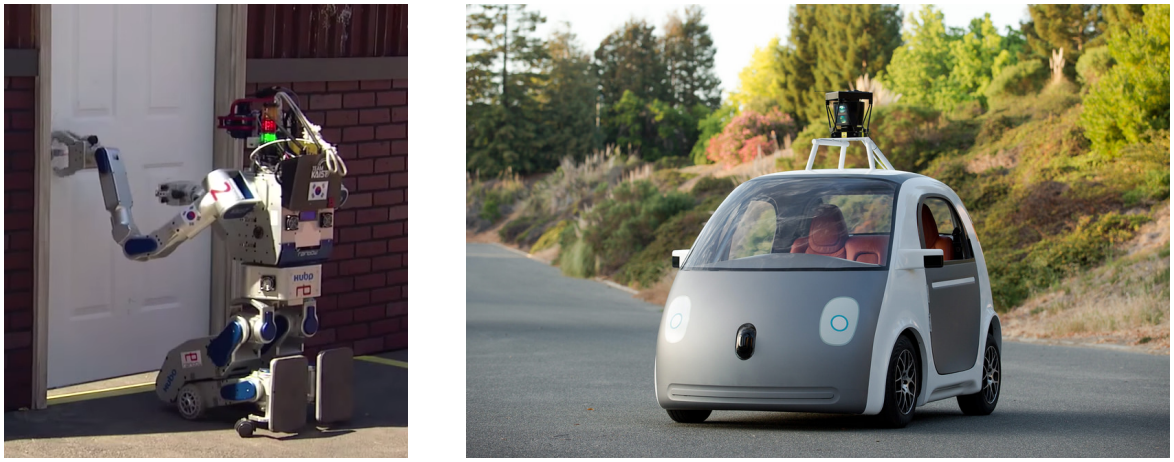
- Navegación: La información de las cámaras sirve como guía para que el robot se desplace en un entorno de forma segura, detectando tanto elementos que sirvan como

guía como posibles obstáculos. Ya existen hoy en día coches autónomos de diversos fabricantes que son capaces de ir a un destino sin la necesidad de intervención humana, por lo que todo hace prever que en un futuro cambiará la forma de movernos gracias a este tipo de vehículos (Figura 1.6(b)).

- Autolocalización visual: El robot puede calcular continuamente su posición en el entorno que le rodea a través de una o varias cámaras, sirviendo esta localización como base para realizar otro tipo de tareas. La autolocalización visual continua es la base principal de esta tesis y será tratada en profundidad a lo largo de este documento.

- Construcción de mapas: Gracias al desplazamiento que pueden realizar los robots móviles, pueden utilizarse sus cámaras para generar mapas 3D del entorno. Este tipo de aplicaciones suelen ir acompañadas de técnicas de autolocalización visual, las cuales permitirán construir el mapa 3D con precisión a medida que el robot se desplace.

- Atención visual: En este caso, las cámaras del robot sirven como base para decidir en qué partes del entorno centrar la atención del robot, lo que puede utilizarse para realizar seguimiento de objetos o para buscar zonas del entorno no exploradas anteriormente.



(a) (b)

Figura 1.6: Robot *Hubo* durante la DRC 2015 (a). Coche autónomo de Google (b).

1.4. Localización visual

Dentro de los robots que usan visión, uno de los problemas fundamentales en las aplicaciones robóticas es que el robot estime su ubicación en el entorno que le rodea. En muchas aplicaciones, el comportamiento del robot no será el correcto a menos que sepa situarse correctamente en el mundo, o que conozca su posición relativa respecto a otros objetos. Para esta tarea, pueden darse tres situaciones distintas:

1. Que el robot tenga un mapa del entorno que le rodea y conozca su posición inicial.
2. Que el robot tenga un mapa de su entorno pero no conozca su situación inicial.
3. Que el entorno sea desconocido y su posición inicial pueda ser supuesta en cualquier punto.

En los tres casos, el robot deberá hacer uso de sus sensores para extraer la información del entorno y compararla con el mapa del que disponga. Este mapa responderá a un modelo geométrico dado previamente, o podrá ser calculado por el propio algoritmo de localización en tiempo real. Además, una vez localizado, el robot tendrá que actualizar su posición en función de su desplazamiento.

La localización del robot puede ser calculada en dos dimensiones o en tres dimensiones, dependiendo del escenario de aplicación concreto que se aborde. En entornos en los que se utilicen robots con ruedas o humanoides de los que se conoce su geometría, bastará con calcular la posición del robot en dos dimensiones (puesto que ya se conoce su altura) y su orientación. Sin embargo, cuando se utilicen otros robots como drones o cámaras libres habrá que calcular la localización con seis grados de libertad: tres para su desplazamiento y tres para sus orientaciones.

Hay que tener en cuenta que la localización estimada del robot nunca tendrá certeza absoluta, por lo que siempre habrá que representar la incertidumbre de esa estimación, utilizando por ejemplo distribuciones de probabilidad que indiquen cómo de fiable es la estimación. Además, la localización del robot no suele ser un fin en sí mismo, sino que sirve como base para otras aplicaciones, por lo que dependiendo del problema a resolver se necesitará una localización más o menos precisa.

El problema de la localización está estrechamente ligado al de la reconstrucción de mapas. Así, sin una buena localización no es posible crear un mapa de un entorno desconocido y sin este mapa no es posible localizar al robot en el mundo en coordenadas

absolutas. Este problema se ha abordado desde dos campos de investigación distintos: el de la visión artificial, donde se le conoce con el nombre de *Structrure From Motion*, y el de la comunidad robótica, donde se le llama SLAM (*Simultaneous Localization And Mapping*). Independientemente del nombre, en él se han centrado las investigaciones realizadas en esta tesis doctoral.

1.4.1. Conceptos

Hay una serie de conceptos relacionados con localización visual que conviene definir con precisión antes de entrar en los algoritmos, puesto que se hará uso de ellos a lo largo de este tesis:

- **Calidad**: Se considera en esta tesis que la calidad de la autolocalización depende de tres factores: la eficiencia temporal, la precisión espacial del algoritmo y la robustez.
- **Eficiencia**: La eficiencia será medida como el tiempo de ejecución de cada iteración del algoritmo. Los algoritmos deberán al menos funcionar en tiempo real, es decir, tendrán que ser capaces de realizar al menos 30 iteraciones por segundo (FPS).
- **Precisión**: La precisión vendrá determinada por el error lineal y el error angular entre la posición estimada y la posición real. El error lineal será medido como la distancia euclídea entre las dos posiciones, mientras que el error angular se calculará mediante la distancia angular de ambas orientaciones, teniendo en cuenta la naturaleza circular de los ángulos y la discontinuidad numérica en 2π.
- **Robustez**: El algoritmo de localización será robusto siempre que sea capaz de seguir funcionando con normalidad ante situaciones imprevistas (como oclusiones, mala calidad de la imagen, secuestros, movimiento de objetos en la escena, etc).
- **Hipótesis múltiples**: Los algoritmos pueden tener la capacidad de manejar simultáneamente múltiples hipótesis candidatas como solución al problema de localización. Esto se producirá especialmente en aquellos entornos que tengan simetrías, es decir, que algunas partes del entorno sean visualmente iguales o similares para el algoritmo.
- **Oclusiones**: Se producirá una oclusión cuando la cámara del robot esté tapada total o parcialmente, de modo que no sea posible utilizar la parte de la imagen ocluida para extraer información. Los algoritmos tendrán que tener en cuenta que pueden existir oclusiones y que éstas condicionan sus resultados.

- **Secuestros**: Un secuestro tendrá lugar cuando el robot sea desplazado deliberadamente por un tercero, de forma que su localización anterior ya no sea válida. Los algoritmos deberán detectar los secuestros e intentar localizarse de nuevo una vez que el robot sea situado en otro lugar.

- **Localización Absoluta**: En un mundo con un mapa conocido, la localización absoluta consistirá en estimar la posición del robot dentro del mapa en coordenadas respecto del origen de referencia absoluto de ese mapa.

- **Localización Incremental**: En entornos con mapa desconocido, la localización del robot se establecerá de forma incremental respecto de posiciones previas pasadas (por ejemplo en el instante anterior), lo que dará lugar a un error en la localización incremental que aumentará con el tiempo.

- **Dinamismo de la escena**: El movimiento de los objetos en la escena suele interferir con las estimaciones de autolocalización visual, pues no todos los desplazamientos en las imágenes se deben entonces al movimiento propio del robot.

- **Cierre de bucle**: Se producirá un cierre de bucle cuando el robot vuelva tras un periodo de tiempo a una zona del mundo que ya haya visitado anteriormente. Por ejemplo, en el caso de un recorrido que vuelva al punto de origen, se podrá determinar el error que se ha producido comparando la posición real en el origen con la estimada por el algoritmo de localización.

- **Relocalización**: Consiste en recuperarse de una pérdida o secuestro volviendo a estimar correctamente la posición absoluta del robot dentro del mapa. Esta recuperación puede realizarse desde la incertidumbre total o desde una creencia errónea sobre la posición propia.

También es conveniente matizar que los conceptos de precisión y robustez dependerán del entorno en el que se encuentre el robot y la aplicación final en la que se integra la autolocalización. No se requiere la misma precisión para reconstruir un mapa en 3D (deberá ser de pocos centímetros) que para guiar a una persona por un edificio (podrá ser incluso de metros). De la misma forma, tampoco se precisará la misma robustez en un entorno totalmente controlado (como el campo de la RoboCup) que en mitad de una ciudad, donde pueden darse todo tipo de situaciones.

Para evaluar, validar y comparar la eficiencia temporal, la precisión y la robustez de los algoritmos desarrollados en esta tesis se emplearán como referencia otros algoritmos

ampliamente utilizados, así como imágenes procedentes de bases de datos públicas internacionales, tal y como se verá en el los capítulos 3, 5 y 7. Así, se podrá realizar una comparación objetiva de los resultados obtenidos frente a otras implementaciones del estado del arte.

1.4.2. Convenios de representación

En esta tesis, las posiciones en 3D utilizarán coordenadas homogéneas (X, Y, Z, H), puesto que presentan ventajas a la hora de realizar operaciones geométricas frente a las coordenadas cartesianas. Si bien, el resultado generado por los algoritmos de localización se realizará en coordenadas cartesianas (X, Y, Z).

Por su parte, las orientaciones se representarán utilizando cuaterniones, puesto que proporcionan una representación más compacta y rápida que otro tipo de representaciones, como los ángulos de Euler. No obstante, en los casos en los que se calcule la localización en dos dimensiones, se podrá simplificar la representación de la orientación utilizando un único ángulo de Euler.

Por último, la incertidumbre de la localización podrá ser representada por una matriz de covarianza o mediante una función de probabilidad (FDP).

1.5. Objetivos

Una vez expuesto el contexto en el que se desarrolla el presente trabajo y su motivación, se detallan a continuación los objetivos concretos que se pretenden alcanzar con esta tesis.

1.5.1. Descripción del problema

El principal objetivo de este trabajo es ofrecer aportes en la autolocalización de robots tanto en entornos conocidos como desconocidos, utilizando como sensor una sola cámara RGB y haciendo uso de técnicas de visión artificial.

El problema general a abordar es el de un robot con cámara moviéndose por su entorno. Los algoritmos desarrollados deberán ofrecer una estimación continua de la posición del robot; la localización estimada deberá ser robusta, precisa y con tiempos de cómputo que permitan su utilización en procesadores poco potentes, como los que se encuentran

en robots móviles. Además, los algoritmos serán validados experimentalmente con robots reales y se compararán con otros algoritmos existentes en el estado del arte.

Para la localización del robot habrá que hacer un amplio uso de la visión artificial, utilizando para ello únicamente una cámara. No se utilizarán cámaras con profundidad (RGBD), puesto que tienen un coste algo mayor que las cámaras tradicionales y limitarían el uso de los algoritmos en escenarios de exteriores o en robots como los drones, que no pueden levantar su peso.

Cabe destacar que el uso de una sola cámara como sensor principal aumenta la complejidad del problema, al no hacer uso de otro tipo de sensores que permitan medir distancias con más fiabilidad (como sensores láser, cámaras RGBD o cámaras en estéreo). Sin embargo, y como punto favorable, el uso de una sola cámara permitirá emplear los algoritmos desarrollados en un amplio número de dispositivos de uso corriente, como teléfonos móviles, drones, robots modestos, etc; además de poder ser utilizados tanto en interiores como en exteriores.

A la hora de articular los subobjetivos del proyecto, hay que distinguir entre los dos problemas de localización que se pretenden abordar; por un lado, que el robot se localice en un *mundo conocido*, con un mapa dado previamente (a), y por otro, la localización en *entornos desconocidos* sin ningún tipo de información previa (b).

Para la localización en entornos con mapa conocido, se organiza el objetivo global en los siguientes subobjetivos:

a1. Diseñar y desarrollar un modelo de observación visual que incluya un análisis 2D de la imagen adaptado a cada escenario y que extraiga información espacial basándose en el mapa.

a2. Utilizando el mapa del mundo disponible, diseñar y desarrollar un algoritmo que combine la información de la última observación instantánea con la acumulada hasta ese momento. El algoritmo deberá ser capaz de manejar simultáneamente múltiples hipótesis cuando la información recibida no permita asegurar de forma fiable la localización del robot.

a3. Validar experimentalmente el algoritmo en distintos escenarios reales para verificar su comportamiento ante imprevistos, como oclusiones, secuestros, etc.

Para la localización en entornos desconocidos, los subobjetivos serán los siguientes:

b1. Diseñar y desarrollar un algoritmo que genere un mapa del entorno que rodea al robot y lo amplíe progresivamente a medida que se exploren nuevas zonas. El mapa generado será no denso, ya que su función principal será servir como base para el seguimiento, quedando fuera de esta tesis la generación de mapas densos que tengan otras metas (como la reconstrucción 3D de alta calidad).

b2. Diseñar y desarrollar un algoritmo capaz de localizarse incrementalmente dentro del mapa generado hasta ese momento realizando un seguimiento (*Tracking*) de los elementos del mismo.

b3. Validar el *software* experimentalmente en escenarios reales, caracterizando su funcionamiento y comparándolo con el de otros algoritmos disponibles.

Queda fuera de esta tesis la fusión de la información de las cámaras con la de otros sensores de posición, como GPS o sensores inerciales (IMU).

Otro factor a tener en cuenta es que los escenarios utilizados serán entornos reales sin ninguna restricción, por lo que podrán existir elementos dinámicos en el entorno y la cámara podrá sufrir oclusiones, cambios de luminosidad, secuestros, etc.

Además, no habrá tampoco ninguna restricción en cuanto a los grados de libertad de los robots, por lo que podrán desplazarse en cualquiera de los seis grados de libertad existentes: tres grados de traslación y otros tres de rotación.

1.5.2. Requisitos

Teniendo en cuenta los objetivos de la sección anterior, el proyecto deberá alcanzar los siguientes requisitos:

- *Eficiencia de los algoritmos*: Los algoritmos utilizados deberán ser lo suficientemente eficientes como para ser ejecutados en procesadores relativamente modestos, como los que se encuentran en robots y dispositivos móviles, no pudiendo hacer uso de otros elementos más potentes, como GPU. Por ello, los algoritmos deben estar altamente optimizados para que funcionen en tiempo real. Esto condiciona enormemente los algoritmos diseñados pero aumenta significativamente su utilidad.

- *Precisión*: La precisión deseada en los algoritmos desarrollados dependerá de la aplicación final, pero se asumirán como referencias aceptables los errores del orden de centímetros. Hay que distinguir entre la precisión necesaria para los algoritmos

con mapas conocidos, donde un error medio de localización en 3D aceptable estaría por debajo de 30 cm, y la precisión requerida en algoritmos con mapas desconocidos, donde debe ser mucho menor.

- *Robustez*: Los algoritmos perceptivos desarrollados deberán ser capaces de responder correctamente ante imprevistos externos, como cambios en la iluminación, oclusiones o secuestros, afectando lo menos posible a la localización.

Estos tres requisitos se podrán satisfacer en mayor o menor medida, una vez superados unos umbrales mínimos de aceptación. La combinación de todos ellos da una medida de la calidad de los algoritmos diseñados y desarrollados.

1.6. Estructura de la tesis

Tras esta introducción, en los siguientes capítulos se expondrá en detalle el trabajo de investigación realizado para esta tesis. Primero, se revisará el estado del arte actual de las técnicas para la localización de robots en entornos conocidos y desconocidos. Posteriormente, en el capítulo 3 se explicarán los escenarios de experimentación empleados para dar un contexto mayor al problema y se describirán las herramientas *software* empleadas en la parte experimental de esta tesis.

Esta tesis aborda un problema común: la localización de robots con cámaras; pero que cuenta con dos soluciones distintas en función de las características del entorno: escenarios con mapas conocidos y con mapas desconocidos. Por esta razón, se dedicarán capítulos distintos a cada una de estas variantes. Así, en los capítulos 4 y 5 se detallarán los algoritmos desarrollados para el caso de entornos con mapas conocidos y se mostrarán sus experimentos de validación; en los capítulos 6 y 7 se hará lo propio con los algoritmos en entornos con mapas desconocidos.

Para finalizar, se detallarán las conclusiones que se desprenden del trabajo realizado y se plantearán posibles líneas de investigación futuras.

Capítulo 2

Estado del arte

El presente capítulo recoge un conjunto representativo e ilustrativo de las técnicas de localización visual existentes, que han servido como base para los algoritmos desarrollados en esta tesis.

Se han agrupado los algoritmos del estado del arte en dos familias: algoritmos para localización visual en entornos con mapas conocidos y algoritmos para entornos desconocidos. Casi todas estas técnicas presentan tres elementos fundamentales: geometría, probabilidad y optimización. Combinando adecuadamente estos elementos se aborda de distinta forma un problema común: localizar a un robot en un escenario a través de su cámara.

2.1. Localización visual con mapas conocidos

El primer tipo de localización visual es aquel en el que se dispone de un mapa conocido del entorno en el que está situado el robot. La definición de mapa en este caso es abierta, pudiendo consistir tanto en información detallada de la geometría del entorno como en balizas u otro tipo de elementos que sirvan de apoyo para la localización.

Una posibilidad para determinar la posición del robot en un mundo conocido es utilizar la localización probabilística. Así, empleando la información que proporcionan los sensores, la probabilidad de que el robot se encuentre en una posición estará delimitada entre 0 y 1, siendo 0 cuando es seguro que el robot no se encuentra en esa posición y 1 cuando se tenga la certeza absoluta de que sí está en la posición dada. La probabilidad de cada posición se acumula a lo largo del tiempo con los datos de entrada proporcionados por los sensores.

En esta sección se verán varias técnicas probabilísticas que han sido utilizadas con éxito a la hora de localizar robots: los filtros de Kalman, los modelos de Markov y los filtros de partículas. Estas técnicas probabilísticas manejan una función de probabilidad (FDP) para estimar la posición del robot, asignando probabilidades a las distintas posiciones del mundo.

2.1.1. Localización visual basada en geometría

La localización basada en geometría consiste en determinar con cierta certeza en qué posición se encuentra un robot en un momento concreto a partir de su información sensorial. En esta tesis, la información geométrica del entorno se obtendrá utilizando como sensor una cámara RGB.

Modelo de cámara Pin-Hole

Para poder extraer información espacial de las imágenes es útil emplear un modelo de cámara que simplifique la complejidad física de las lentes de las cámaras. Disponer de un modelo de cámara preciso y correctamente calibrado también es importante para que los algoritmos de visión puedan extraer correctamente la información del mundo que rodea a los robots.

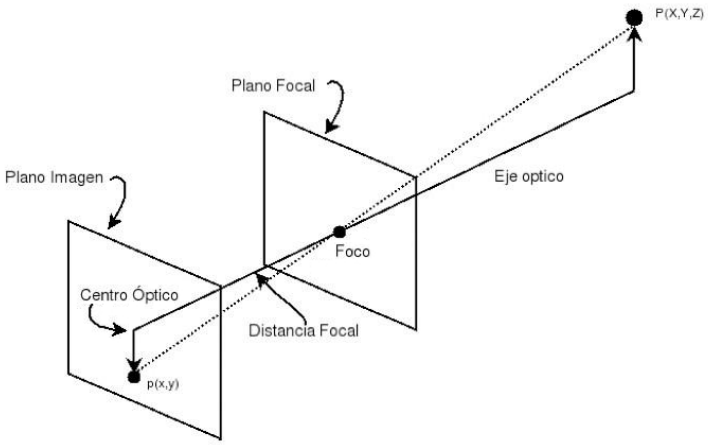


Figura 2.1: Modelo de cámara Pin-Hole.

En esta tesis se utiliza el modelo de cámara Pin-Hole (Figura 2.1); este modelo asume que todos los rayos de luz pasan por el mismo punto (foco) e impactan en un plano formando una imagen. La distancia que separa el plano focal y el plano imagen toma el nombre de distancia focal, de la que depende el tamaño de los objetos observados en la imagen, siendo más pequeños cuanto mayor sea la distancia focal. Este modelo funciona con las cámaras

actuales porque las lentes delgadas ideales son capaces de concentrar los rayos de luz en un solo punto.

Los parámetros intrínsecos de la cámara definen las condiciones de formación de las imágenes y permiten relacionar la información espacial con las imágenes a través de una matriz de calibración. La matriz de calibración K está formada por la distancia focal en cada uno de los ejes (f_x y f_y) y la posición del centro óptico en la imagen (u_0 y v_0), como se muestra en la ecuación 2.1:

$$K = \begin{bmatrix} f_x & 0 & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.1}$$

Este modelo de cámara también cuenta con parámetros extrínsecos, que representan la posición y orientación de la cámara en el mundo y que normalmente estarán definidos mediante una matriz de Rotación-Traslación (RT) de 3x4.

Así, se proyecta un punto 3D en coordenadas homogéneas $(X, Y, Z, 1)$ en la imagen (u, v) mediante la ecuación 2.2:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{2.2}$$

Otra forma de realizar esta proyección consiste en transformar el punto 3D (X, Y, Z) en coordenadas absolutas a un punto 3D (x, y, z) en coordenadas relativas a la cámara (ecuación 2.3) y posteriormente realizar la proyección utilizando la matriz K, siempre que z sea distinto de 0 (ecuación 2.4):

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix} \tag{2.3}$$

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} u_0 \\ v_0 \end{bmatrix} + \begin{bmatrix} f_x & 0 \\ 0 & f_y \end{bmatrix} \begin{bmatrix} \frac{x}{z} \\ \frac{y}{z} \end{bmatrix} \tag{2.4}$$

El modelo Pin-Hole asume que la cámara no tiene lente y por tanto no existe distorsión en las imágenes; sin embargo, en la práctica muchas lentes cuentan con distorsión radial y

tangencial; en este caso, el modelo puede seguir utilizándose si se realiza una rectificación previa de la imagen [Hecht and Zajac, 1974] [Ojanen, 1999].

Perspective-n-Point

Se conoce con el nombre de *Perspective-n-Point* (PnP) al problema de estimar la posición de una cámara calibrada dado un conjunto de n puntos en 3D y sus correspondientes proyecciones en la imagen (Figura 2.2). Este problema fue introducido por primera vez por [Fischler and Bolles, 1981] y, desde entonces, ha encontrado numerosas aplicaciones en el campo de la visión por computador.

Dependiendo del número de puntos de los que se dispongan existen distintas soluciones:

- Dos puntos o menos: Número de soluciones infinitas.
- Tres puntos: Ocho posibles soluciones, de las cuales solo cuatro están frente a la cámara [Haralick *et al.*, 1994].
- Cuatro puntos: Si los cuatro puntos son coplanares solo existe una solución, en cualquier otro caso hay dos soluciones [Horaud *et al.*, 1989].
- Cinco puntos o más: Puede llegarse a una solución con mayor o menor error dependiendo de la técnica que utilicemos. La técnica clásica consiste en utilizar mínimos cuadrados [Hesch and Roumeliotis, 2011], pero también existen otras más complejas que buscan aumentar la eficiencia y la precisión [Lepetit *et al.*, 2009] [Li *et al.*, 2012].

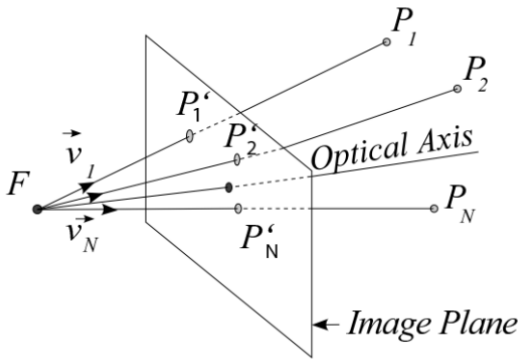


Figura 2.2: Estimación de posición mediante el algoritmo PnP.

2.1.2. Filtros de Kalman

El filtro de Kalman (KF) es un algoritmo probabilístico desarrollado por Rudolf E. Kalman [Kalman, 1960] [Kalman and Bucy, 1961] que permite estimar el estado de un sistema dinámico a partir de observaciones que pueden contener ruido. El KF tiene numerosas aplicaciones en ingeniería, tales como seguimiento, control de vehículos, procesamiento de señales o econometría.

La FDP que maneja el KF es monomodal, donde el estado se considera una variable aleatoria Gaussiana. El KF solo puede aplicarse con sistemas lineales con unas mediciones y ruido que sigan una distribución Gaussiana; si estas condiciones se cumplen, el filtro es capaz de ofrecer una estimación del estado del sistema y una medida de la incertidumbre con la que se ha realizado la estimación, todo ello de manera computacionalmente muy eficiente.

Los filtros de Kalman están basados en sistemas dinámicos discretizados en el tiempo. El estado del sistema es una variable n-dimensional x_t. Además, la observación del sistema en cada iteración es una variable l-dimensional z_t que solo depende de x_t.

Para utilizar el KF, es necesario especificar una serie de parámetros (matrices) para adaptarlo al problema que se esté abordando:

- F_t: El modelo de transición de estado.
- H_t: El modelo de observación.
- Q_t: La covarianza del ruido del proceso.
- R_t: La covarianza del ruido de la observación.

El filtro de Kalman asume que el estado real x_t evoluciona desde el estado en $(t-1)$ mediante la ecuación 2.5:

$$x_t = F_t x_{t-1} + w_t \tag{2.5}$$

siendo F_t el *modelo de transición de estado* que se aplica al estado previo x_{t-1} y w_t el ruido del proceso con covarianza Q_t.

Por otra parte, asume que la observación z_t en la iteración t del estado real x_t se realiza de acuerdo a la ecuación 2.6:

$$z_t = H_t x_t + v_t \tag{2.6}$$

donde H_t es el *modelo de observación* que convierte el espacio del estado en el espacio observado, mientras que v_t representa el ruido de la observación con covarianza R_t.

El KF es un estimador recursivo, es decir, solo utiliza para la estimación del estado actual el estado de la iteración anterior y la observación actual, sin tener en cuenta las observaciones anteriores ni otros estados previos.

El vector de estado x_t sigue una distribución Gaussiana n-dimensional $\hat{x}_t$ con una matriz de covarianza P. A su vez, el vector de observación sigue una distribución Gaussiana l-dimensional $H_t\hat{x}_{t|t-1}$ que tiene una matriz de covarianza S.

La estimación del estado en cada iteración se realiza en dos pasos: el primer paso es conocido como "predicción" y se encarga de predecir el estado $\hat{x}_{t|t-1}$ y su covarianza $P_{t|t-1}$, mientras que el segundo paso se conoce como "actualización", donde se utiliza la observación para refinar el estado estimado y así calcular $\hat{x}_t$ y P_t.

A continuación se muestra el algoritmo completo:

Predicción

Predicción del estado (pronóstico)	$\hat{x}_{t\|t-1} = F_t\hat{x}_{t-1}$
Predicción de la covarianza	$P_{t\|t-1} = F_t P_{t-1} F_t^T + Q_t$

Actualización

Innovación o residuo	$\tilde{y}_t = z_t - H_t\hat{x}_{t\|t-1}$
Covarianza del residuo	$S_t = H_t P_{t\|t-1} H_t^T + R_t$
Ganancia de Kalman	$K_t = P_{t\|t-1} H_t^T S_t^{-1}$
Estimación actualizada del estado	$\hat{x}_t = \hat{x}_{t\|t-1} + K_t + \tilde{y}_t$
Covarianza actualizada del estado	$P_t = (I - K_t H_t) P_{t\|t-1}$

Filtro Extendido de Kalman

Una de las limitaciones del filtro de Kalman clásico es que necesita que el sistema sea lineal, no pudiéndose por tanto utilizar cuando las ecuaciones de estado u observación (o ambas) no son lineales.

A costa de resultar menos óptimo, el filtro de Kalman puede también utilizarse para un número mayor de sistemas dinámicos. Esta variación del algoritmo, que se conoce con

el nombre de filtro extendido de Kalman (EKF) [Jazwinski, 1970], consiste en realizar una aproximación linealizando las matrices F_t y H_t en torno al vector de estado actual, sustituyendo estas matrices por sus jacobianas JF_t y JH_t.

Estas nuevas matrices JF_t y JH_t solo deben sustituirse cuando operen sobre matrices; en caso de operar sobre vectores (en la predicción de x_t y en el cálculo del residuo) es preferible utilizar las funciones $f()$ y $h()$ con las operaciones originales, ya que la aproximación conlleva una pérdida de exactitud en la medición.

Estas modificaciones dan como resultado el siguiente algoritmo:

Predicción	
Predicción del estado (pronóstico)	$\hat{x}_{t\|t-1} = f(\hat{x}_{t-1})$
Predicción de la covarianza	$P_{t\|t-1} = JF_t P_{t-1} JF_t^T + Q_t$
Actualización	
Innovación o residuo	$\tilde{y}_t = z_t - h(\hat{x}_{t\|t-1})$
Covarianza del residuo	$S_t = JH_t P_{t\|t-1} JH_t^T + R_t$
Ganancia de Kalman	$K_t = P_{t\|t-1} JH_t^T S_t^{-1}$
Estimación actualizada del estado	$\hat{x}_t = \hat{x}_{t\|t-1} + K_t + \tilde{y}_t$
Covarianza actualizada del estado	$P_t = (I - K_t JH_t) P_{t\|t-1}$

El EKF ya no es un estimador óptimo, y es menos óptimo cuanto menos lineales sean las funciones de estado y de observación. Además, el EKF cuenta con otras desventajas como que puede no ser estable cuando la hipótesis de partida no es correcta (no llegando nunca a aproximar correctamente el vector de estado) y que tiende a subestimar la matriz de covarianza.

Estos algoritmos han sido utilizados para la localización de robots en entornos conocidos [Wang, 1988] [Baltzakis and Trahanias, 2002] [Kiriy and Buehler, 2002] y también han sido aplicados con éxito en la localización de robots en el escenario de la RoboCup [Quinlan and Middleton, 2009] [Jochmann *et al.*, 2011].

Filtro de Kalman *Unscented*

En ocasiones, las matrices F_t y H_t son altamente no linealizables, dando el EKF un rendimiento muy bajo; en estos casos, suele utilizarse el filtro de Kalman *Unscented* (UKF) [Wan and Van Der Merwe, 2000]. Este algoritmo emplea una técnica de muestreo determinista (conocida como transformada *unscented*) para seleccionar un conjunto de muestras alrededor de la media.

El UKF tiene dos ventajas principales frente al EKF: no necesita calcular jacobianas y además obtiene un estado y covarianzas más precisos respecto a los reales.

Para un estado x con L parámetros, cuya predicción de estado es $\hat{x}$ y tiene una covarianza P, el UKF necesita calcular $2L+1$ estados χ_i (llamados vectores "sigmas") y sus correspondientes pesos $W_i^{(m)}$ y $W_i^{(c)}$ con la siguiente formulación:

$$\begin{cases} \chi_i = \hat{x} \text{ si } i = 0 \\ \chi_i = \hat{x} + (\sqrt{(L+\lambda)P})_i \text{ si } i = 1, ..., L \\ \chi_i = \hat{x} - (\sqrt{(L+\lambda)P})_{i-L} \text{ si } i = L+1, ..., 2L \end{cases}$$
$$\begin{cases} W_i^{(m)} = \frac{\lambda}{L+\lambda} \text{ si } i = 0 \\ W_i^{(m)} = \frac{1}{2(L+\lambda)} \text{ si } i = 1, ..., 2L \end{cases}$$
$$\begin{cases} W_i^{(c)} = \frac{\lambda}{L+\lambda} + (1 - \alpha^2 + \beta) \text{ si } i = 0 \\ W_i^{(c)} = \frac{1}{2(L+\lambda)} \text{ si } i = 1, ..., 2L \end{cases}$$

siendo $\lambda = \alpha^2(L+\kappa) - L$ un factor de escala. El valor de α determina cómo será el reparto de cada sigma alrededor de $\hat{x}$, soliéndose utilizar valores bajos (por ejemplo 10^{-4}). Por su parte, κ es un segundo factor de escala que suele valer 0 y β determina el conocimiento a priori de la distribución de x, con un valor óptimo de 2.

Con estos parámetros, el paso de predicción del UKF queda como sigue:

$$\hat{x}_{t|t-1} = \sum_{i=0}^{2L} W_i^{(m)} F(\chi_i)$$
$$P_{t|t-1} = \sum_{i=0}^{2L} (W_i^{(c)} (F(\chi_i) - \hat{x}_{t|t-1})(F(\chi_i) - \hat{x}_{t|t-1})^T) + Q_t$$

donde $\hat{x}$ es el vector de estado, P su matriz de covarianza asociada y Q la matriz de error de covarianza. Por otro lado, el paso de actualización se calcula con las siguientes ecuaciones:

$$Y_t = \sum_{i=0}^{2L} W_i^{(m)} H(\chi_i)$$
$$S_t = \sum_{i=0}^{2L} (W_i^{(c)} (H(\chi_i) - Y_t)(H(\chi_i) - Y_t)^T) + R_t$$
$$C_t = \sum_{i=0}^{2L} (W_i^{(c)} (\chi_i - \hat{x}_{t|t-1})(H(\chi_i) - Y_t)^T)$$
$$K_t = C_t S_t^{-1}$$
$$\hat{x}_t = \hat{x}_{t|t-1} + K_t(z_t - Y_t)$$
$$P_t = P_{t|t-1} - K_t S K_t^T$$

siendo Y el vector de predicción medio, S la matriz de covarianza de la predicción, C la matriz de covarianza cruzada estado-predicción, K la matriz de covarianza de Kalman, z el vector de observación y R la matriz de error de la observación.

El UKF ha sido empleado para abordar la localización visual de robots con sistemas no lineales, tratando de obtener mejores resultados que con el EKF [Sunderhauf *et al.*, 2007] [Holmes *et al.*, 2009].

2.1.3. Modelos de Markov

Los modelos de Markov [Simmons and Koenig, 1995] son algoritmos probabilísticos que mantienen una FDP discretizada del espacio de trabajo con todas las posibles soluciones al problema dado. Se trata de un proceso estocástico en el que el estado siguiente X_{t+1} solo depende del estado actual X_t, lo que se conoce como propiedad de Markov. Esto implica que el sistema no tiene memoria de cómo se llegó a X_t, puesto que se entiende que este estado resume toda la historia pasada (ecuación 2.7):

$$P(X_{t+1} = x_{t+1}|X_t = x_t, X_{t-1} = x_{t-1}, \ldots, X_1 = x_1) = P(X_{t+1} = x_{t+1}|X_t = x_t) \qquad (2.7)$$

Los modelos de Markov de orden m son aquellos en los que el estado futuro depende de los m estados anteriores. El caso anterior se corresponde con un modelo de Markov de orden 1, puesto que el estado siguiente solo depende del estado inmediatamente anterior.

Estas técnicas son más generales que los filtros de Kalman, permitiendo representar FDP multimodales y funcionando en sistemas no lineales. Sin embargo, necesitan bastante potencia de cómputo al representar explícitamente la FDP en todo su dominio (aunque esté discretizado), de forma que no son operativos cuando el tamaño del dominio es grande o tiene muchas dimensiones.

Localización Markoviana

Para estimar la localización de un robot en el mundo utilizando modelos de Markov se divide el mundo en una rejilla con celdas con cada una de las posibles posiciones del robot, donde cada celda (i, j) cuenta con una probabilidad $P(i, j)$ asociada que representa la probabilidad de que el robot se encuentre en esa posición, siendo la suma de las probabilidades de todas las celdas igual a 1 (Figura 2.3).

Si no se dispone de información a priori sobre la localización del robot, se inicializan todas las celdas mediante una distribución uniforme. A medida que se obtienen nuevos datos, se modifica la probabilidad de cada celda en función de éstos, de forma que las celdas con posiciones plausibles con los datos aumentan su valor y el resto disminuyen. Cuando

la mayoría de la probabilidad se centre alrededor de una celda, se habrá encontrado la posición del robot en el mundo con cierta certidumbre.

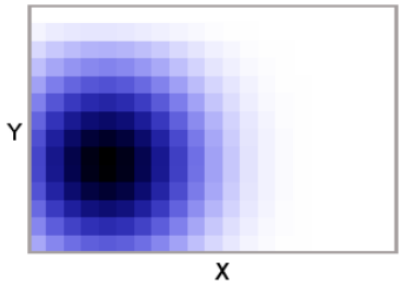


Figura 2.3: Rejilla probabilística de dos dimensiones para estimar la posición de un robot.

Para actualizar el valor de cada celda, la localización Markoviana utiliza tanto la información que le proporcionan sus sensores (las observaciones en el caso de las cámaras) como el modelo de movimiento del robot, que normalmente se obtiene a través de su odometría.

Las ventajas principales de la localización Markoviana son que permite dotar al sistema con información a priori inicializando las celdas de manera no uniforme y que, al contrario de lo que sucedía con los filtros de Kalman, el sistema puede relocalizarse. Sin embargo, el mayor problema de estas técnicas es su baja eficiencia temporal, ya que el tiempo de ejecución aumenta proporcionalmente al número de soluciones al problema. Esto hace que en la práctica su utilización no sea factible en algoritmos que deban funcionar en tiempo real a no ser que el espacio de estados sea de tamaño reducido.

Los modelos de Markov han sido utilizados para la localización de robots utilizando tanto sensores de profundidad [Fox *et al.*, 1999a] como cámaras [Kröse *et al.*, 2001]. Además, también han sido empleados para la localización visual de robots en la RoboCup combinándolos con filtros de Kalman [Gutmann, 2002] [Martín *et al.*, 2007].

2.1.4. Métodos de Monte Carlo

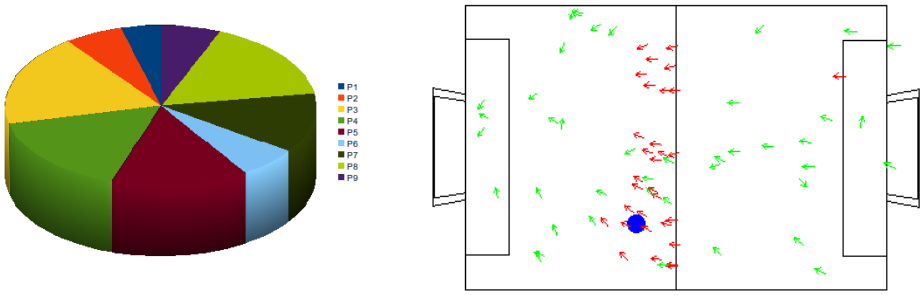
Los métodos de Monte Carlo (MC) [Fox *et al.*, 1999b] [Dellaert *et al.*, 1999] son algoritmos probabilísticos que utilizan la aleatoriedad para encontrar una solución óptima de forma eficiente, representando la FDP de modo muestreado. Estos algoritmos suelen conocerse también con el nombre de "filtros de partículas", ya que utilizan un número P de elementos, llamados partículas.

Los algoritmos de Monte Carlo funcionan de forma iterativa y deben contar con dos mecanismos fundamentales:

- Modelo probabilístico de observación: Mecanismo utilizado para evaluar cada partícula, que da como resultado una probabilidad entre 0 y 1.
- Remuestreo: Procedimiento donde se generan las partículas de la siguiente iteración teniendo en cuenta su probabilidad.

Remuestreo en Monte Carlo clásico

En el algoritmo de Monte Carlo clásico, la fase de remuestreo se realiza de forma aleatoria mediante el mecanismo conocido como "ruleta" [Dellaert *et al.*, 1999]. Esta técnica consiste en generar una ruleta como la que se ve en la Figura 2.4(a), donde el peso de cada partícula viene dado por su probabilidad, de modo que las partículas con mayor probabilidad tienen una porción más grande de la ruleta y, por tanto, tienen más posibilidades de ser seleccionadas.



(a) (b)

Figura 2.4: Selección por ruleta en Monte Carlo (a). Distribución de partículas en entorno de la RoboCup (b); posición real del robot en azul, partículas con mayor probabilidad en rojo.

Esta técnica genera muestras completamente aleatorias para la iteración t, usando la FDP$(t-1)$ como función extractora. En total se realizan P extracciones, añadiendo a cada partícula seleccionada un ruido Gaussiano. Así, se obtendrá una nueva generación de partículas que previsiblemente convergerá hacia la solución óptima (Figura 2.4(b)).

En caso de que haya degeneración de muestras, de modo que ya no sean representativas de la FDP, se remuestrea de manera uniforme para representar la total incertidumbre en la localización.

Monte Carlo Aumentado

El algoritmo de MC aumentado [Thrun *et al.*, 2001] varía la forma en la que se realiza el remuestreo, puesto que se tiene también en cuenta que las partículas no siempre convergen hacia la solución correcta.

Además de la ruleta utilizada en MC clásico, se dispone de un mecanismo adicional para distribuir de forma aleatoria P_A partículas (dentro de las P partículas del algoritmo) y cuyo objetivo es buscar nuevas posiciones donde pueda encontrarse el robot. Para calcular el número de partículas P_A que se distribuyen de forma aleatoria primero es necesario calcular la probabilidad media w_{avg} de las partículas, mediante la ecuación 2.8:

$$w_{avg} = \sum_{i=1}^{P} w_i \tag{2.8}$$

siendo w_i la probabilidad de cada partícula calculada mediante una función de coste. Con este valor se actualizan los pesos w_{slow} y w_{fast} en cada iteración. Estos pesos inicialmente valen 0 y modifican su valor con las ecuaciones 2.9 y 2.10:

$$w_{slow} = w_{slow} + \alpha_{slow}(w_{avg} - w_{slow}) \tag{2.9}$$

$$w_{fast} = w_{fast} + \alpha_{fast}(w_{avg} - w_{fast}) \tag{2.10}$$

determinando α_{slow} y α_{fast} cómo de reactivo es el algoritmo, siendo sus valores típicos 0.05 para α_{slow} y 0.4 para α_{fast}. Con esto, el número de partículas P_A que se distribuyen de forma aleatoria en una iteración viene dado por la ecuación 2.11:

$$P_A = P \cdot max(0, 1 - \frac{w_{fast}}{w_{slow}}) \tag{2.11}$$

Así, al producirse un descenso en la probabilidad media w_{avg}, el valor de w_{fast} decrece más rápido que el de w_{slow} y se crean más partículas aleatorias.

Monte Carlo Adaptativo

Este algoritmo, también conocido como *KLD-Sampling* [Fox, 2001], se caracteriza por modificar el número de partículas en cada iteración dinámicamente basándose en el error estimado mediante la divergencia de Kullback-Leibler [Kullback and Leibler, 1951].

Esta técnica intenta solucionar uno de los problemas principales de Monte Carlo, que el tiempo de cómputo de cada iteración es lineal con el número de partículas utilizadas. Al iniciarse el algoritmo se necesitan un gran número de partículas para poder cubrir la mayor superficie del mapa posible, sin embargo, una vez que el algoritmo ha convergido en una posición este alto número de partículas ya no es necesario y se desperdicia mucho tiempo manteniéndolo.

Con el algoritmo *KLD-Sampling* el número de partículas total se calcula en cada iteración, variando en función de la incertidumbre de la localización. Así, en cada iteración se determina el número de partículas tal que, con probabilidad $1 - \delta$, el error entre la aproximación estimada y la real sea menor que ϵ (siendo ambos parámetros configurables).

El algoritmo divide el espacio de estados (el mapa) en un conjunto discreto de elementos, generando así una rejilla con posibles soluciones (celdas). Estas celdas están inicialmente vacías y se llenan a medida que se generan partículas dentro de ellas.

Así, para seleccionar las partículas de la siguiente iteración, en lugar de utilizar directamente el mecanismo de ruleta visto anteriormente, el algoritmo va creando partículas hasta que su número llega a un máximo n_x. Este valor n_x se actualiza tras crear cada partícula mediante la ecuación 2.12:

$$n_x = \frac{k-1}{2\epsilon}\left\{1 - \frac{2}{9(k-1)} + \sqrt{\frac{2}{9(k-1)}}z_{1-\delta}\right\}^3 \tag{2.12}$$

siendo k el número de celdas no vacías, ϵ el error máximo y $z_{1-\delta}$ el cuantil de la distribución normal para $1 - \delta$.

Con las primeras partículas, el valor de k aumentará casi con cada nueva partícula, puesto que inicialmente todas las celdas estarán vacías, haciendo que el número de partículas deseadas n_x también aumente. A medida que se generen nuevas partículas, cada vez menos celdas estarán vacías y n_x solo aumentará ocasionalmente, terminando este procedimiento cuando el número total de partículas n alcance a n_x.

En las primeras iteraciones, las partículas estarán distribuidas en el mapa y el número de celdas utilizadas k será alto, haciendo que n_x y n sea grande. A medida que el algoritmo converja, la mayoría de partículas irán a parar a celdas ya utilizadas, por lo que k no crecerá y el número de partículas empleadas será bajo.

Este algoritmo evita tener que fijar un número predeterminado de partículas, calculándose el mismo en base al tamaño del mapa y a calidad de la localización.

Otras implementaciones de Monte Carlo

Además de las implementaciones del algoritmo de Monte Carlo que se han descrito, existen otras muchas versiones del algoritmo que, en su mayoría, son ligeras modificaciones a las anteriores, como por ejemplo *Markov Chain Monte Carlo* [Gilks and Berzuini, 2001] o *Multi Modal Monte Carlo* [Rudoy and Wolfe, 2006].

Los métodos de Monte Carlo, con sus diversas variaciones, han sido muy utilizados en localización visual, tanto en escenarios genéricos [Thrun *et al.*, 2001] [Wolf *et al.*, 2005] como en la RoboCup [Rofer and Jungel, 2003] [Heinemann *et al.*, 2006] [Burchardt *et al.*, 2011].

2.1.5. Algoritmos Evolutivos

A partir de 1950, los científicos comenzaron a estudiar los sistemas evolutivos que se pueden encontrar en la naturaleza, con el fin de utilizar este tipo de comportamientos como herramientas de optimización y para solucionar problemas de ingeniería. Uno de los investigadores que destacó a la hora de adaptar la evolución biológica a la computación fue John Holland, con numerosos trabajos de investigación destacados en este campo desde los años 70 [Holland, 1975] [Holland, 1992].

Los algoritmos evolutivos cuentan con una población de individuos que representan soluciones al problema dado; para valorar la calidad de cada individuo se utiliza una función de salud (o *fitness*) que determina la probabilidad de que un individuo pueda ser seleccionado para reproducirse.

El algoritmo evolutivo comienza con una población inicial de individuos con valores típicamente aleatorios; estos individuos son evaluados mediante la función de salud y se seleccionan para reproducirse aquellos que tengan una mayor puntuación. Los individuos seleccionados se reproducen para generan los individuos de la siguiente generación mediante operadores genéticos, que principalmente son dos:

- **Cruce**: A partir de dos o más padres de la población original se genera un hijo mezclando los parámetros de los padres.
- **Mutación**: Se realiza un cambio aleatorio en alguna de las propiedades del padre, introduciendo así nuevas características en la población que ampliarán el espacio de soluciones.

De esta forma, a lo largo de las generaciones se propagarán las propiedades de los mejores individuos y la población convergerá hacia la solución óptima del problema.

Los algoritmos evolutivos tienen una gran versatilidad y son utilizados con éxito en campos como la ingeniería, la física o la robótica, existiendo además diversas técnicas que se engloban dentro de los algoritmos evolutivos: los algoritmos genéticos, la optimización por enjambre de partículas o la optimización por colonias de hormigas. Estos algoritmos tienen una estructura similar a los filtros de partículas, pero sin ofrecer las garantías del marco probabilístico.

El problema de la autolocalización se puede plantear como un problema de optimización en el que los algoritmos evolutivos son aplicables. Así, el espacio de posibles soluciones es el espacio de todas las posibles localizaciones, dentro del cual la localización óptima será aquella en la que realmente se encuentra el robot. En la función de salud se incorpora la compatibilidad de las distintas soluciones con el flujo de observaciones sensoriales; los individuos más compatibles con las observaciones mantendrán una salud alta y los incompatibles se descartarán al tener baja salud. Gracias a los operadores genéticos se generan nuevas soluciones candidatas (individuos) que exploran diferentes zonas del espacio de soluciones (posibles localizaciones).

Este tipo de técnicas han sido ya utilizados para la localización robots, principalmente empleando sensores de profundidad [Moreno *et al.*, 2002] [Vahdat *et al.*, 2007].

2.2. Localización visual con mapas desconocidos

En esta sección se describen algunas de las técnicas existentes para localizar robots con una sola cámara en entornos desconocidos. Estos algoritmos se engloban dentro del término SLAM *Simultaneous Localization and Mapping*, donde el objetivo es estimar de forma eficiente la posición de un robot a la vez que se genera un mapa de su entorno.

El requisito fundamental de estos algoritmos es que deben calcular la posición de la cámara en tiempo real, por lo que tienen que estar optimizados.

2.2.1. Odometría visual

La odometría visual es el proceso de determinar incrementalmente la posición y orientación de un robot a través del análisis de las imágenes obtenidas. El término de odometría visual (*Visual Odometry* en inglés) fue utilizado por primera vez en 2004 por

Nister [Nister *et al.*, 2004]. Usualmente se ha utilizado el término "odometría" para referirse a la estimación del desplazamiento incremental realizado por robots con ruedas, por lo que se ha adaptado este término al uso de cámaras para calcular este desplazamiento. Para ello, el robot estima su desplazamiento incremental analizando de forma secuencial las observaciones de una o varias de sus cámaras.

Los algoritmos de odometría visual pueden ser utilizados por cualquier robot o dispositivo que disponga de una cámara (al contrario de la odometría clásica, que solo está disponible cuando se conoce la geometría del robot) y proporciona una alta precisión.

Por contra, estas técnicas son sensibles a la iluminación y necesitan que las imágenes obtenidas cuenten con cierta textura para poder extraer el desplazamiento realizado. Además, las imágenes deben ser capturadas de tal forma que la escena observada sea vista desde varios puntos de vista.

Visión monocular

La odometría visual con una sola cámara permite utilizar las técnicas de SLAM en un gran número de robots o dispositivos. Sin embargo, utilizar una sola cámara aumenta la complejidad del problema al no poder inferir de las imágenes la escala real de la escena que se observa.

Así, para poder calcular la distancia a la que se encuentran los objetos, es necesario utilizar dos o más observaciones de una misma cámara en instantes distintos observando la misma parte del entorno, tratando de llegar así a un problema similar al de la visión estéreo, que se trata más adelante. Si, además, se quiere obtener la escala real de la escena observada, habrá que conocer las medidas de alguno de los objetos que se observan [Castle *et al.*, 2010], utilizar otros sensores como referencia [McCann, 2011] o incluso emplear sensores RGBD, que ya proporcionan directamente profundidad.

Los algoritmos de odometría visual que utilizan visión monocular se dividen tres tipos:

- Basados en características: Buscan píxeles que destaquen en la imagen, como bordes o esquinas, y realizan un seguimiento puramente óptico de éstos en cada imagen [Mouragnon *et al.*, 2006] [Tardif *et al.*, 2008]. El desplazamiento de la cámara se obtiene minimizando el error de reproyección de los puntos emparejados, utilizando para ello algoritmos iterativos de optimización.

  Su ventaja principal es que existen detectores de píxeles y descriptores muy robustos, que permiten realizar el emparejamiento entre imágenes incluso cuando existen

grandes desplazamientos entre ellas. Por contra, los algoritmos de detección son muy dependientes de los umbrales que se establezcan y su precisión puede verse afectada por emparejamientos incorrectos, teniendo que mitigar estos falsos positivos utilizando un gran número de puntos.

- Métodos directos: Utilizan la información de intensidad de los píxeles de la imagen para determinar el desplazamiento global entre dos imágenes [Milford and Wyeth, 2008] [Lovegrove *et al.*, 2011].

  Este tipo de métodos funcionan mejor que los basados en características cuando las imágenes no tienen mucha textura o en caso de imágenes borrosas, puesto que utilizan toda la información de la imagen, aunque son menos precisos en otros casos.

  Además, aunque se tarda más en calcular el error fotométrico que el error de reproyección de los algoritmos basados en características, se ahorra mucho tiempo de cómputo al no tener que extraer puntos y calcular los emparejamientos.

- Métodos híbridos: Utilizan una mezcla de los dos métodos anteriores [Scaramuzza and R., 2008], intentando aprovechar las ventajas de ambos.

Visión estéreo

La mayoría de las aproximaciones de odometría visual se han realizado utilizando cámaras estéreo, es decir, dos cámaras que toman imágenes de forma simultánea y que comparten una parte de su campo de visión.

Al utilizar dos cámaras se puede determinar mediante geometría epipolar (Figura 2.5) la distancia a la que se encuentran los objetos en 3D.

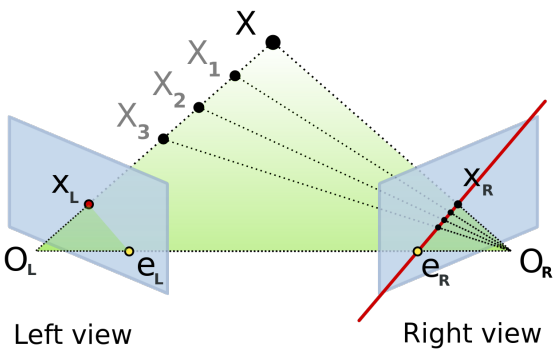


Figura 2.5: Estimación de distancia mediante geometría epipolar.

Así, además de emparejar las imágenes que vienen de una cámara (como en el monocular), se empareja también con la otra cámara para estimar profundidades; para ello, es necesario conocer tanto el desplazamiento que hay entre las dos cámaras como las posiciones (píxeles) en las dos imágenes donde que observa al objeto.

La odometría visual mediante visión estéreo ha sido utilizada tanto para navegación de robots [Matthies and Shafer, 1987] [Olson *et al.*, 2000] como para algoritmos SLAM [Mei *et al.*, 2009] [Lin and Wang, 2010].

Structure From Motion

La comunidad de visión artificial conoce con el nombre de *Structure From Motion* (SFM) al problema de obtener las posiciones 3D y la estructura de una escena a partir de un conjunto de imágenes [Longuet-Higgins, 1981] [Harris and Pike, 1988].

En la Figura 2.6 se muestran los resultados del algoritmo de SFM desarrollado en [McCann, 2015] donde, a partir de imágenes de un mismo lugar obtenidas con distintas cámaras, son capaces de obtener la estructura de la escena.

Figura 2.6: Reconstrucción del Coliseo de Roma mediante SFM.
Fuente: [McCann, 2015].

Este problema es más genérico que el de la odometría visual, puesto que las imágenes obtenidas pueden no ser secuenciales y el resultado final suele ser optimizado mediante algoritmos que requieren un gran tiempo de cómputo, puesto que no necesitan funcionar en tiempo real como sucede con la odometría visual.

El problema de SFM es el equivalente en odometría visual a las técnicas SLAM, con la diferencia de que estas últimas deben ser eficientes para poder funcionar en tiempo real.

2.2.2. MonoSLAM

Los algoritmos conocidos como SLAM (*Simultaneous Localization and Mapping*) han sido utilizados históricamente para generar mapas en entornos desconocidos utilizando robots o vehículos autónomos, localizando a éstos de forma simultánea dentro del mapa generado. En ocasiones, también se utiliza información a priori, como por ejemplo un mapa inicial del entorno en el que se encuentra el robot. Típicamente se han utilizado estos algoritmos utilizando sensores láser [Hahnel *et al.*, 2003], sonares [Newman *et al.*, 2005] o cámaras estéreo [Mei *et al.*, 2009] [Lin and Wang, 2010].

El término MonoSLAM (*Monocular SLAM*) se refiere a la utilización de una sola cámara RGB para la localización y creación de mapas en entornos desconocidos. Fue propuesto y desarrollado por primera vez por Andrew Davison a partir del año 2002 [Davison, 2002] [Davison, 2003].

El algoritmo propuesto por Andrew Davison utiliza un filtro extendido de Kalman (explicado en la sección 2.1.2) para estimar la posición y orientación 3D de la cámara, así como una serie de puntos en el espacio 3D que conforman el mapa. Para determinar la posición inicial de la cámara es necesario dotar al filtro de Kalman de información a priori con la posición en 3D de al menos cuatro puntos; a partir de ese momento, el algoritmo es capaz de situar la cámara en 3D y de generar nuevos puntos del mapa que sirven como apoyo a la propia localización de la cámara.

A continuación se muestran las principales características del EKF utilizado:

Estado y modelo de transición

El vector de estado $\hat{x}$ (ecuación 2.13) está compuesto por el estado propio de la cámara x_v y por el estado f_i de cada uno de los puntos 3D existentes en el mapa. Esto significa que tanto el vector de estado como su covarianza asociada cambian su tamaño de forma dinámica dependiendo del número de puntos del mapa construido hasta ese momento.

El estado de la cámara x_v se compone de la posición 3D de la cámara r^W, su velocidad lineal v^W, la rotación de la cámara q^{WR} (representada con un cuaternión) y la velocidad angular ω^R. Por su parte, cada punto se representa con su posición 3D en el mundo en coordenadas cartesianas.

$$\hat{x} = \begin{bmatrix} x_v \\ f_1 \\ f_2 \\ \vdots \\ f_N \end{bmatrix} \qquad x_v = \begin{bmatrix} r^W \\ v^W \\ q^{WR} \\ \omega^R \end{bmatrix} \tag{2.13}$$

El modelo de transición (ecuación 2.14) actualiza el estado de la cámara aplicando un modelo de velocidad constante a la posición y orientación de la cámara teniendo en cuenta las velocidades lineal y angular de la iteración anterior.

$$x_{v|k} = f(x_{v|k-1}) = \begin{bmatrix} r^W_{|k-1} + v^W_{|k-1}\Delta k \\ v^W_{|k-1} \\ q_g(\omega^R_{|k-1}\Delta k) \times q^{WR}_{|k-1} \\ \omega^R_{|k-1} \end{bmatrix} \tag{2.14}$$

Modelo de Observación

El modelo de observación H_k se compone de las proyecciones h_i de cada uno de los puntos 3D en el plano imagen teniendo en cuenta la posición y orientación de la cámara, como se muestra en la ecuación 2.15.

$$H = \begin{bmatrix} h_i \\ h_2 \\ \vdots \\ h_N \end{bmatrix} \qquad h_i = \begin{bmatrix} u_i \\ v_i \end{bmatrix} \tag{2.15}$$

Eliminación de espurios

Uno de los puntos débiles de los algoritmos de localización basados en filtros de Kalman, como MonoSLAM, es su baja tolerancia a espurios, es decir, a puntos que no estén correctamente emparejados. Por la propia naturaleza del algoritmo, el filtro trata de minimizar el error entre los píxeles emparejados y los predichos, de forma que un emparejamiento erróneo afecta directamente a la estimación de localización del algoritmo. Esto significa que, en el peor de los casos, un solo emparejamiento inexacto puede llevar a una localización errónea del algoritmo, por lo que el porcentaje máximo de espurios que tolera el algoritmo es del 0 %.

Para resolver esto, el algoritmo *1-Point RANSAC* [Civera *et al.*, 2010] permite detectar espurios para no incluirlos en la fase de actualización del filtro de Kalman. Esta técnica utiliza a su vez el algoritmo RANSAC (*Random Sample Consensus*) [Fischler and Bolles, 1981], un método iterativo para estimar parámetros de un modelo con observaciones que contengan falsos positivos.

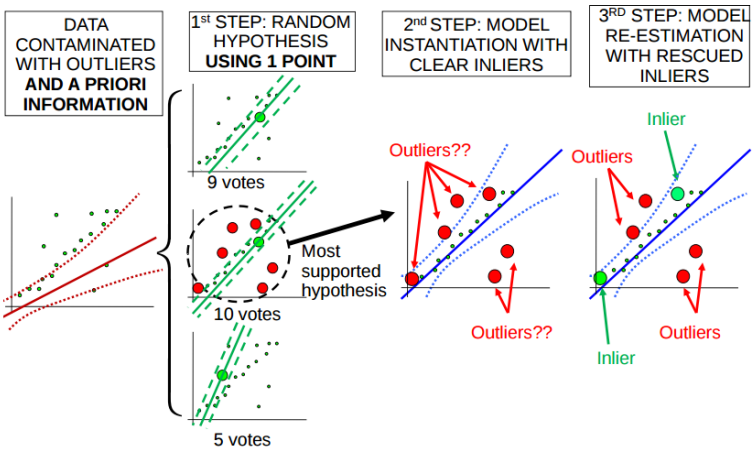


Figura 2.7: Funcionamiento de *1-Point RANSAC* para estimar una recta en 2D.
Fuente: [Civera et al.*, 2010].*

El algoritmo *1-Point RANSAC* (Figura 2.7) divide la fase de actualización del filtro de Kalman en dos partes:

1. Se utiliza RANSAC de forma iterativa para seleccionar en cada iteración 1 único punto y actualizar el estado del filtro (no la covarianza) con esa única observación. Con este nuevo estado, se comprueba cuántos puntos validan la hipótesis (seguidores), es decir, cuántos puntos están lo suficientemente cerca de la posición predicha.

   Una vez ejecutado RANSAC un número suficiente de iteraciones (valor que cambia dinámicamente en función del número de seguidores), se etiquetan como *inliers* todos aquellos puntos que validaron la mejor iteración y como *outliers* el resto de puntos, actualizándose el EKF completo (estado y covarianza) solo con los primeros.

2. Tras actualizar el EKF, se comprueba si alguno de los puntos etiquetados como *outliers* se encuentra ahora cerca de la posición predicha (lo que puede suceder con puntos cercanos a la cámara) y se "rescatan" estos puntos como *inliers*, actualizándose de nuevo el EKF completo.

Este algoritmo permite eliminar espurios mejorando la fiabilidad del algoritmo a expensas de un mayor coste computacional, aunque el algoritmo puede seguir funcionando en tiempo real.

2.2.3. PTAM

PTAM[1] son las siglas de *Parallel Tracking and Mapping*, un algoritmo desarrollado en 2007 por [Klein and Murray, 2007] que tenía como objetivo resolver el mismo problema que MonoSLAM (la localización y elaboración de mapas de forma simultánea), pero haciéndolo desde un punto de vista totalmente diferente.

El mayor problema de los algoritmos basados en MonoSLAM es que su tiempo de ejecución aumenta exponencialmente con el número de puntos del mapa. Esto es debido a que, en cada iteración del algoritmo, se actualiza tanto la posición de cada elemento del mapa (*Mapping*) como la posición actual de la cámara (*Tracking*). Por tanto, aunque el *Tracking* pueda mantenerse en tiempos de ejecución razonables, el *Mapping*, a partir de un cierto número de puntos, impide la ejecución del algoritmo en tiempo real.

PTAM parte de la idea de que solo es necesario funcionar en tiempo real en la parte de *Tracking*, mientras que el *Mapping* no tiene porqué realizarse en cada iteración ni necesita ser tan eficiente. Así, el *Tracking* y el *Mapping* funcionan en dos hilos separados de forma asíncrona.

Tracking

Los pasos que sigue el hilo de *Tracking* son:

1. Preprocesado de la imagen: La primera etapa por la que pasa cada una de las observaciones es el análisis de la imagen; ésta se subdivide en cuatro subimágenes a distinta resolución (pirámide de la imagen) y a cada una de ellas se le aplica un filtro de esquinas FAST [Rosten and Drummond, 2006] para detectar píxeles característicos.

2. Actualización de la posición con modelo de movimiento: Se actualiza la posición de la cámara aplicando un modelo de velocidad constante, de forma similar a lo realizado en la fase de predicción de MonoSLAM. Así, la velocidad v_t y la nueva posición x_{t+1} se calculan con las ecuaciones 2.16 y 2.17:

$$v_t = \frac{x_t - x_{t-1}}{\Delta t} \tag{2.16}$$

$$x_{t+1} = x_t + \Delta t \prime(v_t) \tag{2.17}$$

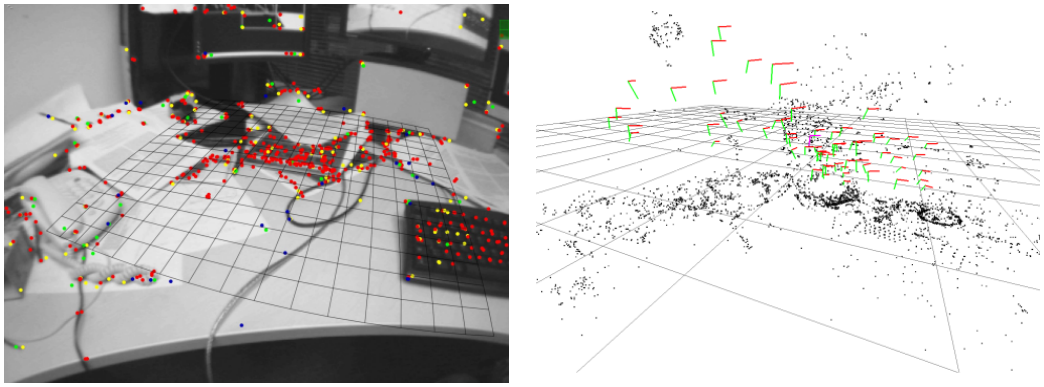
[1] `http://www.robots.ox.ac.uk/~gk/PTAM/`

donde Δt es el tiempo transcurrido entre t y $t-1$, mientras que $\Delta t\prime$ es el tiempo transcurrido entre t y $t+1$.

3. Actualización de grano grueso: Se selecciona un subconjunto de puntos 3D (hasta 60) y se proyectan en la imagen. Estos puntos tratan de emparejarse comparando parches (subimágenes de 8x8 píxeles) en un rango amplio alrededor del píxel 2D en el que han proyectado. Con los puntos encontrados se actualiza la posición de la cámara tratando de minimizar el error de reproyección utilizando el algoritmo iterativo de Gauss-Newton [Kelley, 1999], es decir, se emplea una técnica de optimización.

4. Actualización de grano fino: Se seleccionan hasta mil puntos del mapa y se sigue el mismo procedimiento que en el paso anterior. En este caso, la posición estimada de la cámara será más precisa y el rango de búsqueda de estos puntos podrá ser menor. Con los puntos 3D emparejados se actualiza nuevamente la posición de la cámara minimizando el error de reproyección.

Estos sencillos pasos logran que el *Tracking* pueda ejecutarse en tiempo real, proporcionando además una gran precisión y robustez (Figura 2.8(a)).



(a) (b)

Figura 2.8: Puntos utilizados en el *Tracking* (a) y mapa generado (b) con PTAM.
Fuente: [Klein and Murray, 2007].

Mapping

El hilo de *Mapping* obtiene el mapa inicial del entorno mediante una inicialización en estéreo de dos fotogramas que tengan un desplazamiento suficiente. Para ello, el algoritmo realiza un seguimiento en la imagen de una serie de píxeles característicos, a partir de los cuales calcula el mapa inicial mediante el algoritmo de cinco puntos [Stewenius *et al.*, 2006].

Los puntos iniciales calculados, así como la posición de la cámara, pueden no tener una posición en 3D totalmente real, pero representarán a la realidad en una escala diferente.

Una vez efectuada esta inicialización, el hilo de *Mapping* realiza los siguientes pasos en cada iteración:

1. Añadir nuevos *Keyframes*: No todas las observaciones son añadidas al mapa, sino que solo un subconjunto de éstas cumplirán los requisitos para almacenarse como fotogramas clave (*Keyframes*). Las condiciones a cumplir son: que la distancia a otro *Keyframe* sea suficiente, que hayan pasado suficientes iteraciones desde que se almacenó el último *Keyframe* y que la localización actual sea de buena calidad.

2. Añadir nuevos puntos 3D al mapa: Cada vez que se añade un nuevo *Keyframe* se seleccionan píxeles característicos en la imagen y se buscan en los *Keyframes* anteriores. Los emparejamientos obtenidos se añaden al mapa calculando su posición 3D mediante triangulación. Estos puntos son los que utiliza el hilo de *Tracking* para estimar la posición de la cámara.

3. Optimizar el mapa: Siempre que el hilo de *Mapping* se encuentra desocupado, se refina la posición 3D tanto de los *Keyframes* como de los puntos almacenados. Así, se utiliza el algoritmo de optimización *Bundle Adjustment* [Triggs *et al.*, 1999] [Hartley and Zisserman, 2003] cuyo objetivo es minimizar el error de reproyección entre m posiciones en 3D de la cámara y n puntos 3D (ecuación 2.18):

$$min_{a_j,b_i} \sum_{i=1}^{n} \sum_{j=1}^{m} v_{ij} d(Q(a_j, b_i), x_{ij})^2 \tag{2.18}$$

   donde cada cámara está parametrizada con un vector a_j y cada posición en 3D con un vector b_i. $Q(a_j, b_i)$ es la proyección predicha del punto 3D i en la posición j de la cámara, x_{ij} es la proyección detectada, $d(p_1, p_2)$ es la distancia euclídea entre dos píxeles de la imagen y v_{ij} es una variable binaria que indica si el punto i es visible desde la posición j.

   El algoritmo necesita como entrada tanto los emparejamientos en cada imagen como una posición tentativa de las posiciones de la cámara y de los puntos 3D. El tiempo de cómputo de este algoritmo es muy alto; por ello, a pesar de que el *Mapping* no necesita funcionar en tiempo real, PTAM trata de reducir este tiempo optimizando el mapa únicamente con los *Keyframes* cercanos y ejecutando el algoritmo con todos los *Keyframes* en pocas ocasiones.

4. Mantenimiento del mapa: Se comprueba la coherencia de los datos entre los distintos *Keyframes*, eliminando aquellas asociaciones de puntos mal realizadas y añadiendo nuevos puntos cuando sea posible.

El mapa final obtenido (Figura 2.8(b)) puede contener miles de puntos que se relacionan entre sí en múltiples *Keyframes* y que, por tanto, guardan coherencia espacial entre ellos, evitando así puntos mal situados debido a localizaciones puntuales erróneas.

Comparativa con MonoSLAM

Se han realizado diversas comparativas entre los dos algoritmos, como la realizada por [Strasdat *et al.*, 2012]. En ellas se concluye que, para mapas que tengan muy pocos puntos, es más eficiente utilizar algoritmos de tipo MonoSLAM, mientras que para el resto de casos es mejor utilizar algoritmos de tipo PTAM.

Esto se debe a que PTAM es capaz de manejar miles de puntos simultáneamente y seguir funcionando en tiempo real. Además, esta técnica es más robusta, puesto que puede recuperarse ante observaciones no favorables o secuestros, algo que no es posible con MonoSLAM.

Por su parte, en caso de contar con mapas con muy pocos puntos, MonoSLAM es más eficiente que PTAM a la hora de generar y mantener el mapa, ya que la optimización con *Bundle Adjustment* de PTAM es muy costosa computacionalmente, aunque esto no influye en la eficiencia del *Tracking*.

2.2.4. SVO

Otro de los algoritmos de SLAM que han destacado en los últimos años ha sido SVO[2] (*Fast Semi-Direct Monocular Visual Odometry*) [Forster *et al.*, 2014]. Este algoritmo destaca por su rapidez, pudiendo ser utilizado en tiempo real en ordenadores con poca capacidad de cómputo.

Como ya se explicó en la sección 2.2.1, existen dos técnicas principales para poder calcular la posición de una cámara a partir de sus imágenes de entrada: basadas en características (técnica que emplean MonoSLAM y PTAM) y mediante métodos directos. Una de las características fundamentales de SVO es que utiliza un híbrido de los dos métodos anteriores, lo que reduce notablemente el tiempo de cómputo del algoritmo. Por

[2] `https://github.com/uzh-rpg/rpg_svo`

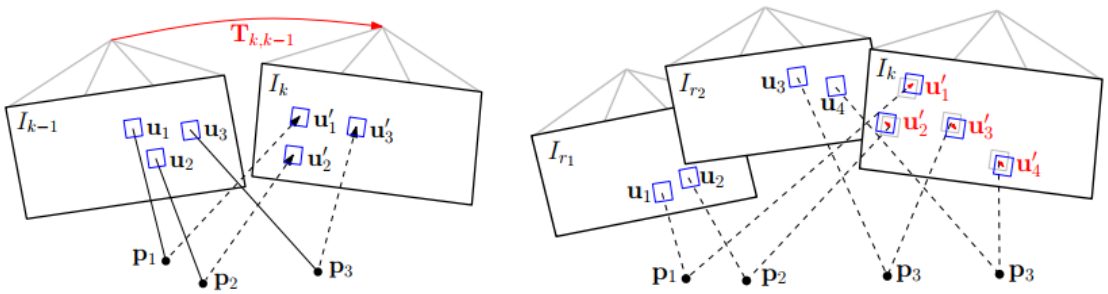
lo demás, el algoritmo tiene una estructura similar a PTAM, separando en dos hilos el cálculo del desplazamiento de la cámara (*Tracking*) y la generación del mapa (*Mapping*).

Tracking

El hilo encargado del seguimiento de la cámara es el que implementa la técnica híbrida que se ha mencionado. El primer paso para calcular la posición de la cámara consiste en minimizar el error fotométrico entre el fotograma actual y el anterior. Si se utilizase toda la imagen para realizar este cálculo, se necesitaría mucho tiempo de cómputo; por ello, el algoritmo SVO solo minimiza el error fotométrico de ciertas partes de la imagen. Estas partes se corresponden con parches de 4x4 alrededor de los píxeles en los que proyectan los puntos 3D visibles en el fotograma anterior (Figura 2.9(a)).

Posteriormente, se ajusta la posición de cada punto 3D por separado teniendo en cuenta el desplazamiento estimado en el paso anterior, minimizando el error fotométrico de cada punto usando parches algo más grandes (8x8 píxeles) que en el alineamiento general (Figura 2.9(b)).

Esta técnica permite obtener el desplazamiento entre imágenes de forma muy eficiente, con tiempos de cómputo de pocos milisegundos que permiten utilizar el algoritmo a más de 60 FPS.



(a) (b)

Figura 2.9: Cálculo de la posición de la cámara en algoritmo SVO: Minimización del error fotométrico (a) y ajuste por separado de cada punto 3D (b).

Fuente: [Forster et al.*, 2014].*

Mapping

Al igual que en PTAM, el algoritmo SVO selecciona algunos de los fotogramas que considera clave (*Keyframes*) y basa la generación del mapa en 3D en esos *Keyframes*.

Inicialmente, los primeros puntos del mapa son calculados mediante homografía al igual que en PTAM. A partir de ahí, los nuevos puntos no se añaden directamente al mapa triangulando entre dos *Keyframes* (como sucedía en PTAM), sino que siguen un proceso más complejo.

Cuando se observa un nuevo punto desde dos *Keyframes*, se calcula la profundidad a la que se encuentra el punto mediante una distribución de probabilidad como la que se ve en la Figura 2.10, donde la incertidumbre de la profundidad es inicialmente muy alta.

A medida que se obtienen nuevas observaciones del punto 3D, se actualiza la distribución de probabilidad hasta que la varianza de la distribución es suficientemente pequeña, momento en el cual se valida el punto en 3D y se añade al mapa.

La ventaja de este método es que los puntos que se añaden al mapa han sido observados desde varios fotogramas y, por tanto, su posición en 3D es más precisa que en PTAM, obteniendo un mapa con menos puntos pero más fiable.

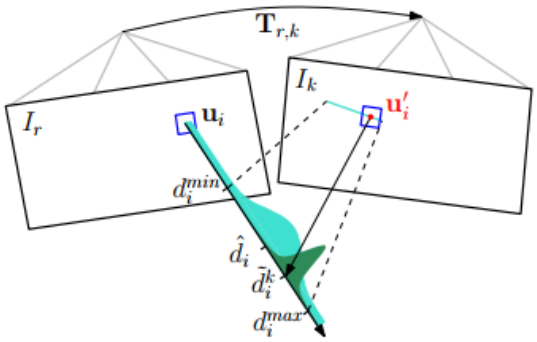


Figura 2.10: Inicialización de puntos 3D con estimación probabilística de profundidad en SVO. *Fuente: [Forster* et al.*, 2014].*

Consideraciones generales

El algoritmo SVO destaca por su rapidez y la fiabilidad del mapa generado, debido a las características que se acaban de describir. En la práctica, el algoritmo está pensado para ser utilizado con cámaras que apuntan hacia el suelo, por lo que puede no funcionar en otras condiciones.

Además, la inicialización de puntos necesita reconocer un mismo punto 3D desde varias observaciones hasta que éste converge, lo que puede llevar a que se pierda el algoritmo en rotaciones bruscas o desplazamientos rápidos.

2.2.5. LSD-SLAM

El algoritmo LSD-SLAM[3] (*Large-Scale Direct Monocular SLAM*) [Engel *et al.*, 2014] es capaz de construir mapas a gran escala utilizando métodos directos (es decir, sin extraer píxeles característicos).

Los algoritmos explicados hasta ahora se centraban especialmente en la localización de la cámara, pero el mapa generado era tan solo un apoyo al *Tracking* en lugar de un fin en si mismo. Por su parte, LSD-SLAM centra su atención en la construcción de un mapa consistente y global del entorno (Figura 2.11).

Además de los hilos de *Tracking* y *Mapping* que tenían el resto de algoritmos, LSD-SLAM también tiene un componente encargado de estimar la profundidad del mapa. A continuación se detalla el funcionamiento de estos tres componentes.

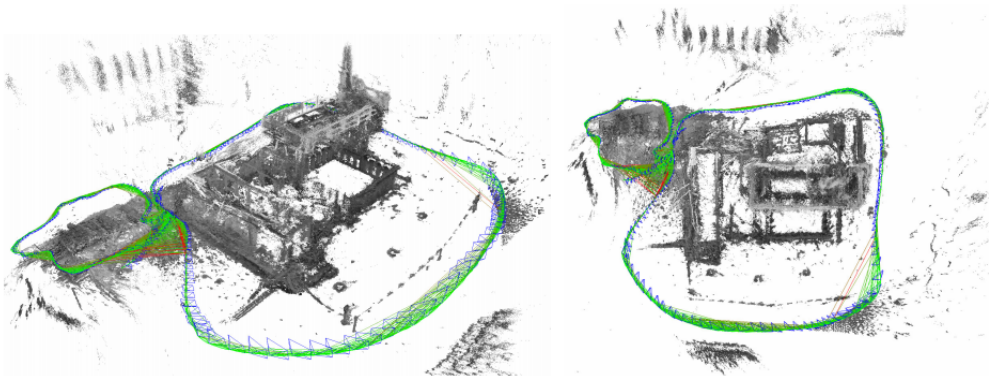


Figura 2.11: Mapa generado con LSD-SLAM.
Fuente: [Engel et al.*, 2014].*

Tracking

Partiendo de un *Keyframe*, el desplazamiento se calcula minimizando el error fotométrico normalizado por la varianza.

El algoritmo ofrece una novedosa técnica para medir la alineación entre imágenes mediante optimización ponderada de Gauss-Newton. En ella, se tiene en cuenta ruido variable en la estimación de la profundidad, que depende del tiempo transcurrido desde que ha sido observado cada punto.

[3] `http://vision.in.tum.de/research/vslam/lsdslam`

Estimador de profundidad

Cada vez que se genera un nuevo *Keyframe*, el mapa de profundidad se inicializa proyectando sobre él los puntos del *Keyframe* anterior y, tras un proceso de escalado y filtrado de *outliers*, sirve como base para el *Tracking* en los siguientes fotogramas.

Los fotogramas que no son utilizados como *Keyframe* sirven para refinar el *Keyframe* actual, comparando zonas de la imagen con suficiente separación estéreo y añadiendo así nuevos píxeles al mapa de profundidad. Esta técnica se explica detalladamente en [Engel *et al.*, 2013].

Mapping

Este componente es el encargado de añadir al mapa los *Keyframes* que ha dejado de utilizar el estimador de profundidad y de optimizar el mapa de forma continua utilizando optimización de grafos de posición [Kümmerle *et al.*, 2011]. Además, LSD-SLAM cuenta también con un mecanismo de cierre de bucle que se ejecuta cada vez que se añade un nuevo *Keyframe*.

Otra de las características de este algoritmo es el método que emplea para crear el mapa inicial. En lugar de utilizar los dos primeros *Keyframes* como sucedía en algoritmos como PTAM, usa una sola imagen para generar un mapa con una incertidumbre en la profundidad suficientemente grande. Este mapa irá convergiendo hacia una configuración de profundidad correcta a medida que se desplace la cámara.

Consideraciones generales

El algoritmo está centrado en la generación de mapas y obtiene, en este aspecto, una fiabilidad y robustez muy altos.

Como desventaja se podría indicar que, a pesar de que puede funcionar en tiempo real en ordenadores convencionales, su tiempo de cómputo es alto y hace poco viable su funcionamiento en otros dispositivos con menos capacidad de cómputo.

2.2.6. ORB-SLAM

El algoritmo ORB-SLAM[4] [Mur-Artal *et al.*, 2015] [Mur-Artal and Tardos, 2016] es un algoritmo de SLAM basado en píxeles característicos que puede funcionar con una

[4] `http://webdiis.unizar.es/~raulmur/orbslam/`

sola cámara, con cámaras estéreo o con cámaras de profundidad RGBD. Su característica principal es que utiliza descriptores ORB [Rublee *et al.*, 2011] para extraer y emparejar píxeles característicos, así como un modelo de bolsa de palabras [Gálvez-López and Tardos, 2012] para detectar cierres de bucle y, en su caso, para relocalizarse.

ORB-SLAM está dividido en tres hilos de ejecución. Además de los hilos de *Mapping* y *Tracking* de los que disponen el resto de algoritmos, añade un hilo más para detectar cierres de bucle. Utilizando estos tres hilos, el algoritmo puede funcionar en tiempo real en ordenadores con alta capacidad de cómputo sin necesidad de intervención de GPU.

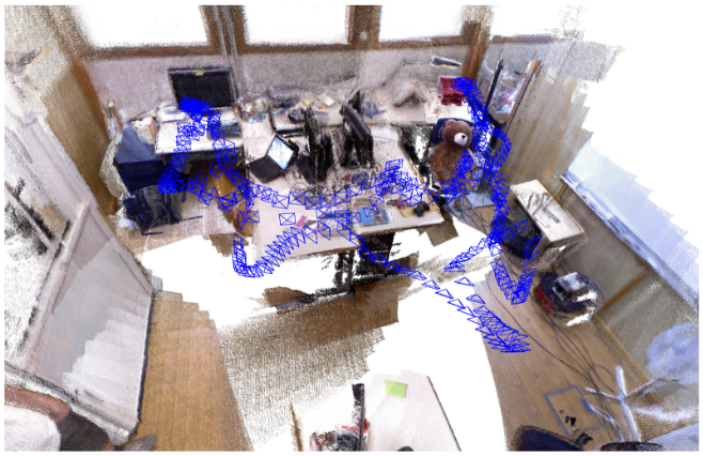


Figura 2.12: Mapa generado con ORB-SLAM utilizando cámaras RGBD.
Fuente: [Mur-Artal and Tardos, 2016].

Mapping

La inicialización del mapa se realiza calculando en paralelo un mapa por homografía y otro mediante una matriz fundamental. Ambos modelos obtienen una puntuación para determinar cuál será el utilizado para inicializar el mapa, de forma que en aquellas escenas que cuenten con un plano principal se utilizará la homografía y en el resto de los casos se utilizará la matriz fundamental. Esto evita que el algoritmo suponga siempre que se encuentra ante escenas que contengan un plano, como hacen el resto de algoritmos.

Una vez obtenido el mapa inicial, el hilo de *Mapping* se encarga de procesar los *Keyframes* creando nuevos puntos 3D y de optimizar el mapa localmente mediante *Bundle Adjustment*. Los nuevos puntos 3D se generan mediante triangulación emparejando descriptores ORB.

Otra característica de este algoritmo es que genera un grafo donde los vértices se corresponden con *Keyframes* y las aristas entre vértices se generan siempre que los

Keyframes tengan varios puntos 3D en común. Este grafo se utiliza entre otras cosas para eliminar *Keyframes* redundantes y evitar así que el número de *Keyframes* crezca innecesariamente.

Tracking

Para realizar el *Tracking*, se aplica un modelo de movimiento y se tratan de emparejar los puntos 3D visibles en el fotograma anterior, calculando la posición actual a partir de los emparejamientos realizados. La diferencia principal de este algoritmo respecto a los anteriores es que los emparejamientos se realizan utilizando descriptores ORB.

En caso de que el robot se pierda, se utiliza un modelo de bolsa de palabras para obtener *Keyframes* candidatos que concuerden con la observación actual. En estos candidatos se buscan correspondencias con los descriptores ORB disponibles y se calcula la posición de la cámara mediante el algoritmo PnP.

Looping

El hilo de *Looping* se encarga de comprobar si se ha producido un cierre de bucle. Para ello. utiliza el grafo de *Keyframes* conectados y el modelo de bolsa de palabras, con el objetivo de buscar *Keyframes* candidatos con apariencia similar a la observación actual.

Si se detecta un cierre de bucle, se eliminan los puntos 3D duplicados y se corrige la posición de los *Keyframes* para alinear los extremos del bucle.

Consideraciones generales

El algoritmo ORB-SLAM es robusto y preciso en multitud de entornos, tanto en escenarios pequeños como en mapas de grandes dimensiones. Gran parte de esa robustez se explica por el uso de descriptores ORB, que, a pesar de requerir mayor tiempo de cómputo que los parches tradicionales, parecen ser más fiables que éstos.

La principal desventaja de este algoritmo es que requiere de procesadores con alta capacidad de cómputo para funcionar en tiempo real.

2.2.7. Otras técnicas de SLAM

Hasta ahora, en este capítulo se han explicado en detalle algunos de los algoritmos que se consideran más relevantes en este campo de investigación y más próximos a los

algoritmos propuestos en esta tesis. Sin embargo, desde el desarrollo del algoritmo de MonoSLAM [Davison, 2002] [Davison, 2003] han sido muchas las propuestas realizadas por otros investigadores para intentar dotar a los algoritmos de SLAM de mayor precisión, robustez y escalabilidad. A continuación se citan otros algoritmos que también han sido importantes en el avance del estado del arte en el campo de la localización visual en entornos desconocidos.

Una de las primeras aproximaciones de las técnicas de SLAM se realizó simplificando el problema suponiendo que solo existía rotación. Esta técnica era más limitada que el problema general pero permitía realizar mosaicos de escenas de forma muy precisa [Montiel and Davison, 2006] [Civera *et al.*, 2009] [Lovegrove and Davison, 2010].

También se han propuesto mejoras a la hora de realizar el emparejamiento de puntos, usando algoritmos como *Joint Compatibility* [Clemente *et al.*, 2007] o técnicas de *Active Matching* [Chli and Davison, 2008] [Handa *et al.*, 2010].

Otro de los problemas clásicos de las técnicas SLAM es cómo diferenciar puntos cercanos (con gran desplazamiento) de otros más alejados que cuentan con más incertidumbre en su profundidad. Así, algunas investigaciones han propuesto dividir los puntos en jerarquías en función de su profundidad [Chli and Davison, 2009] [Strasdat *et al.*, 2011].

La mayoría de las técnicas de SLAM no generan mapas demasiado detallados, por eso se han propuesto varios algoritmos para generar mapas más densos utilizando cada píxel de las observaciones en lugar de píxeles característicos [Zia *et al.*, 2016] [Zienkiewicz *et al.*, 2016].

También se han desarrollado varias técnicas para pasar de la primitiva punto a otras más complejas, como líneas [Smith *et al.*, 2006], planos [Salas-Moreno *et al.*, 2014] [Concha and Civera, 2014] e incluso objetos [Salas-Moreno *et al.*, 2013].

Por último, también cabe destacar los avances que se han realizado utilizando cámaras con profundidad (RGBD) que, aunque necesitan utilizar procesadores de altas prestaciones (como GPU), obtienen una alta precisión y mapas muy detallados [Newcombe *et al.*, 2011] [Labbe and Michaud, 2014].

Capítulo 3

Escenarios de experimentación

Antes de presentar los algoritmos de localización propuestos, tanto para entornos con mapas como sin mapas, se describen en este capítulo las herramientas y los escenarios concretos en los que se han validado experimentalmente. Primero sobre robots simulados, segundo sobre robots reales y tercero sobre bases de datos públicas estándar.

En cuanto a los escenarios, se han empleado el entorno de la RoboCup, entornos de interiores (oficinas) y entornos de robótica aérea y cámaras libres. Cada uno de ellos con sus requisitos particulares en cuanto a autolocalización. Estos escenarios permiten probar los algoritmos desarrollados, caracterizar su comportamiento, compararlos con otros algoritmos y validarlos.

3.1. Simuladores

El *software* desarrollado en esta tesis ha sido probado tanto en simuladores como en robots reales. Los simuladores tienen la ventaja de que permiten probar distintos algoritmos en las mismas condiciones experimentales; también permiten verificar su funcionamiento en escenarios sencillos antes de llevarlos a los robots y escenarios reales, los cuales son siempre más ruidosos y problemáticos. Además, permiten graduar la dificultad para la localización de los diferentes escenarios en los que se prueban.

En concreto, para la realización de esta tesis se han utilizado los simuladores Webots[1] y Gazebo[2].

[1] `http://www.cyberbotics.com/`

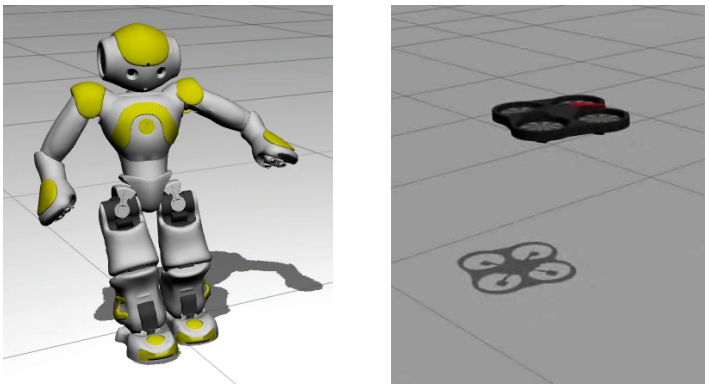
[2] `http://gazebosim.org/`

3.1.1. Simulador Gazebo

El simulador Gazebo comenzó a desarrollarse en el 2002. En los últimos años se ha convertido en uno de los simuladores más utilizados, especialmente a partir del 2011 cuando Willow Garage empezó a financiar el proyecto y a raíz de su elección como simulador de referencia en el *DARPA Robotics Challenge*.

Este *software* permite simular múltiples robots en 3D, pudiendo utilizar sus sensores y actuadores, y puede ser utilizado con diferentes motores de físicas que le dotan de realismo. Además, el simulador también permite crear objetos 3D con los que interactuar.

Gazebo proporciona varios modelos de robots y una serie de mundos en los que utilizar estos robots. Entre los modelos de robots actualmente mantenidos por la comunidad de desarrollo se encuentran algunos de los robots reales utilizados en esta tesis, como el robot *Nao* y el *AR.Drone* (Figura 3.1).



(a) (b)

Figura 3.1: Robots *Nao* y *AR.Drone* en el simulador Gazebo.

El simulador también permite crear mundos, robots y objetos personalizados en los que configurar las propiedades que sean necesarias. Gazebo permite simular tanto cámaras ideales como con distorsión; en el presente trabajo se han empleado cámaras ideales siguiendo el modelo Pin-Hole.

La versión de Gazebo utilizada en esta tesis ha sido Gazebo 7, lanzada en el año 2016, aunque en la actualidad ya está disponible la versión 8. Este simulador ha sido empleado para realizar experimentos tanto con el robot *Nao* como con el *AR.Drone*.

3.1.2. Simulador Webots

El simulador Webots es un entorno de desarrollo creado por la compañía Cyberbotics para modelar, programar y simular robots móviles. Al igual que Gazebo, Webots permite configurar una serie de robots y objetos que interactúan entre sí en un entorno compartido. Además, se pueden establecer las propiedades de cada objeto, como forma, color, textura, masa, etc. También hay disponibles un amplio conjunto de actuadores y sensores para cada robot. En concreto, las cámaras pueden ser simuladas utilizando ciertos tipos de distorsión (como el ojo de pez) o usando parámetros de cámaras ideales. Estas últimas han sido las elegidas para el desarrollo de los experimentos realizados en esta tesis.

Dentro de la plataforma de desarrollo, existen varios modelos de robots y objetos preestablecidos, entre los que se encuentran el robot *Nao* y los objetos necesarios para simular el campo de la RoboCup (Figura 3.2).

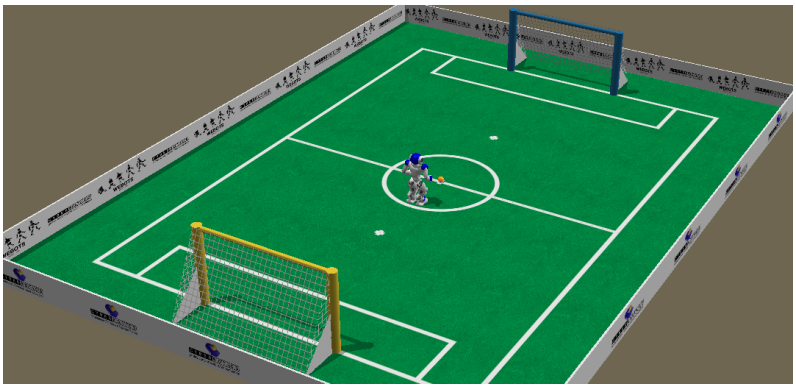


Figura 3.2: Campo de la RoboCup en el simulador Webots.

Para los experimentos de esta tesis se ha utilizado la versión 6.3 de Webots, estando actualmente disponible la versión 8.5 de este *software*.

3.2. Robots reales

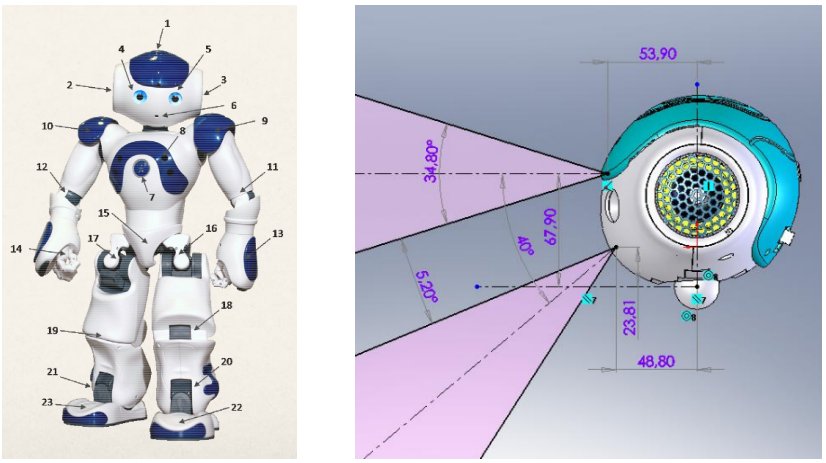
En este trabajo se han utilizado robots reales para validar experimentalmente los algoritmos en entornos realistas y en condiciones desfavorables. A continuación se describen brevemente los robot físicos utilizados como plataforma para implementar los algoritmos desarrollados.

3.2.1. Robot *Nao*

El robot *Nao* es el robot utilizado en la liga de la plataforma estándar de la RoboCup desde el año 2008, cuando sustituyó al robot *Aibo* de Sony. Se trata de un robot humanoide desarrollado por Aldebaran Robotics desde el año 2004 y que actualmente se encuentra en su versión *Nao Evolution V5*, lanzada en el año 2014.

Algunas de las características principales de este robot humanoide son:

- 21 grados de libertad, permitiendo una gran amplitud de movimientos (Figura 3.3(a)).
- Detectores de presión en pies y manos, conocidos como FSR.
- Dos cámaras con distintos campos de visión (Figura 3.3(b)).
- Cuatro sensores de ultrasonido.
- Inerciales, de gran utilidad en caso de caídas.
- Ordenador incorporado con comunicación vía Ethernet o Wi-Fi, permitiendo así las comunicaciones inalámbricas.
- LED en diversas partes del cuerpo, que resultan de gran ayuda para conocer el estado del robot.



(a) (b)

Figura 3.3: Grados de libertad (a) y cámaras (b) del robot *Nao*.

La versión del robot *Nao* utilizado en los algoritmos de esta tesis ha sido la versión *Nao RoboCup Edition V3+*, lanzada a comienzos de 2010. Esta versión cuenta con un

procesador modelo *x86 AMD GEODE 500MHz* y usa como sistema operativo Linux. Para el movimiento autónomo dispone de una batería que le otorga una autonomía de 45 minutos en espera o de 15 minutos caminando.

La cámara del robot real cumple una función fundamental en esta tesis porque es la fuente de información de los algoritmos de localización; esta cámara tiene una baja resolución pero no tiene apenas distorsión. Sus parámetros intrínsecos han sido obtenidos utilizando las herramientas de calibración de la librería *OpenCV* [Bradski and Kaehler, 2008].

El robot *Nao* proporciona odometría pero es poco fiable, ya que está calculada mediante *software* en función de los ángulos de las distintas articulaciones del robot, siendo común que estos ángulos no sean precisos.

El robot dispone también de un entorno de programación (*Naoqi*) que permite interactuar de forma sencilla con la arquitectura *software*. Este entorno ha sido desarrollado por Aldebaran Robotics y permite programar aplicaciones en C++ y Python.

Se ha utilizado este robot tanto en el simulador Webots como en condiciones reales. El robot cuenta con 3 grados de libertad, aunque es posible llegar a tener de forma limitada 6 grados de libertad de movimiento utilizando sus múltiples articulaciones.

3.2.2. Robots Aéreos

Se han utilizado varios robots aéreos (cuadricópteros) durante la realización de esta tesis: el *Parrot AR.Drone*, el *3DR Solo Drone* y el *HoverWasp* (Figura 3.4).

(a) (b) (c)

Figura 3.4: Robot Aéreos *AR.Drone 2.0* (a), *3DR Solo Drone* (b) y *HoverWasp* (c).

El *Parrot AR.Drone* es un robot cuadricóptero de bajo coste que comenzó a comercializarse en el año 2010. Este robot puede ser teleoperado de forma sencilla desde un dispositivo móvil y cuenta con cuatro grados de libertad, tres de desplazamiento y uno de rotación. Viene equipado además con dos cámaras con distintos campos de visión (una frontal y otra ventral) y acelerómetros, así como con una batería que le otorga una autonomía de aproximadamente 15 minutos.

Por su parte, el robot *3DR Solo Drone* es un cuadricóptero que está especialmente diseñado para llevar a bordo cámaras externas de alta calidad, como las cámaras *GoPro* [3]. Dispone de un motor de 880 KV, lo que le proporciona una potencia muy alta que le permite transportar cámaras sin dificultad, con un tiempo de vuelo de hasta 25 minutos.

Por último, el drone *HoverWasp* ha sido diseñado y desarrollado en la Universidad Rey Juan Carlos [Cano, 2016]. Este cuadricóptero puede ser teleoperado por radio control, Wi-Fi o Bluetooth. Además cuenta con un motor de alta potencia y permite llevar a bordo cámaras externas, con una autonomía de hasta 25 minutos.

El robot *AR.Drone* ha sido utilizado principalmente en el simulador Gazebo, puesto que las imágenes obtenidas de las cámaras del robot real tenían poca calidad. En su lugar, para grabar los vídeos con drones de esta tesis se han utilizado los otros dos robots, el *3DR Solo Drone* y el *HoverWasp*, ya que ambos permitían utilizar cámaras externas de alta definición. Los parámetros intrínsecos de estas cámaras han sido obtenidos utilizando *OpenCV*.

3.3. Bases de datos internacionales

Para evaluar correctamente los experimentos es necesario que éstos sean repetibles en las mismas condiciones; asimismo, es necesario contar con algún sistema objetivo de medición del error en las estimaciones. Para ello, se ha recurrido al uso de numerosos vídeos para servir como base experimental del *software* presentado. Algunos de estos vídeos han sido grabados con el fin de ser utilizados en esta tesis, mientras que otros han sido obtenidos de bases de datos internacionales.

El uso de bases de datos internacionales permite medir con mayor certeza la precisión de los algoritmos y compararlos con otras técnicas del estado del arte. En concreto, se han utilizado los bases de datos descritas a continuación.

[3] `https://es.gopro.com/`

TUM RGB-D

Banco de pruebas desarrollado por la Universidad Técnica de Múnich (TUM)[4] [Sturm *et al.*, 2012]. Está compuesto por 89 secuencias divididas en diferentes categorías y obtenidas en dos escenarios diferentes: un escenario de oficinas y una sala industrial. Las escenas cuentan con verdad absoluta gracias a un sistema de captura de movimiento.

Este banco de pruebas proporciona diversos escenarios y tipos de desplazamiento: escenas con desplazamientos lentos para depuración, trayectorias amplias con cierre de bucle, desplazamientos bruscos, secuestros o escenas con elementos dinámicos. No todas las escenas están pensadas para algoritmos de SLAM, ya que algunas incluyen solo rotaciones.

A continuación se detallan las escenas concretas más ilustrativas que se han utilizado en los experimentos de esta tesis:

- *Freiburg1 Floor* (F1F): En esta secuencia (Figura 3.5(a)) se realiza un desplazamiento de la cámara mirando hacia el suelo. La habitación en la que se graba es de reducidas dimensiones pero se realizan giros bruscos en algunas partes del vídeo que pondrán a prueba a los algoritmos de SLAM. La secuencia no está completa y presenta saltos, por ello en los experimentos solo se utilizarán los fotogramas anteriores al primer salto.

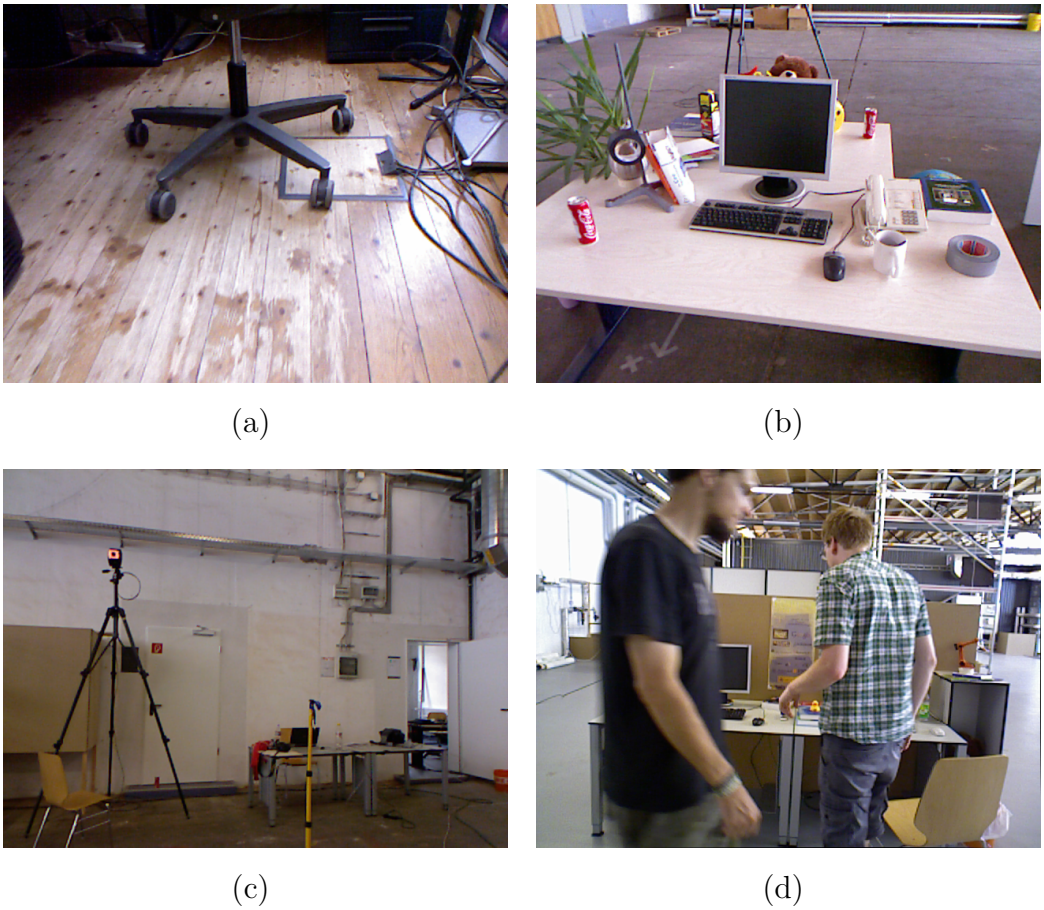
- *Freiburg2 Desk* (F2D): En este vídeo (Figura 3.5(b)) se sigue una trayectoria circular alrededor de una mesa de oficina mirando en todo momento hacia la mesa, terminando el vídeo cerca de donde comenzó. Esta escena servirá para comprobar el funcionamiento de los algoritmos cuando observan un mismo punto 3D desde distintas orientaciones y su precisión a la hora de cerrar el bucle.

- *Freiburg2 360 Kidnap* (F2K): Se realiza una trayectoria circular a lo largo de la sala con una cámara a bordo de un robot (Figura 3.5(c)). En un determinado momento se tapa la cámara y se secuestra al robot, lo que servirá para comprobar la capacidad de relocalización de los algoritmos.

- *Freiburg3 Walking Static* (F3W): La cámara se mantiene fija frente a un escritorio con dos personas que se desplazan por la escena (Figura 3.5(d)). Esta secuencia servirá para comprobar el funcionamiento del algoritmo cuando existen elementos dinámicos.

[4] `http://vision.in.tum.de/data/datasets/rgbd-dataset`

Además, este banco de pruebas incluye herramientas para evaluar la precisión de los algoritmos de SLAM que serán utilizadas en algunos de los experimentos de esta tesis, tanto para evaluar esta base de datos como para el resto de bases de datos internacionales.



(a) (b)

(c) (d)

Figura 3.5: Captura de las secuencias F1F (a), F2D (b), F2K (c) y F3W (d) del banco de pruebas TUM RGB-D.

ICL-NUIM

Este banco de pruebas ha sido desarrollado por el Imperial College de Londres[5] [Handa *et al.*, 2014] con el objetivo de servir como evaluación a algoritmos de odometría visual y SLAM. Se compone de dos escenas diferentes generadas de forma sintética: una sala de estar y una habitación de oficina, con cuatro secuencias de vídeo cada una, en las que se dispone de verdad absoluta.

Los dos escenarios se caracterizan por contar con elementos con poca textura, como las pareces y el suelo, lo que dificulta la detección y el emparejamiento de puntos 3D. Las

[5] `https://www.doc.ic.ac.uk/~ahanda/VaFRIC/iclnuim.html`

escenas han sido grabadas con movimientos lentos, pero en ocasiones se realizan giros sin apenas desplazamiento que complican la localización.

En los experimentos de esta tesis se ha utilizado una de las secuencias del escenario de oficinas, llamada *Office Room 3* (OR3), de la que se muestra un ejemplo en la Figura 3.6. En esta secuencia se realiza un giro completo alrededor de una habitación y se termina cerca de la posición inicial, por lo que es fácil comprobar si la trayectoria seguida ha sido correcta. Además, este escenario cuenta con pocos elementos con textura, de modo que el número puntos 3D obtenidos será reducido.

Figura 3.6: Captura de la secuencia OR3 del banco de pruebas ICL-NUIM.

EUROC MAV

Banco de pruebas desarrollado por el Instituto Tecnológico de Zúrich (ETH)[6] [Burri *et al.*, 2016] utilizando un micro vehículo aéreo (MAV). Está compuesto por dos bases de datos distintas: la primera realizada en una nave industrial de amplias dimensiones (5 escenas) y la segunda grabada en una habitación pensada para realizar reconstrucción 3D (6 escenas). Las escenas proporcionadas se han etiquetado como de nivel fácil, medio o difícil en función de la velocidad de los desplazamientos y los cambios de luminosidad que se producen.

El vehículo aéreo utilizado dispone de dos cámaras estéreo y un sensor inercial (IMU). La verdad absoluta ha sido estimada utilizando dos sistemas con precisión milimétrica, un láser *Leica MS50* y un sistema de captura de movimiento *Vicon*.

La primera de las bases de datos se ha grabado en un entorno industrial de alrededor de 400 m^2 y cuenta con tres trayectorias distintas divididas en 5 escenas. Las trayectorias

[6] `http://projects.asl.ethz.ch/datasets/doku.php?id=kmavvisualinertialdatasets`

se caracterizan por su larga duración y por la gran cantidad de elementos disponibles para calcular la localización del robot, por lo que el mapa generado tendrá grandes dimensiones y un gran número de puntos.

La segunda base de datos está rodada en una habitación que dispone de un sistema *Vicon* de posicionamiento y cuenta con dos escenarios distintos con 3 escenas cada uno. La habitación tiene unas dimensiones más reducidas que la base de datos anterior y menos elementos en los que apoyarse para la localización.

En los experimentos se utilizan dos vídeos correspondientes a la primera trayectoria realizada en el entorno industrial, llamados *Machine Hall 01* (MH01) (Figura 3.7) y *Machine Hall 02* (MH02). Estos vídeos proporcionan un banco de pruebas adecuado para evaluar trayectorias en escenarios de grandes dimensiones.

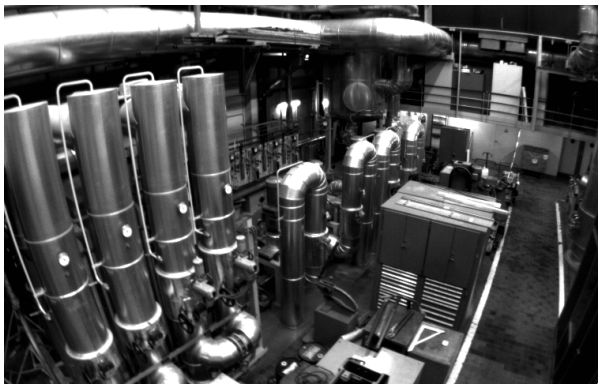


Figura 3.7: Captura de la secuencia MH01 del banco de pruebas EUROC MAV.

Zurich Urban MAV

Banco de pruebas publicado por la Universidad de Zúrich (UZH)[7] [Majdik *et al.*, 2017] utilizando un micro vehículo aéreo (MAV). Está compuesto por una sola secuencia de exteriores en la que el drone recorre 2 km por varias calles de la ciudad de Zúrich (Suiza), pensada para evaluar algoritmos de odometría visual, SLAM y reconstrucción 3D en entornos urbanos.

El vehículo aéreo cuenta con una cámara de alta resolución, un sensor GPS y un sensor inercial. Para obtener la verdad absoluta el banco de pruebas parte de los datos de posición del GPS y posteriormente refina las posiciones mediante *Bundle Adjustment*. Las transiciones realizadas en esta secuencia son lentas y cuentan con un gran número de puntos en los que apoyarse, por lo que el mapa generado tendrá grandes dimensiones.

[7] `http://rpg.ifi.uzh.ch/zurichmavdataset.html`

Para los experimentos se han utilizado solo los primeros 5 minutos de esta secuencia (etiquetada como ZU), reescalando las imágenes para tuviesen un tamaño similar al del resto de bases de datos (Figura 3.8). Esta secuencia servirá para comprobar el funcionamiento de los algoritmos en exteriores.

Figura 3.8: Captura de la secuencia ZU del banco de pruebas Zurich Urban MAV.

3.4. Entorno de la RoboCup

La RoboCup [8] es un proyecto internacional que tiene como objetivo promover la inteligencia artificial (IA), la robótica y otros campos relacionados. Se trata de promocionar la investigación sobre robots e IA proporcionando problemas estándar donde un amplio abanico de tecnologías pueden ser integradas y examinadas.

Para solucionar estos problemas deben incorporarse diversas tecnologías a las que debe hacer frente cada robot: razonamiento en tiempo real, utilización de estrategias, trabajo colaborativo entre robots, uso de múltiples sensores o, el campo en el que está centrado nuestro proyecto, la autolocalización visual.

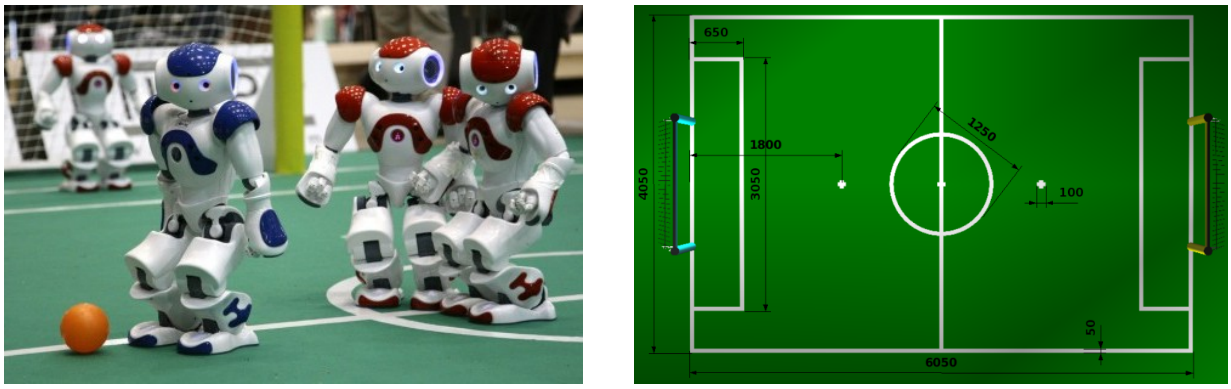
Las actividades realizadas por la RoboCup no se limitan a las competiciones entre robots, sino que también se incluyen conferencias técnicas, programas educativos, infraestructuras de desarrollo, etc; la RoboCup se divide en cinco grandes competiciones:

- RoboCupSoccer: Proyecto principal donde se realizan competiciones de fútbol entre robots autónomos.
- RoboCupRescue: Se pone a prueba a los robots en tareas de búsqueda y salvamento en terrenos desfavorables.

[8] `http://www.robocup.org/`

- RoboCupJunior: Trata de acercar las metas de la RoboCup a estudiantes de educación primaria y secundaria.
- RoboCup@Home: Centrada en el uso de robots autónomos para realizar tareas del hogar y la vida diaria.
- RoboCup@Work: Cuya meta es la utilización de robots en escenarios de trabajo.

Esta tesis se encuadra dentro de la competición RoboCupSoccer. Esta competición se subdivide en categorías según el tamaño y el tipo de robot a utilizar; en nuestro caso, la liga en la que se compite es de la plataforma estándar (SPL) (Figura 3.9). En ella, todos los participantes utilizan el mismo robot (el robot *Nao*) y solo se centran en el desarrollo del *software* de éste.



(a) (b)

Figura 3.9: Robots *Nao* durante la competición de la RoboCup SPL 2009 (a). Dimensiones del campo establecidas en el reglamento de la RoboCup (b).

Los robots utilizados deben ser totalmente autónomos y utilizan las cámaras como sensor principal. Uno de los factores determinantes para el robot jugador es saber qué hacer cuando percibe la pelota, puesto que la actuación del robot deberá ser distinta cuando se encuentre en su propia área (intentará despejar) que cuando esté en el área contraria (intentará tirar a portería). Por ello, uno de los problemas más importantes que se plantean es la localización dentro del campo, que debe resolverse utilizando visión artificial.

El campo de la plataforma estándar de la RoboCup es similar al que se puede ver en la Figura 3.9(b), compuesto por dos porterías y una serie de líneas blancas que delimitan el campo. Las porterías utilizadas en los experimentos de esta tesis son una de color azul y otra de color amarillo, mientras que la pelota es de color naranja. En las últimas ediciones de la RoboCup, las normas de los elementos del campo han cambiado para evolucionar

hacia condiciones idénticas al fútbol humano, de forma que tanto las porterías como la pelota son también de color blanco.

Hay que tener en cuenta que los partidos se componen de 6 jugadores (3 de cada equipo), por lo que nuestro robot puede ver también a los otros jugadores. Fuera de los límites del campo puede haber cualquier tipo de objetos, como espectadores, carteles publicitarios, etc, que deberán considerarse a la hora de realizar la detección de los distintos elementos del campo.

La RoboCup es un escenario dinámico en el que los secuestros son frecuentes, ya sea por caídas de los robots o por sanciones arbitrales, y donde las oclusiones son también habituales al existir otros robots en el campo, lo que implica que los algoritmos de autolocalización deben ser robustos. Por la naturaleza de este escenario, una precisión de alrededor de 30 cm es suficiente para que el comportamiento del jugador sea el correcto. Asimismo, los algoritmos no pueden ser pesados computacionalmente, puesto que deben ser ejecutados a bordo del robot *Nao*, que cuenta con recursos muy limitados.

3.4.1. Software para la RoboCup

Los algoritmos de autolocalización desarrollados en esta tesis se han integrado en una plataforma *software* concreta para ser utilizados en la RoboCup. Esta plataforma *software* recibe el nombre de BICA (*Behavior-based Iterative Component Architecture*) [Agüero *et al.*, 2010] y su necesidad se debe a las limitaciones de cómputo que tiene el robot *Nao*, requiriendo el desarrollo de un *software* específico que funcionase en tiempo real.

La arquitectura BICA está basada en componentes que se ejecutan de forma iterativa en un único hilo y que se comunican entre sí mediante llamadas a métodos. Cada componente se ejecuta de forma iterativa a una frecuencia determinada; esta frecuencia vendrá definida por una frecuencia máxima que se establece en cada componente y dependerá del tiempo de cómputo que consuman los otros componentes.

En el caso de que un componente dependa de otros, se distinguirá entre componentes perceptivos (de los que depende el propio componente) y de actuación (los que modula el componente). Independientemente de la frecuencia que tenga el componente, antes de su puesta en marcha se llamará a sus componentes perceptivos y después se llamará a sus componentes de actuación. Esto hace que, a partir de un componente raíz, se propaguen las llamadas a todos los componentes utilizados durante la ejecución del programa.

BICA se encarga de la comunicación con los robots a través del *software* integrado en cada uno de ellos (*Naoqi*), de forma que los componentes desarrollados puedan hacer

llamadas a *Naoqi* para utilizar los sensores y actuadores del robot. Además, la arquitectura BICA permite comunicarse tanto con los robots reales como con el simulador Webots.

Los algoritmos de localización basados en mapa desarrollados en esta tesis se integraron como componentes de BICA (componente *Localization* señalado en verde en la Figura 3.10), utilizando diversos componentes perceptivos para obtener las imágenes de las cámaras del robot y para controlar el desplazamiento de éste.

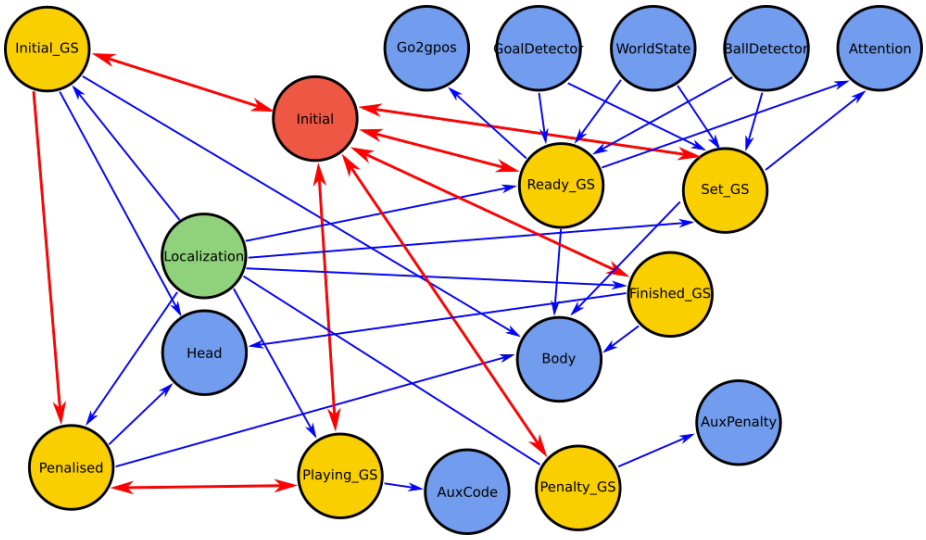


Figura 3.10: Componentes del *software* del jugador URJC de la RoboCup.

3.5. Entorno de interiores

Para los experimentos realizados en entornos con mapa conocido era preciso buscar un escenario de interiores que cumpliese las siguientes características:

- Que se pudiese obtener un mapa característico del mundo en coordenadas absolutas.
- Que fuese suficientemente grande como para poder utilizar grandes desplazamientos con robots reales, más grande que el escenario de la RoboCup.
- Que tuviese elementos identificables naturales de ese escenario con los que poder localizarse sin necesidad de poner elementos ex profeso.
- Que varias zonas del mapa fuesen similares (simetrías), para comprobar cómo se comportaban los algoritmos bajo múltiples hipótesis.

Dentro de los distintos escenarios de interiores, como oficinas, universidades, hogares, etc, se optó por el uso de un entorno de oficinas compuesto por un pasillo con múltiples puertas similares (Figura 3.11), forzando así a que los algoritmos tuviesen que lidiar con múltiples hipótesis debido a las simetrías.

(a) (b)

Figura 3.11: Entorno de oficinas (a) e imagen obtenida del entorno con el robot *Nao* (b).

Otra de las características de este escenario es que cuenta con poca textura en el suelo y las paredes, por lo que no existen demasiados elementos en los que apoyarse para calcular la localización.

Además de los componentes estáticos del escenario, también es posible encontrarse con otros elementos dinámicos que dificultarán la localización, como personas u otros robots, que tendrán que ser tenidos en cuenta por los algoritmos.

La precisión requerida en el entorno de oficinas para validar los algoritmos será de alrededor de 50 cm.

3.5.1. Software JdeRobot

En este escenario de interiores, se ha utilizado como *middleware* principal de desarrollo la plataforma JdeRobot [9]. Esta plataforma ha sido desarrollada por el grupo de robótica de la URJC y conforma un *middleware* de desarrollo de *software* para aplicaciones con robots y visión artificial. Se ha elegido JdeRobot por la facilidad con la que es posible comunicarse con distintos robots, accediendo a sus sensores y actuadores de forma sencilla. Además, también permite utilizar otras aplicaciones ya creadas para esta plataforma de desarrollo, pudiendo reutilizar partes del código de otros proyectos que ya han demostrado su aplicabilidad y robustez.

[9] `http://jderobot.org/`

JdeRobot está desarrollado en los lenguajes de programación C, C++ y Python y proporciona un entorno de programación distribuido basado en componentes, donde la aplicación se organiza en distintos componentes asíncronos. Los componentes se comunican entre ellos a través del *middleware* de comunicación ICE y pueden incluso estar ejecutándose en distintos ordenadores.

La plataforma facilita el acceso al *hardware* de los robots. Así, obtener las mediciones de los sensores u ordenar comandos a los motores es tan simple como llamar a funciones locales. JdeRobot se encarga de transformar esas llamadas a funciones en llamadas a través de la red a los componentes que están conectados a los dispositivos (ya sean reales o simulados), que pueden estar ejecutándose tanto local como remotamente.

Las funciones que se utilizan forman parte de una API que proporciona una capa de abstracción *hardware*, simplificando así el acceso a sensores y actuadores y enmascarando las llamadas a funciones reales necesarias para interactuar con cada dispositivo.

La plataforma proporciona varios componentes que sirven como *drivers* para soportar distintos sensores físicos, actuadores y simuladores. Entre los robots soportados actualmente se encuentran:

- Sensores RGBD: *Kinect* y *Kinect2* de Microsoft, *Xtion* de Asus.
- Robots con ruedas: *TurtleBot* de Yujin Robot y *Pioneer* de MobileRobotics Inc.
- Cuadricópteros: *AR.Drone* de Parrot y *Solo Drone* de 3DRobotics.
- Simulador Gazebo.
- Cámaras Firewire, USB y cámaras IP.
- Robot humanoide *Nao* de Aldebaran Robotics.

Además, JdeRobot cuenta con herramientas de programación, como teleoperadores, visores de distintos robots, o calibradores de cámaras, así como con librerías de programación para procesamiento de visión artificial, geometría proyectiva, etc.

En particular, para desarrollar esta tesis se han utilizado algunos de componentes y librerías que proporciona la plataforma (Figura 3.12), que se detallan a continuación:

- Cameraserver: Componente que permite tanto el acceso a cámaras conectadas localmente mediante USB como la reproducción de vídeos grabados. A través de este componentes se obtienen imágenes en el formato deseado que se actualizan en tiempo real.

- Naoserver: Este componente permite establecer comunicación con el robot *Nao* de Aldebaran Robotics para obtener tanto las imágenes como el valor de sus sensores.
- Gazeboserver: Con este componente es posible conectar las aplicaciones al simulador Gazebo, lo que permite comprobar el funcionamiento de los algoritmos en robots simulados antes de hacerlo en robots reales.
- Ardroneserver: Componente para conectarse al cuadricóptero *AR.Drone* de Parrot.
- MAVLinkServer: Componente que permite la conexión al cuadricóptero *Solo Drone* de 3DRobotics.
- Pose3D: Interfaz que representa la posición y orientación del robot en cada momento; se utiliza tanto para obtener la verdad absoluta del robot en los simuladores como para ofrecer el resultado calculado por la localización.
- Geometry: Librería de geometría proyectiva.
- Visionlib: Librería de visión artificial.

Figura 3.12: Conexiones del componente de localización en JdeRobot.

La versión utilizada de JdeRobot ha sido la 5.4, lanzada en julio de 2016, que cuenta con 14 componentes, 18 librerías y 28 herramientas.

3.6. Entorno de robótica aérea y cámaras libres

Se entiende por entorno de robótica aérea, aquellos vídeos que han sido grabados con motivo de tesis con los robots aéreos presentados anteriormente en este capítulo. En este caso, los escenarios no dispondrán de un mapa conocido y las observaciones se obtendrán mirando hacia el suelo. Normalmente, se trabajará en escenarios de exteriores, con la posibilidad de que las observaciones contengan vibraciones y movimientos bruscos.

En concreto, se han grabado cuatro secuencias en el campus de Fuenlabrada de la Universidad Rey Juan Carlos[10] (Figura 3.13), utilizando el drone *HoverWasp* con una cámara *GoPro* a bordo. Tres de estas secuencias (URJC1, URJC2 y URJC3) han consistido en realizar trayectorias largas a lo largo del campus, mientras que, en la última (URJC4), se ha situado al drone en una posición casi fija mientras determinados elementos del entorno (personas y coches) se desplazaban por debajo. Esta última secuencia servirá para comprobar el funcionamiento de los algoritmo en entornos con elementos dinámicos.

(a) (b)

Figura 3.13: Captura de las secuencias URJC1 (a) y URJC4 (d) realizadas con drone *HoverWasp*.

Para calcular la verdad absoluta en estas secuencias se ha utilizado un sensor GPS, teniendo siempre presente el error de este tipo de sensores. Así, la precisión de los algoritmos requerida en estos escenarios será de pocos centímetros y la posición calculada será en 3 dimensiones, ya que los drones cuentan con seis grados de libertad.

También se han realizado experimentos utilizando cámaras libres, es decir, cámaras externas o cámaras USB cuya trayectoria se realizaba manualmente. En este caso, también existen 6 grados de libertad, al igual que sucede cuando se utilizan robots aéreos.

La plataforma *software* empleada en este escenario ha sido nuevamente JdeRobot, ofreciendo la posición calculada por los algoritmos a través del interfaz *Pose3D*.

[10] `http://jderobot.org/Eperdices_PhD`

Capítulo 4

Localización visual con mapas conocidos

Este capítulo se centra en la localización visual en entornos conocidos. Se entiende por entorno conocido aquel del que se dispone de un mapa con elementos situados en tres dimensiones, que servirán al robot como base para su localización. Los elementos que aparecen en el mapa serán estáticos, mientras que el resto del entorno puede ser dinámico; esto implica la posible existencia de otros objetos que también tendrán que ser tenidos en cuenta por los algoritmos de localización para obtener una localización robusta.

En concreto, a lo largo de este capítulo se explicarán dos técnicas para la localización visual con mapas conocidos: un algoritmo de Monte Carlo y un algoritmo evolutivo. El algoritmo de Monte Carlo se ha implementado a partir de las técnicas que se describieron en el capítulo 2 del estado del arte y ha sido elegido como referencia para las comparaciones; por su parte, el algoritmo evolutivo ha sido específicamente diseñado e implementado para resolver el problema de localización tratado en este capítulo.

Estas dos técnicas serán utilizadas tanto en el escenario de la RoboCup como en entornos de oficinas. En ambos escenarios la robustez es un factor importante, mientras que la localización no necesita ser muy alta. La localización calculada se simplifica para estos escenarios, puesto que será en 2 dimensiones con tres grados de libertad (X, Y, θ).

También cabe destacar que los dos escenarios suponen retos distintos. El entorno de la RoboCup destaca por tener muchas oclusiones y dinamismo de objetos; por otra parte, en el escenario de oficinas los elementos son más estáticos, pero hay menos texturas y cuenta con un elevado número de simetrías, lo que complica la autolocalización.

Otro aspecto fundamental a tener en cuenta es que una única observación no suele ser

concluyente a la hora de localizar al robot, puesto que dicha observación puede no contener suficiente información como para localizarse o se pueden producir errores al analizar las imágenes, debido a oclusiones, falsos positivos, etc. Por ello, para lograr una localización fiable y robusta, es necesario contar con algoritmos que sean capaces de realizar una acumulación temporal de muchas observaciones.

Existen diversas técnicas para localizar a un robot en un mundo conocido, que pueden ser divididas en dos bloques principales:

- Técnicas constructivas: Calculan la solución final de forma analítica a partir de la información recibida. Algunas de estas técnicas han sido descritas en el capítulo 2, como el algoritmo *Perspective-n-Point*.
- Técnicas abductivas: Calculan la posición en la que se encuentra el robot a partir de suposiciones que se validan a posteriori desde la información espacial y descartando las incompatibles.

Los algoritmos explicados en esta sección tienen un enfoque abductivo, ya que manejan poblaciones de soluciones candidatas y comparan la imagen real obtenida con las imágenes teóricas que se obtendrían si el robot estuviese en una posición determinada, con el objetivo de validar, descartar o evaluar cada posición candidata.

4.1. Algoritmo de Monte Carlo

El algoritmo de Monte Carlo [Fox *et al.*, 1999b] y algunas de sus variantes [Thrun *et al.*, 2001] [Fox, 2001] fueron explicadas en el estado del arte del capítulo 2. Se trata de técnicas probabilísticas que manejan funciones de probabilidad para representar la información de posición y que evolucionan a medida que el robot recoge nuevas observaciones.

Bajo ciertas asunciones razonables, Monte Carlo permite plantear el problema probabilístico de manera incremental, representando las funciones de probabilidad de modo muestreado como un conjunto de partículas, en lugar de hacerlo de manera analítica o paramétrica, como se haría con otras técnicas probabilísticas clásicas (como los filtros de Kalman). Este muestreo es lo que permite al algoritmo acumular la información parcial de las observaciones en tiempo real y de forma robusta.

En este algoritmo se utilizan un número P de elementos, llamados partículas, que representan distintas posiciones en las que puede encontrarse el robot en un momento

determinado. A través de estas partículas, el algoritmo de Monte Carlo trata de encontrar la posición real del robot de una forma eficiente.

Existen diversas variaciones del algoritmo de localización de Monte Carlo (MC clásico); para este trabajo se han implementado tanto el algoritmo de MC clásico como la versión de MC aumentado (*augmented MC*).

Las partículas del algoritmo guardan información sobre la posición del robot (X, Y, θ) y la probabilidad obtenida con una función de coste, que tendrá que ser adaptada en cada entorno. Por ello, esta función de coste será detallada para cada entorno en los experimentos del capítulo 5.

4.1.1. Iteraciones Regulares

En la primera iteración del algoritmo, las partículas deben ser distribuidas por el mundo para buscar localizaciones del robot plausibles. Para ello, puede realizarse una distribución aleatoria por el mundo o se puede utilizar la información de entrada de las cámaras para situar las partículas manualmente en posiciones en las que se sepa que su función de coste será alta (es decir, su localización será buena). Para que el algoritmo sea genérico, se ha utilizado la primera opción: repartir las partículas por el mundo de forma aleatoria.

Una vez realizado el caso particular de la primera iteración, el algoritmo de Monte Carlo realiza los siguientes pasos de forma iterativa:

1. **Modelo de movimiento**: En el caso de que el robot disponga de odometría y se conozca el movimiento que ha realizado desde la última iteración, se puede actualizar la posición de todas las partículas teniendo en cuenta este movimiento incremental.

2. **Modelo de observación**: El siguiente paso consiste en calcular la probabilidad de cada una de las partículas en el mundo a través de una función de coste. Esta función de coste depende del entorno y será explicada en el capítulo 5 de experimentos.

3. **Remuestreo**: La función de probabilidad almacenada de modo muestreado en las partículas ya no es representativa de la función de probabilidad real. Por ello, es necesario realizar un remuestreo donde se determine la posición de cada partícula en la siguiente iteración en función de su probabilidad.

4. **Selección de la posición actual**: Por último, se calcula la localización del robot a partir de las partículas generadas.

Con estos pasos, se seleccionan para la siguiente iteración las partículas que tengan una mejor localización según la función de coste y se generan otras partículas próximas a ellas. Así, con el paso de las iteraciones se creará una nube de partículas que se situará en la posición más compatible con los datos de entrada obtenidos, que se corresponderá idealmente con la localización real del robot.

Además, en caso de que todas las partículas estén por debajo de un umbral mínimo durante más de I iteraciones, el algoritmo se reinicia volviendo a distribuir todas las partículas de forma aleatoria, ya que se considera que el robot no está bien localizado o que se ha producido un secuestro. Los valores de estos parámetros también dependerán del entorno y tendrán que ser ajustados experimentalmente.

4.1.2. Modelo de movimiento

Algunos robots proporcionan información de odometría, que puede ser utilizada para aumentar la precisión de la localización. Los algoritmos desarrollados permiten de forma opcional utilizar esta odometría para aplicar un modelo de movimiento al inicio de cada iteración. La aplicación de este modelo de movimiento no es indispensable para los algoritmo de localización, ya que funcionan sin necesidad de odometría, no obstante, funcionan mejor si se aprovecha esta información sensorial.

Así, cuando el robot se haya desplazado desde la última iteración, este desplazamiento se puede reflejar actualizando la posición (X, Y, θ) de todas las partículas.

Obtener el movimiento realizado por un robot es una tarea complicada y los errores producidos en la medición pueden ser muy altos, especialmente cuando se utilizan robots bípedos. A pesar de ello, se ha desarrollado un modelo de movimiento que se ajustase lo más posible a la realidad, teniendo siempre en cuenta que pueden existir errores odométricos.

El objetivo de este modelo de movimiento es calcular el movimiento incremental en (X, Y, θ) del robot desde la última vez que se midió. Para ello, se han utilizado funciones propias de los robots que permiten conocer el movimiento relativo realizado por el robot en (X, Y, θ) desde su arranque.

Los pasos realizados para calcular el movimiento respecto a la última iteración son:

1. Se guarda en cada iteración el valor actual devuelto por la función de odometría del robot, que se comparará con el de la siguiente iteración.

2. Se comprueba si el robot se ha movido, comparando los nuevos valores obtenidos con los guardados previamente. En el caso de que no haya un desplazamiento significativo se considera que el robot está parado, finalizando entonces el modelo de movimiento sin necesidad de actualizar la posición de las partículas.

3. Se calcula la diferencia (en posición y orientación) entre los valores actuales (P_n) y los anteriores (P_l) para calcular la diferencia P_d, mediante la ecuación 4.1:

$$P_d = \epsilon \cdot (P_n - P_l) \tag{4.1}$$

donde ϵ es un factor de corrección al error sistemático que se produce en la odometría del robot. Este factor variará dependiendo del robot y tendrá que ser calculado experimentalmente.

4. El siguiente paso consiste en calcular el movimiento relativo entre dos iteraciones (P_r). Así, el giro del robot (P_{r_θ}) es directamente la diferencia P_{d_θ} calculada en el paso anterior; mientras que, para calcular la posición relativa del robot (P_{r_x}, P_{r_y}), se aplica una matriz de rotación traslación (RT) teniendo en cuenta la orientación del robot en la última iteración (P_{l_θ}), tal y como se muestra en la ecuación 4.2:

$$\begin{pmatrix} P_{r_x} \\ P_{r_y} \end{pmatrix} = \begin{pmatrix} cos(-P_{l_\theta}) & -sen(-P_{l_\theta}) \\ sen(-P_{l_\theta}) & cos(-P_{l_\theta}) \end{pmatrix} \begin{pmatrix} P_{d_x} \\ P_{d_y} \end{pmatrix} \tag{4.2}$$

Una vez obtenido el movimiento relativo P_r realizado desde la última iteración, se actualiza la posición de cada partícula (P_i) para obtener su nueva posición (P_f). La nueva orientación de cada partícula se calcula con la ecuación 4.3:

$$P_{f_\theta} = P_{r_\theta} + P_{i_\theta} \tag{4.3}$$

Mientras que para obtener los valores finales (P_{f_x}, P_{f_y}) se aplica la matriz RT de la ecuación 4.4:

$$\begin{pmatrix} P_{f_x} \\ P_{f_y} \\ 1 \end{pmatrix} = \begin{pmatrix} cos(P_{i_\theta}) & -sen(P_{i_\theta}) & P_{i_x} \\ sen(P_{i_\theta}) & cos(P_{i_\theta}) & P_{i_y} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_{r_x} \\ P_{r_y} \\ 1 \end{pmatrix} \tag{4.4}$$

4.1.3. Remuestreo

Tras calcular la probabilidad de cada partícula mediante la función de coste, el algoritmo de Monte Carlo establece la posición de las partículas de la siguiente iteración mediante el remuestreo. El remuestreo realizado es diferente para los algoritmos de MC clásico y de MC aumentado, por lo que se van a explicar por separado:

Remuestreo en MC clásico

En MC clásico se seleccionan las partículas de la siguiente iteración utilizando dos mecanismos, el elitismo y la ruleta:

- **Elitismo**: Consiste en seleccionar las P_N mejores partículas en función de la probabilidad obtenida con la función de coste. Este mecanismo tiene como fin preservar las mejores partículas sin modificaciones en la siguiente iteración, para asegurar que la localización calculada no empeore al utilizar la aleatoriedad de la ruleta.

- **Ruleta**: Se genera una ruleta del algoritmo de Monte Carlo clásico como la que se explicó en la sección 2.1.4 del capítulo del estado del arte. Para ello, se selecciona una partícula aleatoriamente en la ruleta y se le aplica un pequeño ruido Gaussiano, es decir, se modifica levemente su posición y orientación, evitando así que existan dos partículas iguales.

  Esta selección por ruleta se realiza para P_R partículas. Al ser mayor la proporción de la ruleta para las mejores partículas, en la siguiente iteración se obtendrá previsiblemente una nueva generación de partículas que convergerá hacia la posición en la que se encuentra el robot.

  En la literatura existen pruebas de que Monte Carlo converge cuando se utiliza esta técnica para representar las funciones de probabilidad.

El número de partículas seleccionadas por cada uno de los métodos anteriores tendrá que ser ajustado experimentalmente, pero deberá cumplir que la suma de P_N y P_R sea igual a las P partículas que utiliza el algoritmo.

Remuestreo en MC aumentado

El remuestreo en MC aumentado parte de la misma base que en MC clásico, utilizándose los mecanismos de elitismo y ruleta de la sección anterior. Además, añade un nuevo

mecanismo de *selección aleatoria* de P_A partículas como el que se explicó en la sección 2.1.4.

Este nuevo mecanismo tiene como objetivo buscar nuevas posiciones donde pueda encontrarse el robot, evitando que el algoritmo converja en un máximo local. Así, cumple una función equivalente a la de los individuos exploradores que se verá en el algoritmo evolutivo.

El número de partículas que se generan con este mecanismo varía en función de cómo de buena sea la localización estimada, lo que depende de la probabilidad media obtenida con la función de coste. Así, cuanto peor sea la localización, más partículas aleatorias existirán. Esto evita en la mayoría de las ocasiones que se tenga que reiniciar el algoritmo cuando el robot se esté perdiendo, algo que sí sucede al utilizar el algoritmo de MC clásico.

Al añadirse un nuevo mecanismo de selección de partículas, ahora deberá cumplirse la restricción de que la suma de partículas obtenidas P_N, P_R y P_A tendrá que ser igual a P partículas.

4.1.4. Estimación final de la posición actual

El último paso del algoritmo en cada iteración consiste en seleccionar la localización final del robot. De nuevo, el método a seguir es distinto entre MC clásico y MC aumentado:

Selección de la posición actual en MC clásico

En Monte Carlo clásico se supone que, tras un número de iteraciones suficientemente grande, el algoritmo acabará formando una nube de partículas alrededor de la posición real del robot. Por ello, la localización final en (X, Y, θ) en cada iteración se calcula simplemente como la media ponderada de las partículas elitistas de esa iteración.

Al utilizar una media se evita que la localización cambie bruscamente de una iteración a otra, dando como resultado un desplazamiento suavizado que se corresponde mejor con el desplazamiento real del robot.

En caso de que se utilizasen todas las partículas disponibles para calcular la posición media, las partículas con peor probabilidad (alejadas previsiblemente de la solución real) también serían tenidas en cuenta para calcular la media y la localización final estimada sería peor. Al utilizar solo las partículas elitistas se evita esta situación y se asegura una mayor precisión en la localización estimada.

Selección de la posición actual en MC aumentado

Uno de los problemas del cálculo de la localización en Monte Carlo clásico es que, si en lugar de converger las partículas hacia una sola posición se crean dos o más nubes de partículas (alrededor de máximos locales), el cálculo de la posición media dará como resultado una localización totalmente errónea.

Para evitar esta situación en MC aumentado, antes de calcular la localización final del robot, se dividen todos los elitistas en grupos (*clusters*). Cada grupo se compone de un conjunto de partículas elitistas que tienen una posición y orientación similares, lo que hará que generalmente lleven a una misma solución.

Una vez que se han dividido las partículas elitistas en G grupos, se calcula la localización media para cada grupo de la misma forma que en MC clásico, obteniendo la localización media de los elitistas que forman el grupo.

La localización final (X, Y, θ) calculada será la posición del grupo que cuente con un mayor número de partículas elitistas. El grupo se selecciona con este criterio para dar estabilidad a la localización, puesto que un mayor número de partículas elitistas se suele corresponder con una nube de partículas mayor.

En caso de que se encuentre una localización donde los elitistas tengan una probabilidad mayor pero su número sea menor, el mecanismo de ruleta del algoritmo de Monte Carlo haría que, con el paso de las iteraciones, estos elitistas fuesen aumentando su número progresivamente, hasta que previsiblemente superasen a los elitistas del grupo anterior. Así, se produce cierto retardo a la hora de cambiar de grupo, pero se evitan saltos continuos en la localización.

4.1.5. Fiabilidad de la localización

El algoritmo desarrollado, además de calcular una localización del robot en cada iteración a partir de las observaciones recibidas, también ofrece un valor de la fiabilidad de esta localización.

Este valor de fiabilidad servirá a otros componentes que utilicen la localización como fuente de información para tomar decisiones. Por ejemplo, en el entorno de la RoboCup los robots utilizaban la localización para decidir hacia dónde tenían que disparar la pelota; si la localización era poco fiable, era preferible que el robot no hiciese nada a que disparase hacia su propia portería, siendo la fiabilidad un factor clave en su comportamiento.

La fiabilidad de la localización depende directamente de la calidad de las observaciones que reciba el robot. Una imagen tendrá una alta calidad cuando los elementos que se pueden extraer de ésta son relevantes para calcular la localización. Por ejemplo, en la RoboCup una observación con alta calidad es aquella en la que se aprecian las porterías, mientras que en el entorno de oficinas la calidad depende de si se ven una o varias puertas. El cálculo de la calidad de la imagen en cada escenario se desarrollará en el capítulo 5.

Inicialmente, la fiabilidad F_t toma un valor de 0 (nada fiable) y modifica su valor en función de tres elementos: la calidad de la imagen, la fiabilidad calculada en la iteración anterior F_{t-1} y la probabilidad media de las partículas elitistas P_p. Se han configurado experimentalmente valores máximos (P_{max}) y mínimos (P_{min}) para etiquetar las probabilidades como altas o bajas, actualizando la fiabilidad en cada caso mediante las siguientes ecuaciones:

1. Imagen de alta calidad y $P_p > P_{max}$:

$$F_t = F_{t-1} + \Delta_{max}\frac{P_p - P_{max}}{1 - P_{max}} \tag{4.5}$$

2. Imagen de baja calidad y $P_p > P_{max}$:

$$F_t = F_{t-1} + \Delta_{min}\frac{P_p - P_{max}}{1 - P_{max}} \tag{4.6}$$

3. Imagen de alta calidad y $P_p < P_{min}$:

$$F_t = F_{t-1} - \Delta_{max}\frac{P_{min} - P_p}{P_{min}} \tag{4.7}$$

4. Imagen de baja calidad y $P_p < P_{min}$:

$$F_t = F_{t-1} - \Delta_{min}\frac{P_{min} - P_p}{P_{min}} \tag{4.8}$$

siendo Δ_{max} y Δ_{min} las variaciones máximas que pueden producirse en la fiabilidad en una iteración, asegurando así que la fiabilidad converja lentamente. Por defecto, los valores de P_{max} y P_{min} se han fijado en 0.7 y 0.5, mientras que Δ_{max} y Δ_{min} toman los valores 0.15 y 0.03. En caso de que no se cumpla ninguna de estas condiciones la fiabilidad no se modifica. Al establecer el valor de la fiabilidad de forma progresiva se evita que una observación que cuente con oclusiones o falsos positivos afecte demasiado a la fiabilidad total.

En la Figura 4.1 se puede ver un ejemplo de cómo cambia la fiabilidad a medida que pasan las iteraciones.

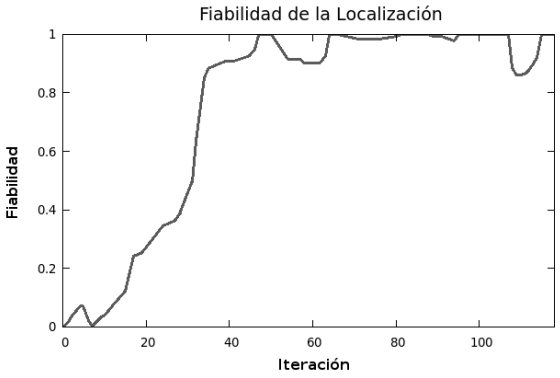


Figura 4.1: Evolución de la fiabilidad de la localización con Monte Carlo.

4.2. Algoritmo de localización evolutivo

Los algoritmos evolutivos son métodos de optimización metaheurística que se inspiran en la evolución biológica para buscar la solución a un problema. En este tipo de algoritmos, las soluciones candidatas son conocidas como individuos, los cuales pertenecen a una población que evoluciona a lo largo del tiempo mediante operadores genéticos.

En cada iteración, los individuos son evaluados a través de una función de calidad que permite conocer su "salud" (*fitness*), es decir, conocer en qué medida se acercan a la solución óptima.

Este algoritmo se puede ver como un algoritmo de búsqueda en el espacio de posibles soluciones de autolocalización donde la información para descartar candidatos se obtiene de las imágenes del robot y del mapa. Es similar a los filtros de Monte Carlo, pero sin las garantías de convergencia que ofrecen estos métodos probabilísticos muestreados. En lugar de probabilidad, se maneja salud y, en lugar de partículas, individuos, pero permite una mayor flexibilidad en la dinámica de las soluciones tentativas y no converge prematuramente a una solución candidata en caso de existir múltiples hipótesis (simetrías).

En la implementación realizada, el algoritmo evolutivo desarrollado cuenta con cuatro elementos básicos:

- **Individuos**: Los individuos representan soluciones candidatas en las que se encuentra el robot, para ello almacenan la posición 2D del robot (X, Y, θ), la probabilidad

obtenida de la función de calidad y un parámetro para indicar si se encuentra entre los mejores de la última población (elitismo).

- **Razas**: Cada raza es un conjunto de individuos (población) que se encuentran cerca de una posición, permitiendo realizar búsquedas de grano fino alrededor de ésta. El algoritmo cuenta con R razas que compiten entre sí para buscar la mejor estimación de localización del robot.

  Cada raza tiene varios parámetros que la definen: su posición principal (X, Y, θ), su "salud", el número de veces que ha sido la mejor estimación y las iteraciones restantes sin que pueda eliminarse.

- **Exploradores**: Individuos independientes encargados de encontrar posiciones en el espacio de soluciones donde la función de calidad sea alta, con el objetivo de generar nuevas razas en esas posiciones.

- **Explotadores**: Cada uno de los individuos que forman parte de una raza, encargados de analizar en profundidad una posición y sus alrededores en busca de la mejor solución local.

Las razas están diseñadas para mantener, cuanto tiempo sea necesario, múltiples hipótesis, siempre que todas ellas sean compatibles con las observaciones visuales recogidas por el robot hasta ese momento. Cada raza se mantiene mientas conserve compatibilidad con las imágenes; de este modo, el algoritmo puede manejar bien la simetría del escenario manteniendo varias razas simultáneamente.

En caso de que las imágenes de entrada del algoritmo coincidan con varias posiciones del mundo (simetrías), el algoritmo crea nuevas razas en cada una de esas posiciones, ya que el robot podría estar situado en cualquiera de ellas. Tras varias iteraciones, previsiblemente, los datos de entrada proporcionarán información que permitirá descartar a la mayoría de las razas y el algoritmo estimará la posición real del robot a través de la raza con mejor "salud".

4.2.1. Iteraciones regulares

En cada iteración del algoritmo se realizan una serie de pasos para calcular la localización del robot en cada momento (Figura 4.2).

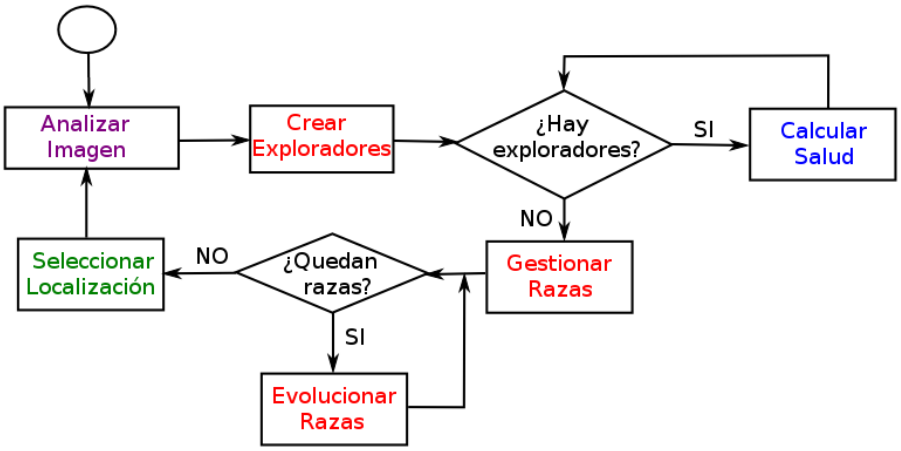


Figura 4.2: Esquema general del algoritmo evolutivo.

A continuación se describen los seis pasos más importantes realizados en cada iteración:

1. **Análisis de imágenes**: Los datos de entrada obtenidos por los algoritmos son las imágenes recibidas a través de la cámara y, de forma opcional, la odometría de los robots. Se querrán relacionar los datos del mapa con los datos visuales en cada imagen de entrada del algoritmo; por ejemplo, las líneas del campo y las porterías en el escenario de la RoboCup, o las puertas y los rodapiés en el escenario de oficinas.

2. **Cálculo de salud**: La salud se asocia a cómo de compatible es la imagen actual recibida de la cámara con la que debería verse en ella si el robot estuviese en una posición candidata. Este cálculo de salud depende del escenario y del análisis de las imágenes que se realice.

3. **Creación de exploradores**: Se crean nuevos individuos (exploradores) encargados de buscar posiciones en del mundo en las que crear nuevas razas.

4. **Gestión de razas**: Se crean, fusionan o eliminan razas en función del estado de éstas y de la salud de los exploradores.

5. **Evolución de razas**: Las razas evolucionan en cada iteración utilizando operadores genéticos. Además, si el robot cuenta con odometría y se ha movido desde la última iteración, se actualiza la posición de todas las razas teniendo en cuenta este desplazamiento.

6. **Selección de posición**: Después de calcular la salud de cada raza, de entre ellas se selecciona aquella que determina la posición en la que se encuentra el robot.

En las siguientes secciones se detalla cada uno de los pasos del algoritmo, exceptuando el análisis de imágenes y el cálculo de salud que se explicarán en el capítulo 5.

4.2.2. Creación de exploradores

Los exploradores son individuos que no pertenecen a ninguna raza y que buscan posiciones en el mundo en las que podrían existir razas con salud alta.

Existen dos formas de distribuir los exploradores en el entorno conocido: de forma aleatoria o de forma abductiva cuando el mundo sea lo suficientemente pequeño o con características especiales. Para conseguir un algoritmo que sea totalmente genérico y sirva para todo tipo de entornos conocidos, se han utilizado siempre mecanismos aleatorios para generar estos exploradores.

La creación de exploradores es muy costosa computacionalmente, por lo que no se ejecuta en todas las iteraciones. Las condiciones para que se realice esta funcionalidad son:

- La observación actual de la cámara debe contener suficiente información como para poder inferir una posición plausible para el robot. Esta condición se ha impuesto debido a que, si no se dispone de suficiente información, las posiciones encontradas no suelen llevar hacia la localización correcta y solo aumentan el tiempo de cómputo del algoritmo innecesariamente.

  Para saber si en una observación hay suficiente información es necesario realizar un análisis del mundo, seleccionando determinados elementos que lleven a un número limitado de posibles soluciones. Esta información, por tanto, dependerá del escenario; por ejemplo, en el campo de la RoboCup será necesario ver alguna de las porterías.

- Debe transcurrir un periodo de tiempo suficiente desde la última vez que se realizó la creación de exploradores. De esta forma se evita directamente que la búsqueda de nuevas posiciones aumente demasiado el tiempo de cómputo del algoritmo.

Además, hay que tener en cuenta que existen dos tipos de búsquedas en función del estado actual del algoritmo. Cuando el algoritmo acaba de iniciarse o el robot se ha "perdido" (todas sus razas cuentan con una salud muy baja), se permite una búsqueda en profundidad, que puede hacer que el algoritmo deje de funcionar en tiempo real, puesto que la alternativa sería no estar localizado. Por el contrario, cuando ya se dispone de una localización fiable, la creación de exploradores se realiza de forma que no afecte demasiado al tiempo de cómputo del algoritmo y éste siga funcionando en tiempo real.

La única diferencia entre estos dos tipos de búsquedas es el número de exploradores aleatorios que se distribuyen en el entorno, siendo en la búsqueda en profundidad de al menos un orden de magnitud mayor que en la búsqueda estándar.

Cuando el algoritmo realiza la creación de exploradores, se generan E individuos exploradores, variando el número E en función del tipo de búsqueda que se realice. El algoritmo calcula la salud de cada uno de esos exploradores a partir de la observación actual y los ordena de mayor a menor en función de su salud. Los mejores exploradores se convierten en candidatos para crear una nueva raza, como se explica en la siguiente sección.

4.2.3. Gestión de razas

A largo plazo, el algoritmo dispone de varias razas que buscan a la vez una localización para el robot, por lo que se ha desarrollado un mecanismo capaz de gestionar la creación, eliminación o fusión de razas. Antes de detallar estos mecanismos, se explican a continuación dos de los parámetros de los que dispone cada raza, el número de victorias y la vida:

- *Número de victorias*: Se considera que una raza ha "vencido" en una iteración cuando su salud ha sido mayor que la del resto de razas. En un principio todas las razas comienzan con 0 victorias, variando este número en función de los requisitos que se verán en la sección 4.2.5. Este parámetro sirve finalmente para seleccionar la raza que establece la localización del robot en cada iteración.
- *Vida*: Este parámetro determina cuántas iteraciones deben transcurrir hasta que una raza pueda ser borrada. La "vida" se ha creado con dos objetivos:

  1. Dar prioridad a las razas ya creadas frente a nuevos candidatos, evitando así la generación continua de razas que se crean y eliminan cíclicamente en cada iteración. Así, al crearse una raza se fija una vida mínima de 3 iteraciones, en las cuales no puede ser borrada ni sustituida (aunque sí fusionada).
  2. Evitar que una raza pueda ser desechada por una mala observación puntual que reduzca su salud. Esto se consigue haciendo que la vida mínima de una raza sea de 9 iteraciones cuando ha sido la ganadora en una iteración. Este mecanismo proporciona robustez ante oclusiones y ante errores espurios en percepción.

  De esta forma se pretende que las razas tengan cierta estabilidad y se evita que se generen y borren razas de forma continua.

Creación de nuevas razas

El algoritmo evolutivo cuenta con un número máximo de razas R que evita que el tiempo de ejecución aumente exponencialmente por tener demasiadas razas. Cuando se obtienen exploradores candidatos para convertirse en razas, hay que decidir si se crean nuevas razas o si se sustituyen las ya existentes.

Primero, se comprueba si se ha alcanzado el número máximo de razas R; si no se ha alcanzado, se compara la posición (X, Y, θ) de cada candidato con la posición de las razas ya existentes, para comprobar si tiene alguna raza cercana. De ser así, se supone que la raza cercana ya representa a ese candidato y se aumenta su vida.

En caso de que ya existan R razas, se reemplaza la raza con menor salud que tenga sus parámetros de vida y número de victorias a 0. Si ninguna raza cumple estas condiciones los candidatos se ignoran.

Fusión de razas

Dos razas se fusionan si evolucionan hacia posiciones (X, Y, θ) similares, puesto que se considera que representan la misma solución final. Esta fusión consiste en eliminar la raza que tenga un menor número de victorias, o en caso de empate, se elimina la que tenga una salud más baja.

Eliminación de razas

En el caso de que una raza tenga tanto su número de victorias como su vida a 0, se elimina siempre que su salud se encuentre por debajo de un umbral. El valor de este umbral es configurable, pero deberá ser lo suficientemente bajo como para considerar que la localización de la raza es incorrecta.

Este mecanismo permite descartar soluciones candidatas de autolocalización que no son compatibles con el flujo de observaciones visuales durante un tiempo suficiente.

4.2.4. Evolución de razas

Cuando se crea una raza desde un explorador, todos los individuos de la raza (explotadores) son generados de forma aleatoria mediante una función de ruido térmico aplicada sobre el explorador que creó la raza, es decir, se modifica su posición y orientación

dentro de unos márgenes. A partir de ese momento, en el resto de iteraciones, los explotadores de la raza evolucionan utilizando tres operadores genéticos:

- **Elitismo**: Consiste en seleccionar los N mejores explotadores de cada raza, ordenados de mayor a menor de acuerdo a su salud. Los explotadores elitistas se guardan sin modificación para la siguiente población.
- **Cruce**: Se seleccionan al azar dos explotadores de la raza y se calcula la media entre ellos en las coordenadas (X, Y, θ).
- **Ruido térmico**: Se modifica levemente la posición y orientación de un explotador seleccionado aleatoriamente. Equivale a realizar búsquedas locales alrededor de la posición de una raza.

Una vez evolucionados los explotadores, se calcula el valor final de la raza como la media ponderada en (X, Y, θ) de todos sus elitistas, evitando de esta forma que se produzcan cambios bruscos en la localización.

Asimismo, se puede utilizar opcionalmente la información de odometría de los robots siguiendo el procedimiento explicado en la sección 4.1.2. En este caso, se refleja este desplazamiento actualizando la posición de todas las razas y de sus explotadores.

4.2.5. Estimación final de la posición actual

Tras evaluar todas las razas existentes, se debe seleccionar una de ellas para que sea la localización final del robot. La raza elegida es la que cuenta con mayor número de victorias tras completar la iteración, por lo que en cada iteración se aumenta o disminuye este valor en cada una de las razas.

El primer paso es seleccionar la raza con mayor salud en la iteración actual (R_i). En caso de que la raza seleccionada fuese la ganadora en la anterior iteración (R_p), se aumenta el número de victorias de R_i y se reduce el del resto. Sin embargo, cuando R_i y R_p son diferentes, solo se modifica el número de victorias cuando la diferencia entre la salud de estas dos razas es suficientemente grande.

Esto se realiza para evitar que se produzcan cambios en la raza ganadora cuando la diferencia entre razas es pequeña, puesto que en caso contrario se podrían producir saltos continuos en la localización final del robot, generando inestabilidad. Cuando una raza ha establecido la localización del robot durante muchas iteraciones (lo que indicaría que la

localización es fiable), solo se cambia de raza cuando la diferencia de salud es muy grande, ya que esto significa que la localización actual no es correcta.

Así, el algoritmo selecciona la raza que más victorias tiene tras los cálculos que se acaban de detallar; la posición final del robot se calcula mediante la media ponderada en (X, Y, θ) de los explotadores elitistas de esa raza.

4.2.6. Fiabilidad de la localización

Para calcular la fiabilidad de la localización, se sigue el procedimiento explicado en la sección 4.1.5 para el algoritmo de Monte Carlo, pero utilizando en este caso la salud de la raza ganadora. Al cálculo de fiabilidad anterior se le ha añadido una pequeña variación para tener en cuenta los saltos entre razas, reduciendo la fiabilidad en caso de que esto se produzca.

Si la distancia d entre la raza actual y la raza anteriormente seleccionada es mayor que una distancia máxima d_{max}, la fiabilidad toma el valor 0; en caso contrario, se calcula mediante la ecuación 4.9:

$$F_t = F_t \left(1 - \frac{d}{d_{max}}\right) \tag{4.9}$$

donde el valor de d_{max} dependerá de las dimensiones del escenario.

Esto hace que la fiabilidad siempre disminuya cuando se produce un cambio de raza. No obstante, cuando la distancia entre las razas es pequeña, la fiabilidad apenas se reduce, ya que se entiende que el salto se ha producido para ganar en precisión.

Capítulo 5

Experimentos con mapas conocidos

Los algoritmos explicados en la sección anterior han sido validados experimentalmente usando tanto simuladores como robots reales. Los simuladores utilizados han sido Gazebo y Webots, mientras que como robot real se ha empleado el robot *Nao* de Aldebaran Robotics; todos ellos descritos en el capítulo 3.

Se han utilizado tres entornos para realizar los experimentos. El primero de ellos es un experimento sintético para evaluar el funcionamiento de los algoritmos en entornos totalmente simétricos, el segundo es el entorno de la RoboCup que fue explicado en la sección 3.4 y el último es el entorno de interiores (oficinas) visto en la sección 3.5.

Para cada uno de ellos, se explicarán los cálculos realizados en el modelo de observación, que se corresponderá con la función de coste en el algoritmo de Monte Carlo y con la función de salud en el algoritmo evolutivo.

5.1. Funcionamiento de autolocalización con simetrías

Uno de los aspectos fundamentales que debe cumplir un algoritmo genérico para su utilización en entornos conocidos es su capacidad para poder gestionar varias hipótesis simultáneas. Esta es la motivación fundamental que llevó a diseñar el algoritmo evolutivo, superando la multihipótesis temporal que es capaz de manejar el filtro de partículas.

Dadas dos posiciones del entorno, se admitirá que son simétricas si la información que puede obtener el robot a través de sus sensores es exactamente la misma en ambas posiciones. Así, en caso de que se produzcan simetrías, no será posible determinar por el momento la posición real del robot; por tanto, deberán conservarse ambas posiciones como hipótesis hasta que el robot se desplace y obtenga información excluyente.

En caso de que los algoritmos no gestionen correctamente las simetrías, se corre el riesgo de que se seleccione una de las posiciones prematuramente y ésta resulte incorrecta.

Para evaluar el comportamiento de los algoritmos desarrollados ante este tipo de situaciones, se ha realizado un experimento sintético. Este experimento utiliza un mapa de dos dimensiones de 20x20 metros, en donde se sitúan N posiciones que cumplen con la distribución normal de probabilidad de la Figura 5.1. De esta forma, se crea un entorno sintético donde existirán N simetrías y en el que podrá comprobarse el funcionamiento de los algoritmos sin que se produzca ruido debido a errores en el análisis de las imágenes.

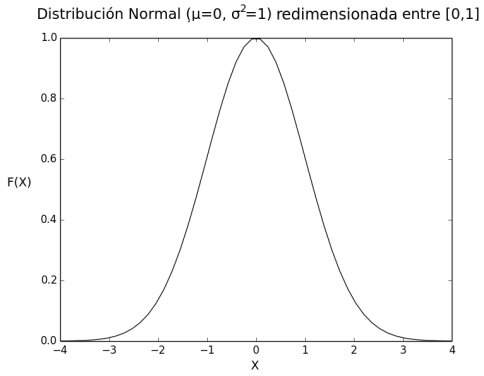


Figura 5.1: Distribución normal ($\mu = 0, \sigma^2 = 1$), redimensionada entre $[0, 1]$.

El modelo de observación calculará la probabilidad de una determinada posición teniendo en cuenta la distancia euclídea entre esta posición y la posición teórica N_i más cercana. Así, en caso de que la distancia sea 0, la probabilidad será 1 y, a medida que esta distancia aumente, disminuirá la probabilidad siguiendo la distribución normal de la Figura 5.1.

5.1.1. Funcionamiento con algoritmo de Monte Carlo

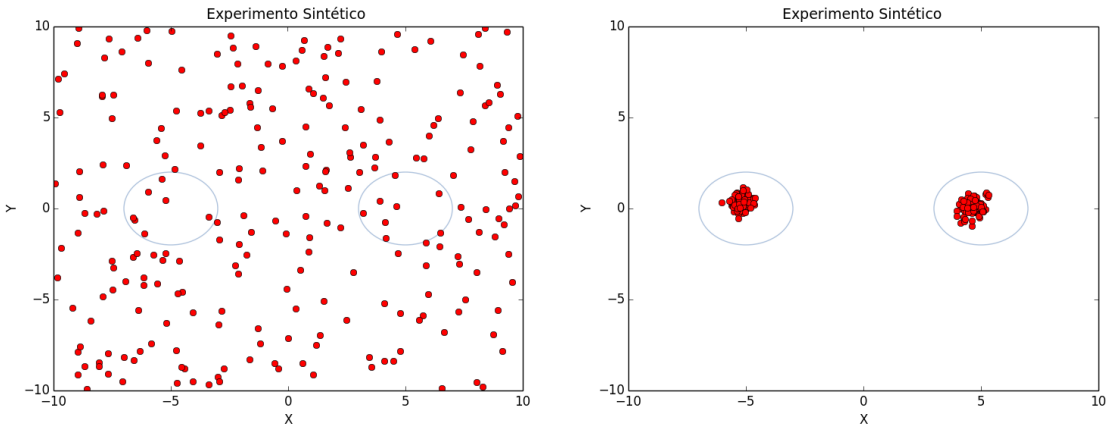
Para este experimento sintético se han utilizado cuatro implementaciones distintas del algoritmo de Monte Carlo:

- El algoritmo clásico de ruleta explicado en la sección 4.1 y que se denotará como **MC**.
- El algoritmo de Monte Carlo aumentado **MCA** descrito en la sección 4.1, que puede recuperarse ante secuestros o localizaciones erróneas sin necesidad de reiniciar todas las partículas.

- Se ha incluido una variante de ambos algoritmos donde el 20 % de sus partículas son catalogadas como elitistas, es decir, no varían de una iteración a otra, obteniendo los algoritmos denotados como **MCe** y **MCAe**.

Para determinar si el algoritmo ha seleccionado correctamente una posición, se calcula el número de partículas que se encuentran como máximo a una distancia euclídea d de la posición correcta ($d = 2$ metros). La solución será válida siempre que exista al menos una partícula dentro de ese radio.

En el caso más básico, utilizando tan solo dos posiciones candidatas N_1 y N_2, el resultado deseado del algoritmo es el que se ve en la Figura 5.2, donde inicialmente se distribuyen aleatoriamente P partículas y, tras una serie de iteraciones, éstas acaban convergiendo en dos grupos de partículas que se corresponden con las dos posiciones candidatas. Estas dos posiciones deberían ser mantenidas en el tiempo indefinidamente puesto que no existe información que permita discriminar entre ellas.



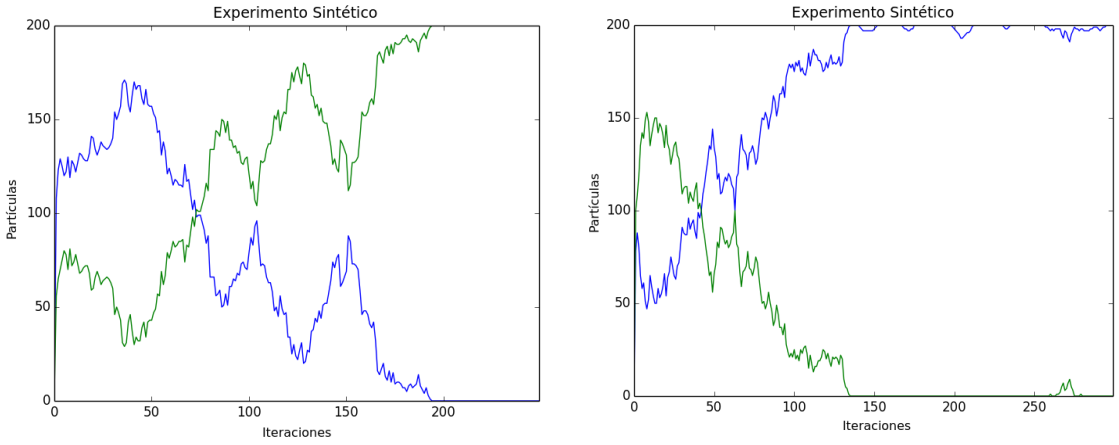
(a) (b)

Figura 5.2: Distribución de partículas con dos posiciones candidatas en experimento sintético con Monte Carlo. Distribución en la primera iteración (a) y tras converger el algoritmo (b).

Resultados con 2 posiciones

Utilizando el caso básico, donde solo existen dos soluciones candidatas, al ejecutar el algoritmo de Monte Carlo Clásico (MC) con 200 partículas, se demuestra experimentalmente como, en la mayoría de las ocasiones, las partículas acaban convergiendo hacia una de las posiciones, perdiéndose por tanto una de las soluciones candidatas.

La Figura 5.3(a) muestra el resultado de una ejecución típica para este caso. La imagen muestra el número de partículas que hay cerca de cada posición con el transcurso de las iteraciones. Inicialmente, las partículas se reparten de forma bastante homogénea entre las dos posiciones y van variando entre ellas hasta que, tras menos de 200 iteraciones, todas las partículas acaban convergiendo en una de las posiciones.



(a) (b)

Figura 5.3: Número de partículas cerca de cada posición en cada iteración con Monte Carlo (a) y Monte Carlo Aumentado (b). Posición N_1 en azul, posición N_2 en verde.

El resto de implementaciones de Monte Carlo realizadas modifican el comportamiento del algoritmo en determinadas situaciones:

- En el caso del Monte Carlo aumentado (MCA), la distribución de partículas funciona como en el algoritmo MC, convergiendo las partículas hacia una de las soluciones. Sin embargo, debido al mecanismo utilizado para buscar nuevas posiciones, es posible que vuelva a encontrar alguna de las soluciones candidatas que se habían perdido, aunque es difícil que se recupere totalmente puesto que esa posición contará inicialmente con muy pocas partículas.

  Esto es lo que sucede en la Figura 5.3(b), donde tras converger todas las partículas hacia una de las posiciones (iteración 130), vuelve a recuperarse la solución candidata perdida (iteración 260); sin embargo, al contener tan pocas partículas, esta solución no se mantiene por mucho tiempo y se vuelve a perder.

- Cuando se emplea el elitismo en los algoritmos MCe y MCAe, es más factible que se mantengan las distintas posiciones a medida que avanzan las iteraciones. Esto es debido a que los elitistas irán convergiendo hacia las posiciones de las

soluciones candidatas, siendo suficiente con que una sola partícula elitista se posicione correctamente en N_1 y N_2 para que se mantengan ambas posiciones indefinidamente.

Sin embargo, siguen existiendo casos en los que se puede perder alguna de las posiciones. Esto ocurre cuando la convergencia de las partículas hacia una de las posiciones es demasiado rápida y no da tiempo a que se creen elitistas en la otra solución.

Resultados con varias posiciones

El caso expuesto anteriormente es el más sencillo, existiendo tan solo dos posiciones simétricas. Sin embargo, en ciertos entornos pueden existir más de dos simetrías a la vez, por ejemplo, en el entorno de oficinas explicado en el capítulo 3. Por ello, se ha generado una batería de pruebas para cada uno de los algoritmos de Monte Carlo implementados. Además, se han variado tanto el número de partículas utilizadas (de 100 a 1000) como el número de soluciones candidatas (de 2 a 4), realizando 100 ejecuciones de cada algoritmo con estas configuraciones. Se considera que el algoritmo ha funcionado correctamente si tras 300 iteraciones aún conserva alguna partícula en cada posición.

Se han elegido 300 iteraciones como punto de referencia, puesto que se corresponden con 10 segundos de ejecución de un algoritmo funcionando en tiempo real a 30 FPS. A continuación se muestran los resultados obtenidos con cada algoritmo:

Algoritmo MC:

Nº Partículas	2 Posiciones	3 Posiciones	4 Posiciones
100	0	0	0
200	10	0	0
500	47	12	0
1000	76	49	16

Algoritmo MCA:

Nº Partículas	2 Posiciones	3 Posiciones	4 Posiciones
100	9	2	0
200	13	1	0
500	43	16	4
1000	79	56	18

Algoritmo MCe con elitistas:

Nº Partículas	2 Posiciones	3 Posiciones	4 Posiciones
100	11	0	0
200	30	8	3
500	79	51	29
1000	96	92	84

Algoritmo MCAe con elitistas:

Nº Partículas	2 Posiciones	3 Posiciones	4 Posiciones
100	9	2	0
200	37	9	2
500	81	47	34
1000	96	90	77

Estos resultados permiten llegar a varias conclusiones. Los algoritmos sin elitistas (MC y MCA), tienen resultados similares, aunque el segundo funciona levemente mejor debido a la búsqueda continua de nuevas posiciones comentada anteriormente. Por su parte, los algoritmos con elitismo (MCe y MCAe) funcionan notablemente mejor, estando cerca del 100 % de soluciones válidas cuando existen pocas soluciones candidatas y muchas partículas.

Para que este funcionamiento tan eficiente se de en la realidad hay que tener en cuenta varios aspectos:

- El número de posiciones simétricas tiene que ser bajo, algo que no siempre sucede.
- Al utilizar un gran número de partículas la eficiencia temporal del algoritmo también disminuye, por lo que, según el contexto, puede no ser viable tener tantas partículas en el mundo.
- El entorno donde se ha realizado este experimento sintético es totalmente estático, no existiendo errores de desplazamiento ni de procesamiento de las imágenes.

Eficiencia temporal

Otro de los factores fundamentales que influye en el funcionamiento de los algoritmos de Monte Carlo implementados es el número de partículas utilizadas.

A pesar de que para este experimento sintético no se ha limitado el número de partículas, en condiciones reales este factor es determinante para poder trabajar en tiempo real. Así, en la implementación realizada para este experimento sintético, desarrollada en el lenguaje de programación Python, la evolución de la eficiencia temporal en función del número de partículas ha sido la siguiente:

Algoritmo	100	200	500	1000
MC	17.86 ms	35.63 ms	87.89 ms	178.02 ms
MCA	17.37 ms	34.11 ms	88.35 ms	172.72 ms
MCe	17.71 ms	36.60 ms	92.84 ms	195.10 ms
MCAe	17.60 ms	36.54 ms	94.12 ms	196.34 ms

Se observa como los cuatro algoritmos tienen una eficiencia temporal similar, progresando de forma lineal según el número de partículas y siendo algo mayor el tiempo en caso de utilizar elitistas.

Resultados en entornos dinámicos

Para ver qué pasaría en un entorno algo más realista, se sometieron los algoritmos que mejor funcionaron en los experimentos anteriores (los que contaban con partículas elitistas) a un entorno levemente dinámico.

Para ello, cada una de las posiciones candidatas se desplazan 1 centímetro en cada iteración, por lo que, tras las 300 iteraciones realizadas en las pruebas, se desplaza un total de 3 metros.

Para ilustrar cómo afecta este leve desplazamiento a los algoritmos, a continuación se muestran los resultados de los algoritmos MCe y MCAe utilizando mil partículas:

Algoritmo	2 Posiciones	3 Posiciones	4 Posiciones
MCe	75	40	20
MCAe	74	43	21

Puede apreciarse como los resultados empeoran notablemente en ambos casos con este leve desplazamiento. Esto es debido a que ya no es suficiente con que las mejores partículas elitistas permanezcan de forma indefinida cerca de las soluciones buscadas, sino que tendrán que ir cambiando con el paso de las iteraciones según se desplacen las posiciones candidatas.

Este experimento lleva a la conclusión de que las distintas implementaciones del algoritmo de Monte Carlo realizadas no pueden ser utilizadas en entornos con simetrías si se requiere una localización robusta y fiable.

5.1.2. Funcionamiento con algoritmo evolutivo

En el caso del algoritmo evolutivo basta con utilizar la implementación que se explicó en el capítulo 4.2. Este algoritmo se caracteriza por separar a los individuos encargados de buscar nuevas posiciones (exploradores) de los encargados de mejorar la posición actual (explotadores). Por lo tanto, los resultados del algoritmo dependerán en gran medida de los exploradores utilizados.

Así, si el número de exploradores es bajo en comparación con el tamaño del mapa, se prolongará el tiempo necesario para encontrar las posiciones candidatas, pero una vez encontradas se mantendrán a lo largo del tiempo en razas separadas.

Resultados con 2 posiciones

Tomando un ejemplo similar al visto para Monte Carlo, se han creado dos posiciones candidatas N_1 y N_2, que tendrán que ser encontradas y mantenidas en el tiempo por el algoritmo evolutivo.

Para este experimento se utilizan 10 exploradores en cada iteración, que buscarán posiciones donde crear nuevas razas; cada raza se compondrá de 20 explotadores en busca de la mejor solución. En la Figura 5.4 se puede ver el resultado de una ejecución típica con la configuración dada.

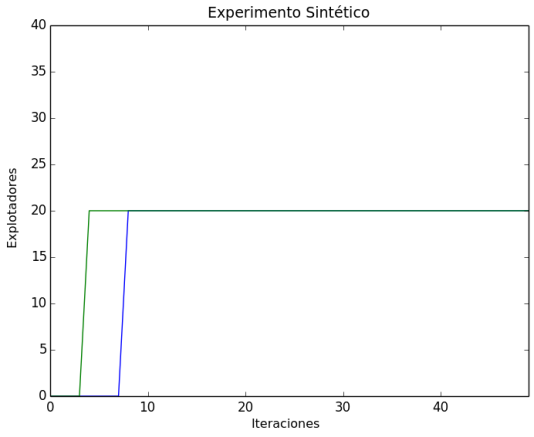


Figura 5.4: Número de explotadores en cada raza del algoritmo evolutivo en experimento sintético. Posición N_1 en azul, posición N_2 en verde.

El algoritmo tarda más en encontrar las posiciones que en los algoritmos de Monte Carlo vistos hasta ahora, ya que solo cuenta con 10 exploradores buscando posiciones candidatas. Una forma simple de solucionar lo anterior es hacer que el número de exploradores sea

dinámico, incrementando su número al arrancar el algoritmo o cuando las razas tienen una baja probabilidad media.

Además, al contrario de lo que sucedía en Monte Carlo, una vez localizadas las posiciones candidatas, éstas se mantienen el tiempo que sea necesario, puesto que los individuos explotadores de cada raza no compiten entre ellos.

Resultados con varias posiciones

Se han realizado varios experimentos en el mismo escenario empleado para el algoritmo de Monte Carlo, variando para este caso el número de posiciones y de exploradores. Cada configuración del algoritmo ha sido de nuevo ejecutada 100 veces durante 300 iteraciones, comprobando tras ese tiempo en cuántas ocasiones se han encontrado y mantenido todas las posiciones. Además, se supondrá que son constantes tanto el número de razas (4) como el número de explotadores (20).

Nº Exploradores	2 Posiciones	3 Posiciones	4 Posiciones
1	100	100	96
5	100	100	100
10	100	100	100
20	100	100	100
50	100	100	100

Como puede apreciarse, incluso cuando el número de exploradores es muy bajo, el algoritmo es capaz de encontrar todas las posiciones candidatas y de mantener su posición hasta el final. Sin embargo, la diferencia que existe según el número de exploradores utilizados es el número de iteraciones que tarda el algoritmo en encontrar todas las posiciones.

A continuación se muestra el número medio de iteraciones que ha tardado cada configuración del algoritmo en encontrar todas las posiciones:

Nº Explotadores	2 Posiciones	3 Posiciones	4 Posiciones
1	78.58	90.22	104.94
5	14.6	17.88	22.96
10	8.67	9.61	12.34
20	4.38	5.18	6.38
50	2.24	2.49	4.39

Eficiencia temporal

Se ha realizado una implementación del algoritmo en Python, la cual ha sido ejecutada utilizando los mismos recursos que los empleados para los experimentos con Monte Carlo.

En la siguiente tabla se puede ver el tiempo medio de ejecución de cada iteración del algoritmo evolutivo según el número de exploradores y de explotadores de cada raza, suponiendo que el número máximo de razas es constante (4):

Explotadores \ Exploradores	1	5	10	20	50
5	2.34 ms	4.59 ms	5.65 ms	7.72 ms	13.14 ms
10	6.47 ms	8.00 ms	8.72 ms	10.43 ms	16.19 ms
20	8.52 ms	13.77 ms	15.82 ms	16.92 ms	23.13 ms
50	15.21 ms	29.25 ms	34.10 ms	37.60 ms	43.33 ms

Como se ve, al igual que sucedía con el algoritmo de Monte Carlo, la eficiencia temporal del algoritmo evolutivo depende del número total de individuos utilizados (ya sean explotadores o exploradores).

De hecho, los tiempos de ejecución de ambos algoritmos son similares cuando se utiliza el mismo número de partículas/individuos. Por ejemplo, en el caso de utilizar 20 exploradores en cada iteración y 20 explotadores por raza (100 individuos en total para 4 razas), el tiempo cómputo de una iteración es de alrededor de 17 ms, un tiempo similar al que se obtenía con Monte Carlo cuando se utilizaban 100 partículas.

Sin embargo, los resultados del algoritmo evolutivo utilizando estos 100 individuos son favorables, mientras que los obtenidos con Monte Carlo con 100 partículas eran bastante pobres, tal y como se vio en la sección anterior. Esto se debe a que el algoritmo evolutivo puede realizar un balance explícito entre individuos exploradores y explotadores, mientras que en Monte Carlo no existe esta distinción.

A tenor de los resultados, se concluye que en entornos con simetrías es más recomendable utilizar el algoritmo evolutivo propuesto en esta tesis en lugar de los algoritmos de partículas como Monte Carlo, tanto por su desempeño con las simetrías como por su eficiencia temporal.

5.2. Experimentos en la RoboCup

Los algoritmos descritos en el capítulo 4 deben ser modificados para cada escenario, detallando el análisis necesario de las imágenes y el modelo de observación utilizado, que en el algoritmo evolutivo se corresponde con la función salud y en el algoritmo de Monte Carlo con el modelo probabilístico de observación.

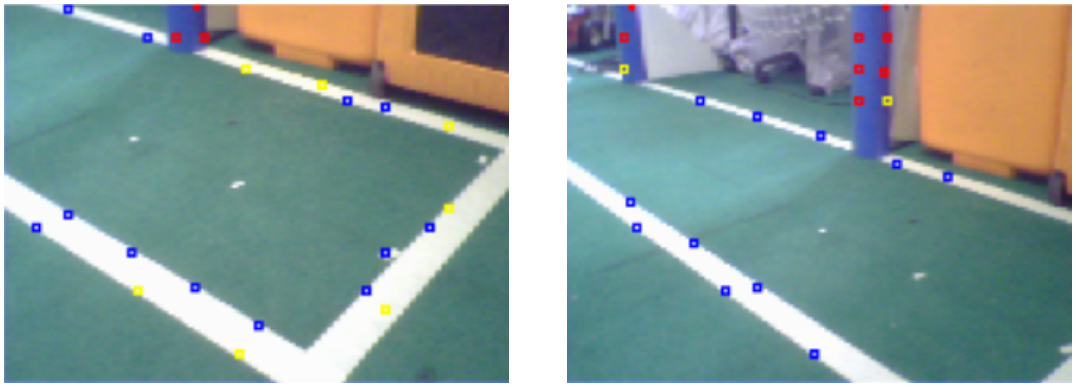
Como queda expuesto en la sección 3.4, el campo de la RoboCup se compone de un campo de fútbol en miniatura, que cuenta con porterías, líneas y una pelota para jugar. Además, pueden existir obstáculos como robots o elementos externos que entren al campo.

5.2.1. Análisis de la imagen 2D: Píxeles informativos

El análisis de las imágenes se realiza en cada iteración de los algoritmos, siendo por ello un factor fundamental la eficiencia temporal del análisis realizado para que los algoritmos funcionen en tiempo real. Mediante este análisis se intenta hallar en las imágenes aquellos elementos más significativos del campo de la RoboCup (las líneas y las porterías), porque son los que aportan información indirecta de autolocalización.

Este análisis no pretende obtener todos los píxeles de líneas y porterías disponibles en la imagen, sino un subconjunto que sea suficiente para estimar la localización. Esto permite agilizar el procesamiento de las imágenes y evita proporcionar información redundante al modelo de observación.

En la Figura 5.5 se muestra un ejemplo del resultado final del análisis realizado. Los píxeles azules pertenecen a líneas, los rojos a porterías y los amarillos son píxeles redundantes descartados de forma aleatoria.



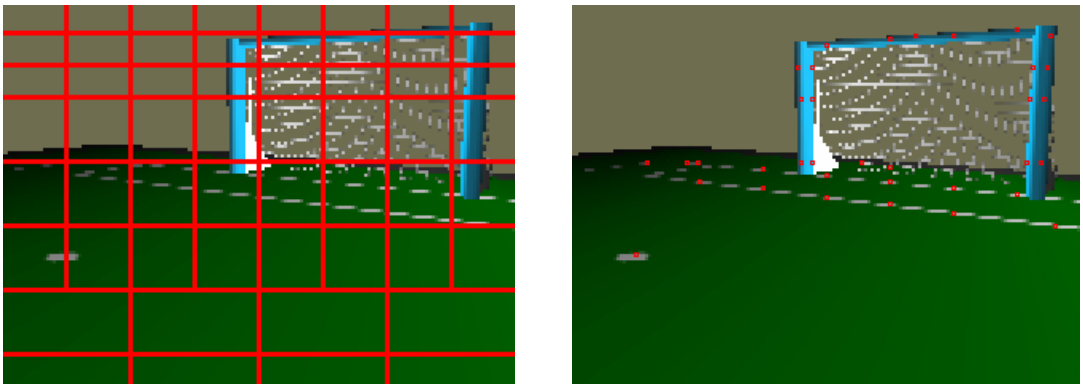
(a) (b)

Figura 5.5: Selección de píxeles informativos en el escenario de la RoboCup.

Para llegar a este resultado, se realiza un muestreo asimétrico de la imagen y posteriormente se filtran los píxeles de líneas y porterías. A continuación se explicarán en detalle cada uno de estos pasos:

Muestreo asimétrico de la imagen

Al tener que realizar un análisis de las imágenes en el menor tiempo posible no es eficiente tratar de analizar toda la imagen al completo. Por ello, se ha realizado un muestreo asimétrico de la imagen como el que se muestra en la Figura 5.6(a). Este muestreo es asimétrico para que se comprueben de forma más exhaustiva los puntos más alejados, los cuales se situarán en la parte superior de la imagen y tendrán una menor resolución.



(a) (b)

Figura 5.6: Zonas de búsqueda (a) y píxeles característicos seleccionados (b).

A lo largo de cada una de las rectas verticales y horizontales con las que se recorre la imagen, se realizan para cada píxel varios filtros de color para tratar de identificar el elemento del campo que se observa. Así, se identifican los colores amarillo y azul para las porterías, el color blanco para las líneas y el color naranja para la pelota, clasificando como desconocido cualquier otro color.

En caso de que varios píxeles consecutivos sean identificados con el mismo color, se seleccionan tan solo los extremos, con el fin de obtener los bordes de estos elementos. Con este barrido se obtienen una serie de píxeles como los que se destacan en rojo en la Figura 5.6(b).

Entre los píxeles obtenidos es posible encontrar falsos positivos que harían que el algoritmo se comportase de forma incorrecta; por ello, se realizan una serie de filtros para identificar estos falsos positivos y eliminarlos, consiguiendo así que los píxeles seleccionados sean más fiables.

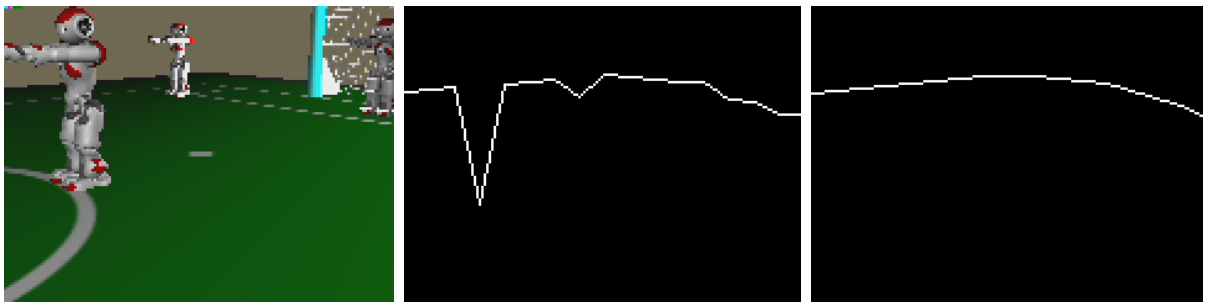
El primer filtro consiste en eliminar todos aquellos píxeles aislados que no tengan otros del mismo color a su alrededor, considerando estos píxeles como ruido en la imagen.

Se realizan además los siguientes filtros específicos para las líneas y porterías:

Filtrado de líneas

Debido a las características del entorno de la RoboCup, todos los píxeles detectados de color blanco (líneas) que se encuentren fuera del campo no son válidos, por lo que se consideran falsos positivos.

Para estimar el fin del campo, se recorre la imagen en vertical cada pocos píxeles y se comprueba dónde comienza el color verde en cada columna. Con este método, se obtiene un fin del campo como el que puede verse en la Figura 5.7(b).



(a) (b) (c)

Figura 5.7: Cálculo del fin del campo mediante *Convex Hull*.

Los saltos que se producen se han eliminado utilizando el algoritmo *Convex Hull* [Barber *et al.*, 1996]. Este algoritmo toma como entrada N puntos y calcula el subconjunto convexo con el menor número de puntos pertenecientes a N que hace que los N puntos se encuentren en su interior. En este caso, el resultado obtenido sería el que se muestra en la Figura 5.7(c).

Asimismo, también se han filtrado aquellos píxeles blancos pertenecientes al propio robot o a otros robots que se encuentren en el campo.

Filtrado de porterías

El filtro de porterías desarrollado basa su funcionamiento en el conocimiento de la geometría de las porterías y hace uso de un nuevo elemento conocido como *blob*.

Los *blobs* engloban en uno o varios objetos los elementos que aparecen en la imagen. A partir de una imagen filtrada con los colores de la portería, se utiliza la librería *cvBlob*[1] para que devuelva los objetos (porterías) que aparecen en la imagen.

Los *blobs* obtenidos son posteriormente validados en función su geometría, teniendo en cuenta su relación de aspecto, su tamaño, etc, para aceptar solo los que puedan corresponderse con una portería. La Figura 5.8 muestra los *blobs* de la portería detectados en una imagen.

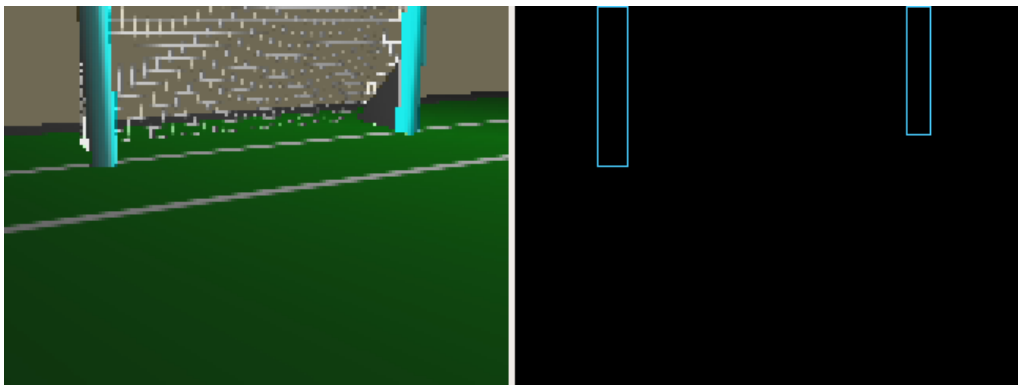


Figura 5.8: Segmentación con *Blobs* de los postes de la portería.

Para cada uno de los píxeles detectados que se correspondían con colores de porterías, han sido validados aquellos que se encontraban dentro de alguno de los *blobs* obtenidos, descartando así el resto de píxeles.

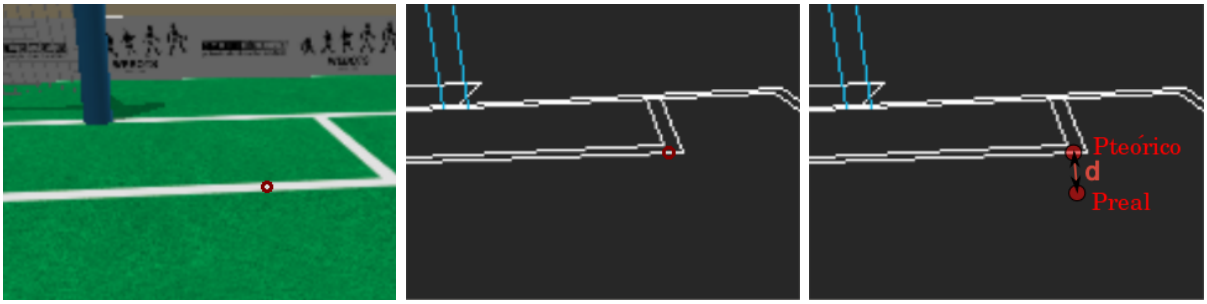
5.2.2. Modelo de observación sensorial

Una vez analizadas las imágenes proporcionadas por los robots, el siguiente paso es traducir esto a información de posición, es decir, saber si una posición es verosímil con los datos de entrada. Dada una posición (X, Y, θ), es posible calcular en qué grado se parece la imagen que vería el robot desde esa posición (imagen teórica) con la imagen observada (imagen real). Si ambas imágenes son parecidas, entonces esa posición será plausible, mientras que si son muy distintas esa posición será improbable.

Tras obtener los píxeles característicos, se calculará la "correlación" entre las dos imágenes a partir de la distancia (en píxeles) existente entre cada píxel encontrado en la imagen real y su homólogo en la imagen teórica, lo que da como resultado una probabilidad que indica cómo de verosímil es la posición dada respecto de la imagen observada.

[1] `http://code.google.com/p/cvblob/`

Se ejemplifica esto último en la Figura 5.9. Dado un píxel en la imagen real (P_{real}), se busca el píxel del mismo color más cercano en la imagen teórica ($P_{teórico}$), que se encontrará a una distancia d, siendo $d \geq 0$.



(a) (b) (c)

Figura 5.9: Distancia entre la imagen real (a) y la teórica (b, c) para un píxel dado.

Al realizar el mismo procedimiento para cada píxel seleccionado en la imagen real, se podrá calcular la probabilidad final mediante la media aritmética de las distancias según la ecuación 5.1. De esta forma, cuando la distancia media sea igual o mayor a D píxeles (parámetro configurable que tendrá que ser calculado experimentalmente), la probabilidad será 0, y ésta aumentará a medida que disminuya la distancia media:

$$P = 1 - \frac{\sum_{i=0}^{N} \frac{d_i}{N}}{D} \tag{5.1}$$

Precálculo de puntos cercanos en 3D

Para comparar cada punto de la imagen real con su equivalente en la imagen teórica, una posible solución es generar en tiempo de ejecución la imagen teórica completa para una determinada posición, con lo que se obtendría una imagen simulada similar a la Figura 5.9(b). Sin embargo, este cálculo es muy costoso computacionalmente, por lo que en la práctica estas distancias son precalculadas.

Tener precalculadas las distancias entre cada píxel en la imagen y cada posible posición del robot requeriría mucha memoria, incluso para mundos pequeños como el de la RoboCup. Por otra parte, las líneas siempre se encuentran en el plano suelo (es decir, con altura 0), mientras que las porterías siempre se encuentran en un plano vertical. Al dividir estos planos 3D cada pocos centímetros, se crea una malla de puntos en 3D con un número de puntos suficientemente pequeño como para ser abordable su manejo en tiempo de ejecución.

Siguiendo esta técnica, para cada punto 3D en el plano se ha precalculado la posición del objeto del mismo tipo más cercano (una línea o una portería). Así, se muestra en la Figura 5.10 el mecanismo utilizado para calcular la distancia con una línea, que se realiza tal y como se expone a continuación:

1. Se calcula el punto 3D P'_{real} que se corresponde con el píxel P_{real} en la imagen, teniendo en cuenta la posición del robot. Para ello, se retroproyecta este píxel en 3D y se calcula la intersección de la recta obtenida con el plano suelo.

2. Se consulta en memoria el punto 3D más cercano a P'_{real} donde se encuentra una línea, obteniendo así $P'_{teórico}$.

3. Este punto 3D $P'_{teórico}$ es proyectado en la imagen para obtener el punto $P_{teórico}$.

4. Con esto, se calcula la distancia en píxeles entre P_{real} y $P_{teórico}$ y se obtiene así la verosimilitud de esta posición.

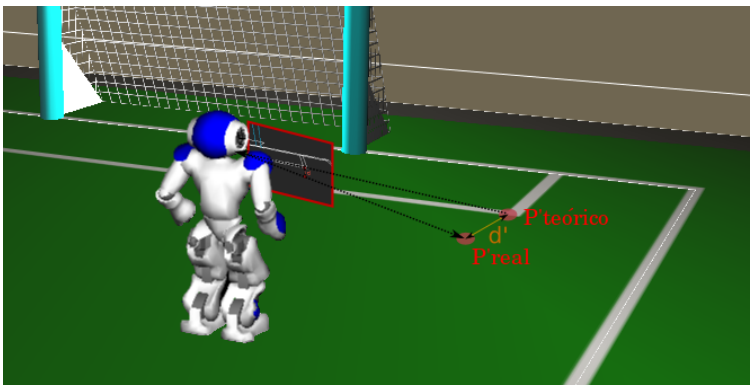
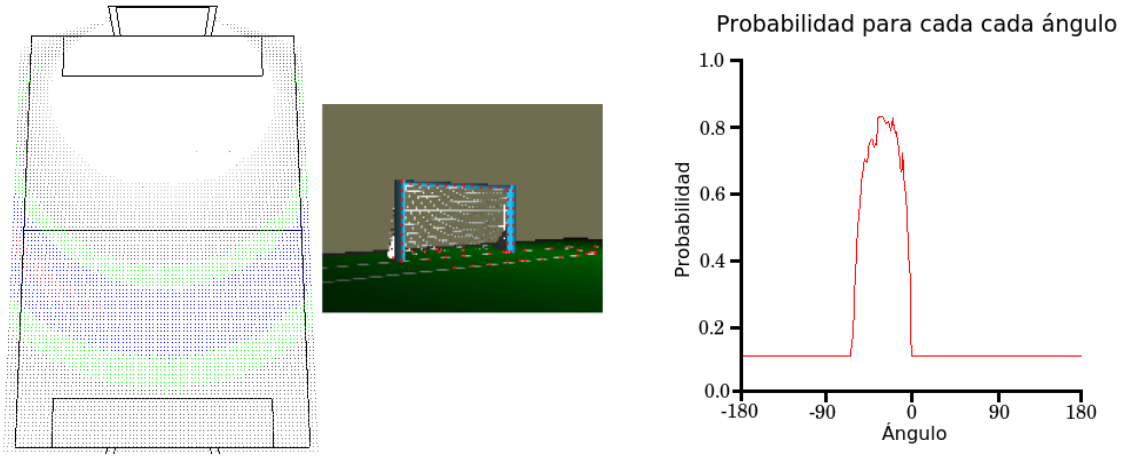


Figura 5.10: Cálculo de distancia entre la imagen real y la teórica con puntos precalculados.

Esta técnica aumenta la eficiencia temporal del modelo de observación notablemente (cerca de un orden de magnitud), mejorando así la eficiencia de los algoritmos.

Discriminación respecto a la posición

El resultado del modelo de observación tiene que guardar un equilibrio a la hora de discriminar una posición. Un modelo de observación muy excluyente haría que la posición real fuese difícil de encontrar, mientras que uno poco excluyente podría dar como válidas posiciones que no fuesen realistas. Además, un buen modelo de observación tendría que comportarse de forma robusta cuando los datos de entrada no sean ideales, lo que se podría producir en caso de oclusiones, falsos positivos, etc.



(a) (b)

Figura 5.11: Discriminación del modelo de observación en el escenario de la RoboCup: probabilidad calculada en cada posición (a) y en cada ángulo dada la posición real (b). Mayor probabilidad en rojo, seguido de azul, verde, negro y por último blanco.

En las imágenes de la Figura 5.11 se plasma el resultado de un experimento típico. En la imagen de la izquierda se muestra cuál sería la probabilidad calculada en cada posición del campo a partir de la imagen de entrada, mientras que en la imagen derecha se presenta el resultado obtenido en cada ángulo suponiendo una posición (X, Y) real. Se observa cómo la zona de mayor probabilidad se corresponde con el lugar donde se encuentra realmente el robot (lateral izquierdo del campo). Además, cuando se fija la posición del robot, el ángulo de rotación con mayor probabilidad también es el adecuado (alrededor de -30 grados).

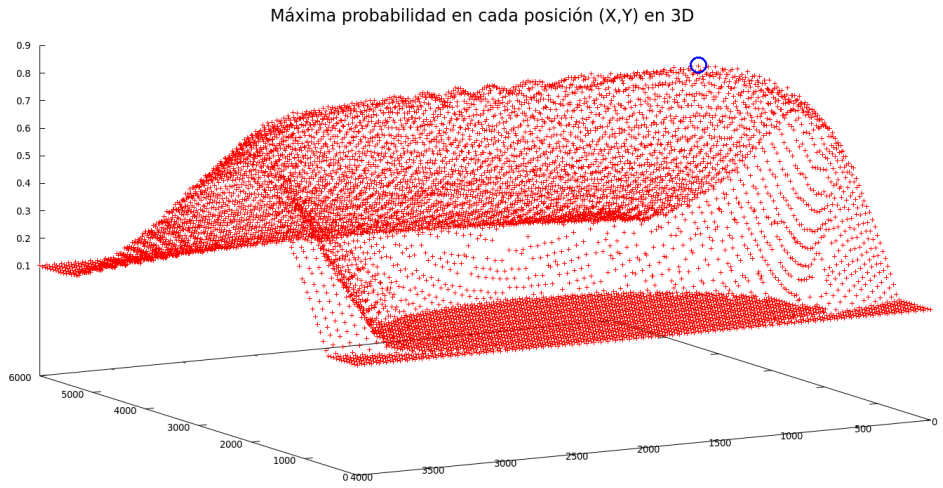


Figura 5.12: Probabilidad obtenida en cada posición del campo de la RoboCup.

Para comprobar si el modelo de observación discrimina adecuadamente, se muestran en la Figura 5.12 las probabilidades obtenidas con el ejemplo anterior en 3D. El crecimiento de la probabilidad es progresivo y tiene su máximo en la posición adecuada (lateral izquierdo del campo), por lo que la discriminación realizada es suficientemente suave y rigurosa como para permitir la convergencia de forma correcta. Sin embargo, este crecimiento también cuenta con máximos locales, lo que tendrá que ser tenido en cuenta a la hora de diseñar y utilizar los algoritmos de localización.

Efecto de las simetrías

Al existir simetrías en el campo, es posible encontrarse con situaciones como la de la Figura 5.13, en donde a partir de un único poste es imposible asegurar cuál es la posición del robot, obteniendo por tanto una alta probabilidad en varias zonas del campo, sobre todo cerca de la portería.

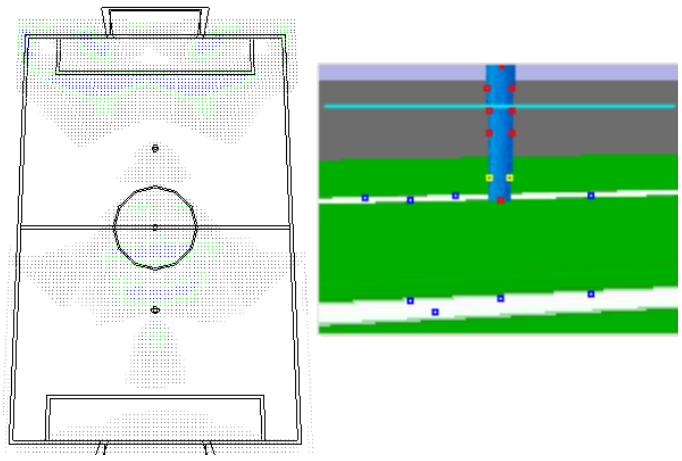


Figura 5.13: Efecto de las simetrías en el resultado del modelo de observación.

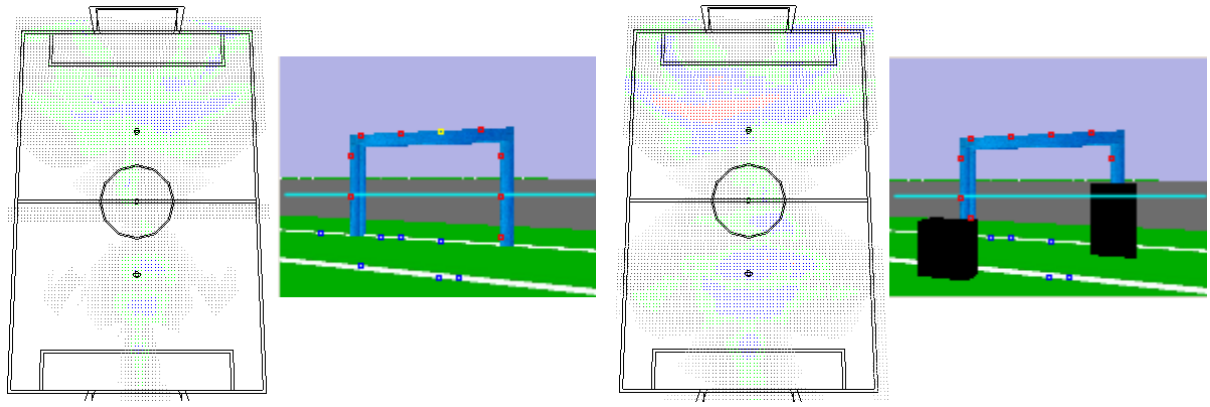
Sorprende el hecho de que haya una alta probabilidad detrás del círculo central del campo, aunque es lógico puesto que en esa posición existiría una línea delante del robot (la línea que divide cada campo) y se vería alguno de los postes en la zona central de la imagen.

Para solucionar esto, el modelo de observación podría haber penalizado el hecho de que no se vean objetos que deberían visualizarse desde la posición en la que se encuentra el robot. Por ejemplo, en este caso, desde el centro del campo se deberían ver dos postes, algo que no sucede. Sin embargo, si se añade esta característica al modelo de observación, el robot se perdería fácilmente cuando existiesen oclusiones, por lo que finalmente se decidió descartar este comportamiento.

En el caso de que solo se vean líneas (algo bastante frecuente), el número de simetrías que se encontrarán será todavía mayor, e incluso en observaciones con tan solo una línea se obtendrá una probabilidad alta en muchas zonas del campo.

Efecto de las oclusiones

Tal y como acaba de ser expuesto, las oclusiones no afectan negativamente a probabilidad calculada. El modelo de observación presentado solo evalúa la compatibilidad de la observación efectuada con la imagen teórica que se obtendría en una posición dada, pero no comprueba si todo lo que se debería visualizar está en la imagen. En la Figura 5.14 se puede ver un ejemplo de cómo influirían las oclusiones en el resultado obtenido.



(a) (b)

Figura 5.14: Efecto de las oclusiones en el resultado del modelo de observación. Porterías sin (a) y con (b) oclusiones.

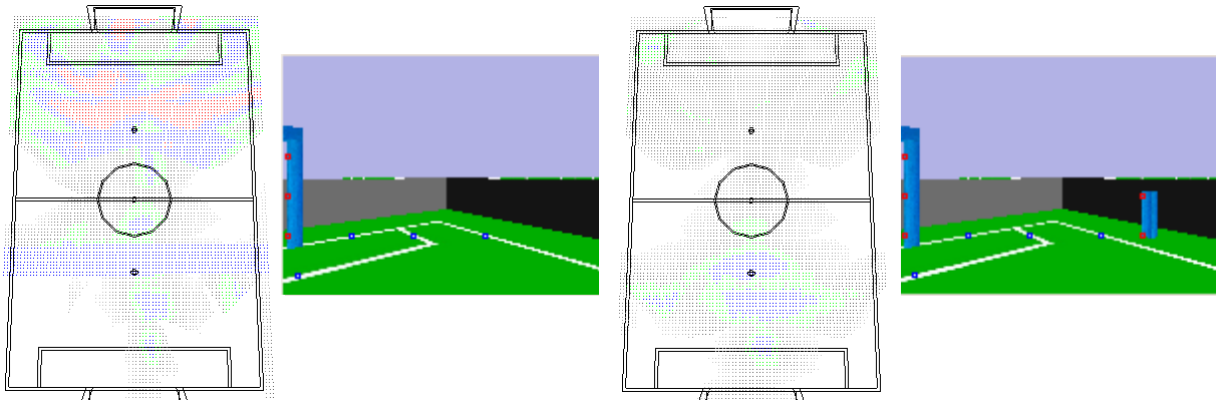
En la primera imagen se muestra el resultado del modelo de observación al ver una portería sin oclusiones, siendo la zona con mayor probabilidad la que se encuentra a 1-2 metros de la portería por la zona central del campo.

En la segunda imagen se han incluido dos elementos que simularían oclusiones producidas por otros robots presentes en el campo. Al no penalizar los elementos que no se ven en la observación, las zonas con alta probabilidad de la primera imagen mantienen una alta probabilidad, pero aparecen nuevas zonas plausibles debido a las simetrías.

Se concluye que el hecho de que se produzcan oclusiones dificulta la localización del robot, pudiendo hacer que su localización sea errónea si estas oclusiones se mantienen durante muchas observaciones.

Falsos positivos

Un aspecto que afecta en gran medida al resultado del modelo de observación es utilizar como datos de entrada píxeles que no son válidos, es decir, falsos positivos. Estos falsos positivos podrían hacer que cambiase totalmente el resultado final, sobre todo en el caso de que el falso positivo se produjese con las porterías. En la Figura 5.15 se expone esta situación.



(a) (b)

Figura 5.15: Falsos positivos en el campo de la RoboCup.

En la primera imagen se observa cómo, de nuevo, las zonas con mayor probabilidad se sitúan en torno a 1-2 metros de la portería y son adecuadas a la realidad. Al añadir un falso poste en la segunda imagen, se producen falsos positivos que hacen que el resultado cambie totalmente, dando a las zonas cercanas a la portería muy poca probabilidad, lo que produciría que fallasen los algoritmos de localización.

Debido a que los falsos positivos son una de las causas principales de una mala localización, se ha hecho especial énfasis en la creación de filtros para minimizarlos.

Calidad de la observación

El objetivo del análisis de las imágenes es servir como base para la localización. Tal y como se vio en el capítulo 4, la localización entrega una posición estimada acompañada de un valor de fiabilidad, que está influenciado en gran medida por la calidad de la observación obtenida.

La calidad de la observación se obtiene heurísticamente a partir de dos elementos: el número de píxeles pertenecientes a líneas y el número de píxeles pertenecientes a porterías.

Así, se puede llegar a 1 (mejor calidad) en el caso de ver el máximo número de píxeles permitidos tanto para las líneas como para las porterías.

5.2.3. Precisión en simulación

Para validar la precisión del algoritmo desarrollado, se han realizado numerosas pruebas tanto en simuladores como robots reales. Una gran ventaja de los simuladores es que permiten conocer en todo momento la posición real del robot, haciendo que la comparativa entre la posición real y la calculada sea muy precisa.

El primero de los experimentos está realizado en el simulador Webots y consiste en iniciar los algoritmos de localización con el robot situado en una posición estática; transcurrido un periodo de tiempo razonable, en el que los algoritmos tienen que haber podido situar al robot en la posición correcta, el robot se comienza a desplazar por todo el campo. Con esto, se puede comprobar si los algoritmos son capaces de seguir la trayectoria realizada. Para mostrar los resultados se dibuja la trayectoria real seguida por el robot (en verde) y la trayectoria calculada por cada algoritmo (en rojo).

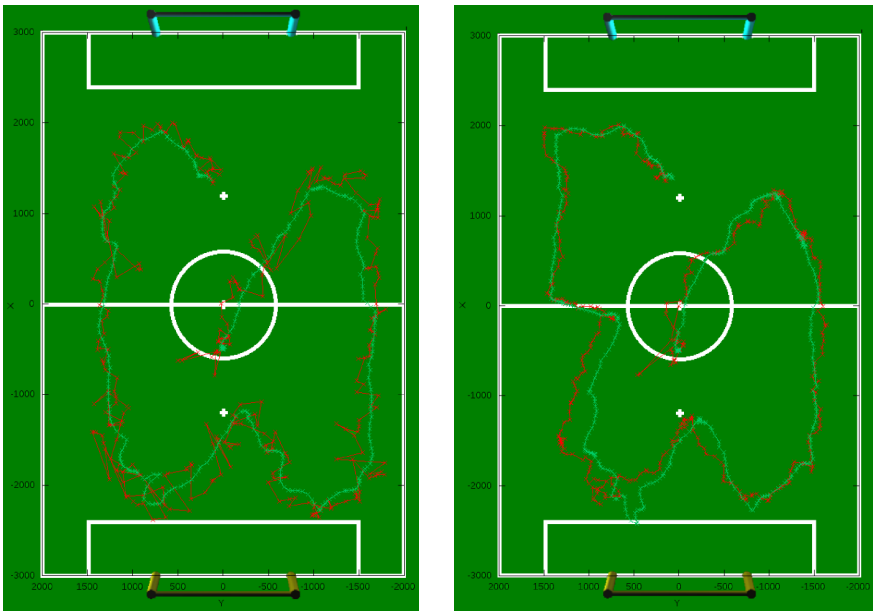
Monte Carlo

El algoritmo de Monte Carlo utilizado para los experimentos ha sido el algoritmo de Monte Carlo aumentado con elitismo (*MCAe*), puesto que es el más avanzado de los desarrollados y el que mejores resultados ha dado en el experimento sintético.

Los valores configurables en el algoritmo son: el número de partículas utilizadas y el porcentaje de partículas elitistas en cada iteración. Tras probar diversas soluciones, se escogieron 200 partículas, de las cuales un 10 % son elitistas, puesto que son los valores que mejor resultado han dado en la relación coste computacional-precisión.

El resultado del experimento es el que se muestra en la Figura 5.16(a). Como puede verse, el algoritmo se localiza inicialmente de forma correcta y es capaz de mantener la estimación de posición a medida que el robot se mueve sin perderse, siendo el error medio en posición de 17.38 cm y en ángulo de 12.03 grados, valores suficientemente buenos como para ser utilizados para realizar otros comportamientos en la RoboCup.

Uno de los hechos que destacan es que el cambio en la estimación entre una medición y otra (que están realizadas cada segundo) es muy brusco, oscilando siempre sobre la trayectoria real.



(a) (b)

Figura 5.16: Desplazamiento en simulador con Monte Carlo (a) y algoritmo evolutivo (b).

Algoritmo Evolutivo

El algoritmo evolutivo cuenta con más parámetros configurables que el método de Monte Carlo. Estos parámetros son: el número máximo de razas, los explotadores en cada raza, el porcentaje de elitistas en cada raza y el porcentaje de individuos que evolucionan mediante cruce. Para cada uno de estos valores se han realizado distintas pruebas, seleccionándose finalmente aquellos valores que mejor se comportaban en este escenario con una eficiencia temporal razonable. Así, el número de razas se ha establecido en 4, el número de explotadores en 30 para cada raza, el porcentaje de elitistas en el 20 % y el porcentaje de cruce también en el 20 %.

El número máximo de razas ha sido un factor fundamental. Cuanto mayor fuese el número de razas mayor sería la precisión espacial obtenida; sin embargo, este aumento de la precisión mermaría la eficiencia temporal, aumentando el tiempo de ejecución de forma lineal.

En la Figura 5.16(b) se puede ver una ejecución típica de este experimento. Al igual que sucedía en Monte Carlo, el algoritmo no pierde la posición del robot, siendo el error medio en posición de 14.24 cm y en ángulo de 8.42 grados. La precisión conseguida es bastante buena y es suficiente para utilizarse en las aplicaciones en el escenario de la RoboCup.

En este caso, el cambio que se produce en la estimación entre cada medición es más

suave que en el caso del algoritmo de Monte Carlo. Esto último se debe al funcionamiento de los explotadores de cada raza, que son capaces de acercarse más al máximo local de la posición calculada.

5.2.4. Precisión en el robot real

A diferencia de lo que sucede en los simuladores, al realizar experimentos en el robot real no se dispone con verdad absoluta. Para calcular el error en la localización, se han tomado varios puntos de referencia en los que se han realizado las mediciones.

Además, hay otros factores asociados al uso de robots reales que hacen de éste un escenario más exigente que el simulador:

- Pueden existir *falsos positivos* al analizar las imágenes, ya que a pesar de que se han intentado minimizar con los filtros vistos anteriormente, siempre pueden encontrarse nuevas situaciones no contempladas o directamente ruido en la imagen obtenida desde el robot.
- Los *errores odométricos* en robots con patas suelen ser muy altos. En los simuladores estos errores pueden minimizarse, ya que suelen tratarse de errores sistemáticos que pueden corregirse. Sin embargo, en el robot real son más impredecibles y contendrán desviaciones.

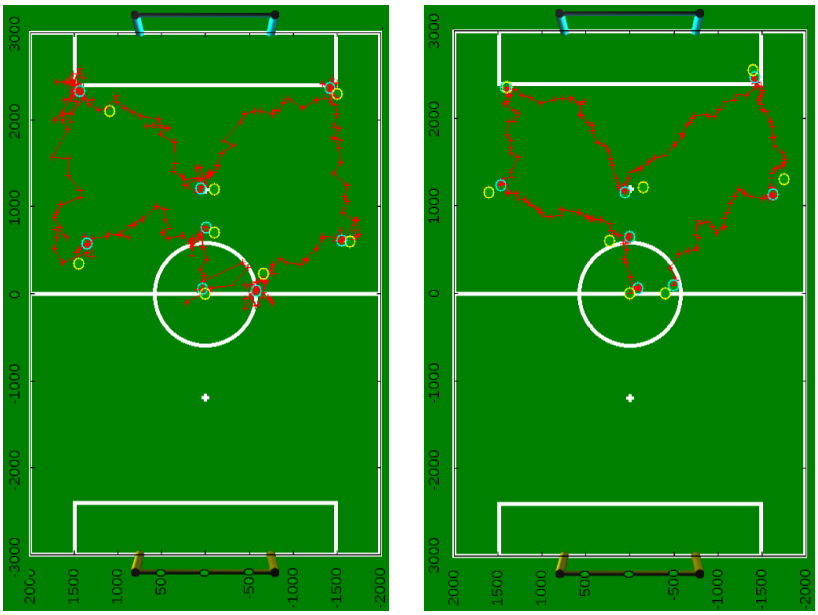
Como consecuencia de estos dos aspectos los resultados en el robot real son mucho más propensos a errores.

El primer experimento real consiste en medir la precisión de la localización al mover al robot por el campo. En este caso, los resultados muestran la trayectoria calculada por el robot (en rojo), la posición estimada del robot en los puntos de referencia (círculos azules) y la posición real en la que se encontraba (círculos amarillos).

Monte Carlo

El resultado de la ejecución típica del experimento con el algoritmo de Monte Carlo es el que se muestra en la Figura 5.17(a). Al igual que en el simulador, la trayectoria seguida por el robot sufre cambios bruscos, siendo el error medio en los 8 puntos seleccionados de 17.62 cm.

El coste computacional del algoritmo en el robot real ha sido de 27.7 ms, siendo el tiempo siempre bastante estable, no sobrepasando nunca los 40 ms.



(a) (b)

Figura 5.17: Desplazamiento con robot real con Monte Carlo (a) y algoritmo evolutivo (b).

Algoritmo Evolutivo

La Figura 5.17(b) muestra una ejecución representativa del algoritmo evolutivo con el experimento en el robot real. El error medio medido en los 8 puntos seleccionados ha sido de 14.36 cm, similar al resultado obtenido en el simulador.

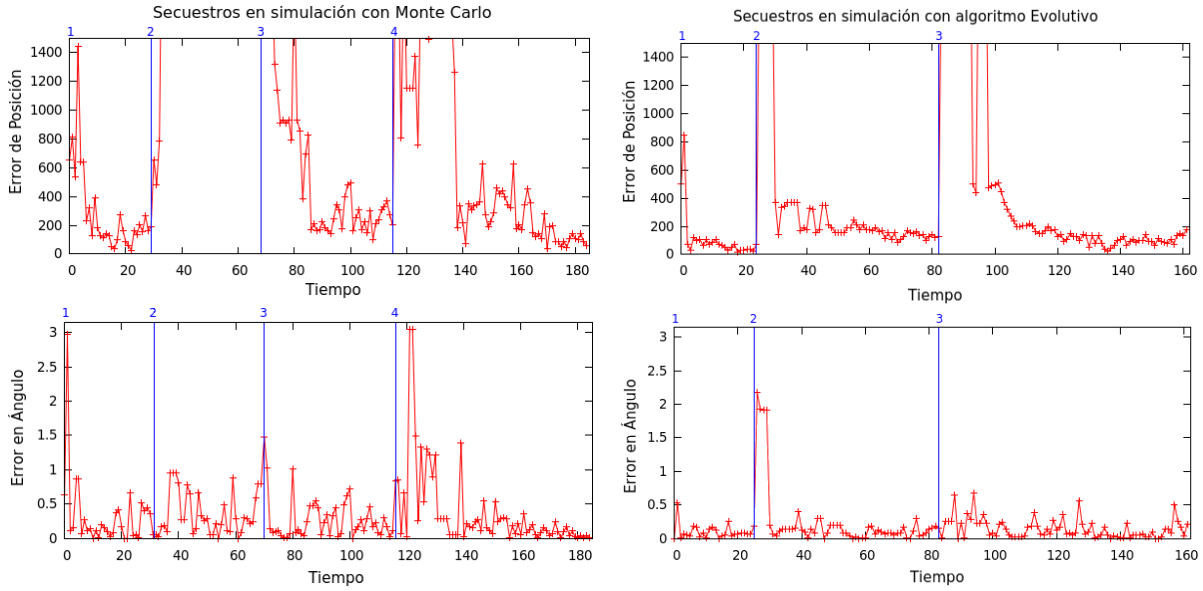
El coste computacional medio del algoritmo en el robot real ha sido de 37.6 milisegundos por iteración, aunque este tiempo cambia en función del número de razas y de si se lanzan exploradores en esa iteración o no, pudiendo ir desde 10 ms hasta 80 ms.

5.2.5. Robustez ante secuestros en simulación

El siguiente experimento trata de mostrar la robustez de los algoritmos frente a secuestros. El experimento representativo consiste en situar al robot en un lugar del campo hasta que se localice y, tras un tiempo en el que se está desplazando, se le traslada a otra parte del campo sin que su odometría lo perciba para ver la evolución en la estimación calculada. Con esto, se comprueba la capacidad que tienen los algoritmos para recuperarse ante situaciones de secuestro o de pérdida (cuando la estimación de posición no cuadra con la realidad).

Monte Carlo

Como se puede ver en la Figura 5.18(a), se han realizado cuatro secuestros distintos, obteniendo en tres de ellos una relocalización similar a la real, con un error medio en posición de 22.21 cm y en ángulo de 11.45 grados.



(a) (b)

Figura 5.18: Secuestros en simulador con Monte Carlo (a) y algoritmo evolutivo (b).

Sin embargo, en el segundo secuestro, la localización seleccionada tiene un error muy grande debido a las simetrías con las que cuenta el campo. En este caso, el algoritmo ha escogido una posición situada en el lateral izquierdo del campo cuando el robot se encontraba en el lateral derecho. Esto se produce porque en ambas posiciones se obtiene una información similar de las observaciones, una serie de píxeles blancos correspondientes a la línea central y una serie de píxeles azules correspondientes a uno de los postes de la portería. Si el algoritmo no recibe información que excluya alguna de las dos soluciones se encontrará ante una simetría en el campo.

Debido a la naturaleza del algoritmo, el método de Monte Carlo solo puede trabajar con una única solución de modo permanente. Esto implica que, en el caso de que existan simetrías, seleccionará una de las posibles soluciones, pudiendo no ser la correcta tal y como sucede en este caso.

El tiempo medio de recuperación entre el secuestro y la selección de una nueva localización fiable (hasta llegar a una fiabilidad mayor de 0.6) ha sido de 16.5 segundos.

Algoritmo Evolutivo

La Figura 5.18(b) muestra la respuesta del algoritmo de localización evolutivo frente a tres secuestros representativos, que ha sido satisfactoria, no quedándose en posiciones simétricas. El error medio en posición ha sido de 14.89 cm y en ángulo de 6.36 grados.

Como puede verse en los tres secuestros, la posición inicialmente seleccionada no ha sido totalmente correcta debido a las simetrías. Sin embargo, al contar con múltiples hipótesis, al transcurrir varias iteraciones y contar con más información, la raza seleccionada ha sido finalmente la correcta. Si se hubiese empleado un algoritmo que no soportase varias hipótesis (como el de Monte Carlo), la localización seleccionada podría haber sido la incorrecta, como sucedía en uno de los secuestros que se vio en la Figura 5.18(a).

En este caso, el tiempo de recuperación hasta llegar a una posición fiable (con fiabilidad mayor que 0.6) ha sido de 20.67 segundos de media. El tiempo de recuperación es mayor que en el algoritmo anterior debido a las indecisiones en la selección de la raza al comienzo de cada secuestro, pero, a cambio, el número de recuperaciones acertadas es mayor.

5.2.6. Robustez ante secuestros en el robot real

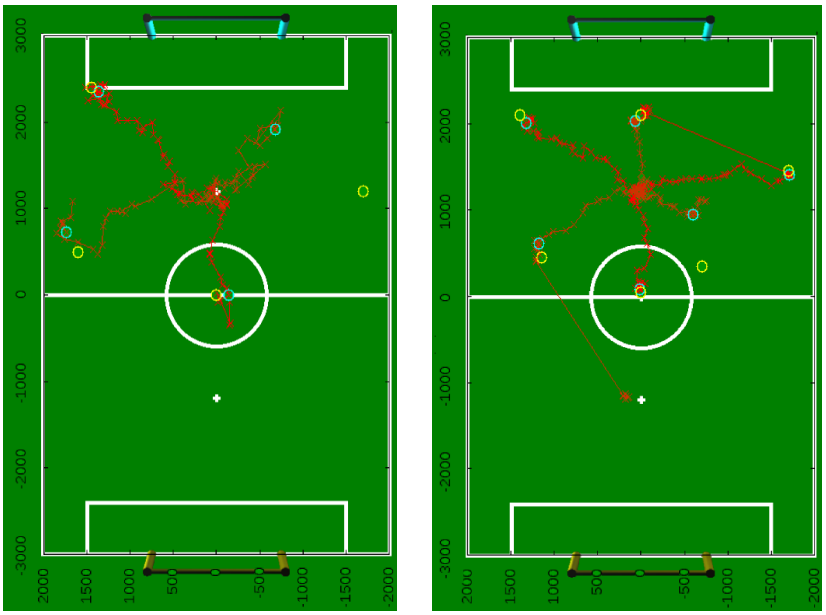
Siguiendo el mismo procedimiento que en el simulador, se han realizado experimentos con secuestros en el robot real. En los experimentos, se ha secuestrado al robot en repetidas ocasiones, ordenándole que se dirigiese al punto de penalti tras localizarse.

Monte Carlo

La Figura 5.19(a) muestra el resultado del experimento, en el que se han realizado 4 secuestros. En 3 de ellos el robot se ha recuperado con acierto. Sin embargo, en uno de ellos ha seleccionado otra posición del campo muy alejada de la realidad, debido a las simetrías. El error medio de las 3 posiciones bien localizadas es de 17.11 cm, mientras que si se tiene en cuenta también la localización errónea, el error medio producido es de 44.01 cm.

En general, el algoritmo cuenta con una gran precisión, pero su punto débil son las simetrías, que hacen que en ocasiones la localización seleccionada esté muy alejada de la posición real.

El tiempo de ejecución en el robot real en este caso ha sido de 27.7 ms, destacando además el hecho de que el tiempo es muy estable en todas las iteraciones, siendo siempre cercano al tiempo medio.



(a) (b)

Figura 5.19: Secuestros en robot real con Monte Carlo (a) y algoritmo evolutivo (b).

Algoritmo Evolutivo

La Figura 5.19(b) muestra los resultados del experimento. De los 6 secuestros realizados, 5 se han realizado con una precisión muy alta, mientras que en uno de ellos el error ha sido de 60.0 cm, siendo el error medio de 17.57 cm. El resultado final del algoritmo ha sido muy satisfactorio, ya que cuenta con una gran precisión y además es capaz de manejar simetrías, al estar especialmente diseñado para ello.

Los errores altos en la localización, como el del caso anterior, se producen porque la raza ganadora seleccionada no ha sido la correcta debido a simetrías o falsos positivos. Es previsible que si el robot siguiese desplazándose por el campo, se llegaría a un punto donde la simetría no se produjese y se cambiaría a la raza correcta.

El tiempo medio de ejecución en este caso ha sido de 37.6 ms; sin embargo, presenta como inconveniente una alta variabilidad, oscilando el tiempo entre los 10 y los 80 ms (cuando se lanzan exploradores).

5.3. Experimentos en entorno de oficinas

Además de los experimentos realizados en el entorno de la RoboCup, también se ha utilizado un escenario más realista donde los elementos presentes no están tan controlados.

Este escenario es el entorno de oficinas que se describió en la sección 3.5, que se compone de un pasillo con múltiples puertas, haciendo que las simetrías puedan ser más numerosas que en el mundo de la RoboCup. Asimismo, los cambios de luz que se producen a lo largo del pasillo hacen que los colores no puedan ser identificados tan fácilmente, por lo que los filtros de color utilizados deben ser menos excluyentes que en el caso de la RoboCup.

Primero se detallará cómo se ha realizado el análisis de la imagen y el cálculo del modelo de observación, para después pasar a los experimentos concretos sobre precisión y robustez.

5.3.1. Análisis de la imagen 2D: Píxeles informativos

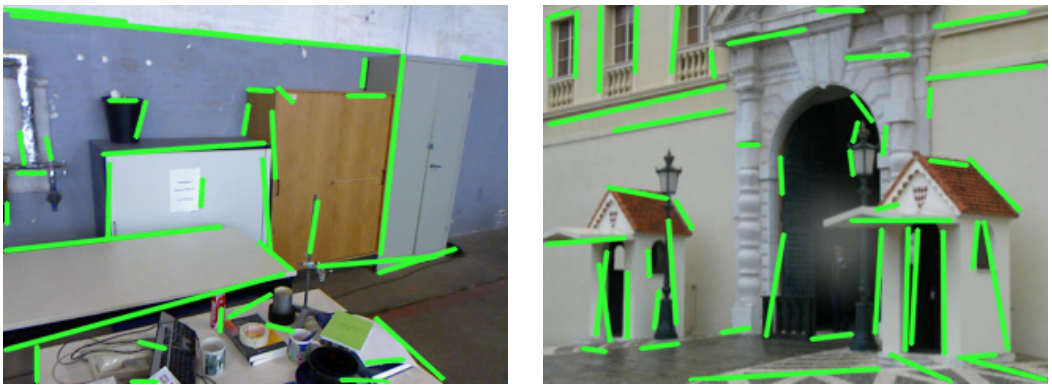
Uno de los elementos característicos de este entorno, al igual que sucede con muchos de los entornos creados por humanos, es que cuenta con una geometría uniforme. En un entorno de oficinas existirán puertas, paredes, mesas, etc, que formarán rectas, rectángulos y otros elementos geométricos que guardan proporcionalidad. Por ello, se realiza una búsqueda previa de rectas en la imagen para identificar con más claridad los elementos que se buscan. Se intentarán identificar siempre dos elementos: las puertas de cada uno de los despachos y los bordes del suelo con las paredes.

Los pasos a realizar, que serán detallados posteriormente, son los siguientes: detección de líneas, cálculo de horizonte y selección de píxeles interesantes.

Detección de líneas

La detección de líneas o segmentos en imágenes se caracteriza por su dificultad y por el gran tiempo de cómputo necesario para obtener una extracción de líneas fiable. Una de las primeras implementaciones para el análisis de segmentos en la imagen fue la combinación clásica de un filtro de bordes Canny ([Canny, 1986]) y la transformada de Hough ([Duda and Hart, 1971]). Está técnica ha sido ampliamente utilizada, pero el resultado alcanzado no era el adecuado, obteniendo un gran número de falsos negativos. Además, esta técnica era muy costosa computacionalmente.

En su lugar, se ha optado por utilizar el algoritmo desarrollado por [Solis *et al.*, 2009]. Este algoritmo utiliza los siguientes pasos para lograr detectar las líneas en la imagen: normalización de la imagen, suavizado Gaussiano, umbrales y detección de bordes de Laplace para extraer los contornos de la imagen de entrada. Este algoritmo ha resultado tener un comportamiento más fiable, robusto y rápido a la hora de detectar segmentos en la imagen que la transformada de Hough.



(a) (b)

Figura 5.20: Detección de líneas con algoritmo de Solis. Líneas detectadas en verde.

El algoritmo de Solis devuelve un conjunto de líneas en la imagen como el de la Figura 5.20. Algunas de las líneas obtenidas no serán válidas o serán demasiado pequeñas para ser tenidas en cuenta, por lo que es necesario realizar un procesado posterior para validar las líneas obtenidas.

Con las líneas obtenidas del paso anterior, primero se seleccionan aquellas que se corresponden con elementos de la imagen interesantes, es decir, con las puertas y los bordes del suelo. Para identificar estos elementos se ha realizado un etiquetado de cada línea en función de los colores que se encuentran a sus lados:

- Líneas de puertas: Estas líneas tendrán a uno de los lados el color rojo correspondiente a la puerta y en otro el color blanco correspondiente a la pared.
- Líneas de suelo: Este tipo de líneas tendrán el color blanco de la pared en un lado y el color amarillo del suelo en otro.
- Resto de líneas: Cualquier otro tipo de líneas pertenecerán a elementos desconocidos, que serán descartadas.

Una vez que etiquetada cada línea, se descartan aquellas que sean de tipo desconocido. Con las restantes, se procede a realizar una fusión de aquellas líneas que pertenecen al mismo tipo, con objeto de reducir el número de líneas encontradas. El resultado de la fusión de líneas puede verse en la Figura 5.21. Si tras la fusión existen líneas que son demasiado pequeñas según un umbral dado, se considera que se tratan de faltos positivos y se eliminan.

(a) (b)

Figura 5.21: Detección de líneas antes (a) y después (b) de fusionar líneas del mismo tipo.

Cálculo de horizonte

Debido a los colores similares que tienen la pared y el suelo, es posible que se produzcan falsos positivos en las paredes. Sabiendo que las líneas válidas estarán en el plano suelo y conociendo la geometría del robot, se pueden descartar todos aquellos píxeles que se encuentren a una determinada altura.

Así, se calcula la línea de horizonte utilizando geometría para descartar las líneas que se encuentren por encima, tal y como se aprecia en la Figura 5.22.

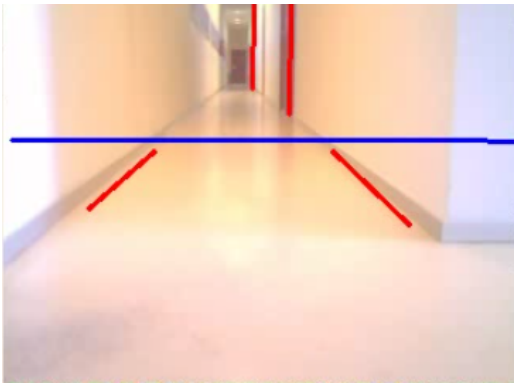
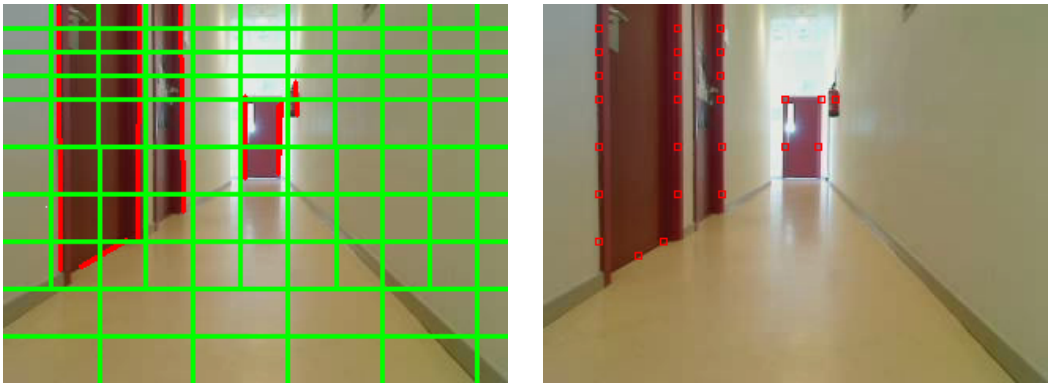


Figura 5.22: Línea de horizonte (en azul) calculada en el entorno de oficinas.

Selección de píxeles interesantes

Para facilitar el cálculo del modelo de observación, el análisis de imágenes debe devolver píxeles en lugar de líneas. Para ello, se ha generado una rejilla de tamaño dinámico, de forma que solo se tienen en cuenta los píxeles donde las líneas seleccionadas se cortan con

esta rejilla (Figura 5.23). De esta forma, se pasa de un conjunto de líneas a un conjunto de píxeles que representarán la imagen observada. El tamaño de las celdas de la rejilla es variable para analizar más en profundidad la parte superior de la imagen que la inferior, puesto que los objetos lejanos se encontrarán en esta zona de la imagen y su resolución será menor.



(a) (b)

Figura 5.23: Selección de píxeles interesantes desde líneas.

Calidad de la observación

Al igual que en el escenario de la RoboCup, es necesario valorar la calidad de la observación obtenida para poder estimar la fiabilidad de la localización.

En este caso, se ha estimado la calidad de la observación a partir del número de píxeles pertenecientes a las puertas. Así, se puede llegar a 1 en el caso de alcanzar el número máximo de píxeles permitidos para las puertas.

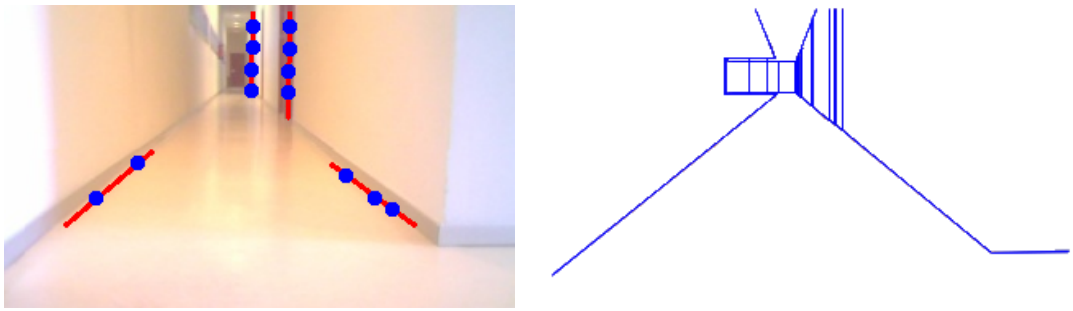
5.3.2. Modelo de observación sensorial

Tras analizar la imagen observada, es posible calcular la probabilidad de que un robot se encuentre en una posición determinada. Dada una posición (X, Y, θ), se comparan los datos obtenidos de la observación actual (imagen real) con lo que se debería ver si el robot estuviese en esa posición (imagen teórica). Si las imágenes son parecidas, la posición tendrá una alta probabilidad, mientras que si no se parecen la probabilidad será baja.

La técnica utilizada es parecida a la vista en el entorno de la RoboCup. Partiendo de los píxeles obtenidos en el análisis de observaciones, se calcula la distancia existente entre cada punto y el objeto del mismo tipo más cercano en la imagen teórica. Sin embargo, en

este caso no es viable precalcular las distancias entre puntos en 3 dimensiones puesto que el tamaño del mundo es demasiado grande.

Por ello, se calculan en tiempo de ejecución las rectas para generar la imagen teórica. En la Figura 5.24 se muestra un ejemplo de la imagen teórica generada en una posición y el mundo observado.



(a) (b)

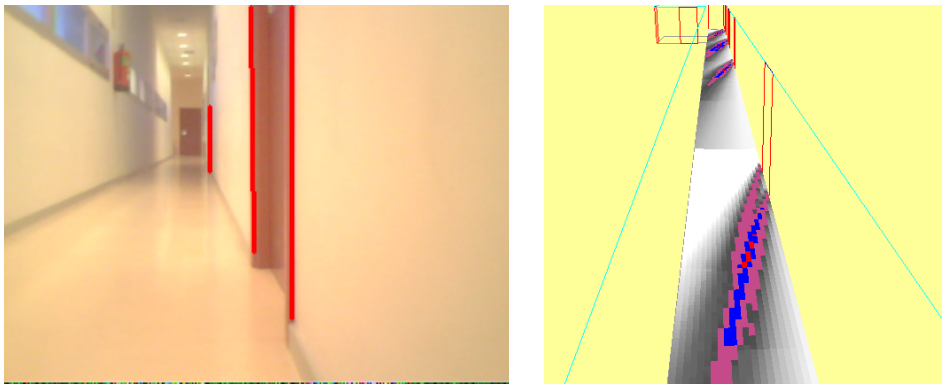
Figura 5.24: Comparativa entre la imagen real (a) y la imagen teórica generada (b).

Para ahorrar tiempo de cómputo, no se emplean todas las rectas del mundo para crear la imagen teórica, sino que solo se utilizan aquellas que se encuentran a una distancia cercana a la posición que se quiere evaluar. Para cada punto, se calcula la distancia entre el punto en 3D y cada una de las rectas de la imagen teórica del mismo tipo, seleccionando aquella más cercana.

Una vez calculada la distancia d_i a la recta más cercana para cada uno de los puntos, la probabilidad final para la posición (X, Y, θ) vendrá determinada por la media de las distancias, utilizando la ecuación 5.1 empleada en el escenario de la RoboCup.

Discriminación respecto a la posición

Para validar el modelo de observación, se ha implementado un mecanismo de depuración para mostrar gráficamente el valor calculado por la función en diferentes posiciones. En la Figura 5.25 se muestra el valor devuelto para todas las posiciones (X, Y) suponiendo θ constante. Como puede verse, las posiciones con mayor probabilidad (rojo) son plausibles con la imagen de entrada.

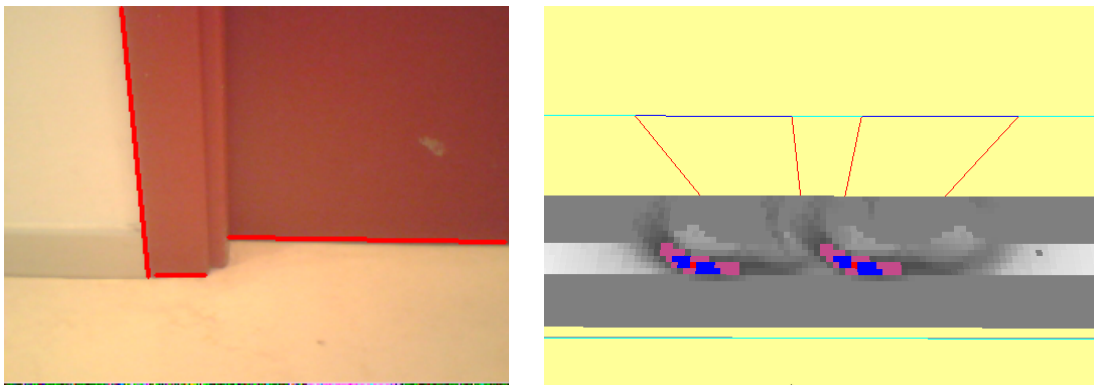


(a) (b)

Figura 5.25: Probabilidad obtenida con modelo de observación con θ igual a 0. Máxima probabilidad en rojo, seguido de azul, rosa, negro y por último blanco.

Efecto de las simetrías

Si existen simetrías, se obtendrá una alta probabilidad en varias posiciones del entorno. El entorno de oficinas está compuesto por múltiples puestas, de forma que si solo se observa una de ellas no es posible determinar frente a cuál se encuentra el robot, produciéndose por tanto una simetría con cada puerta del mundo (Figura 5.26).



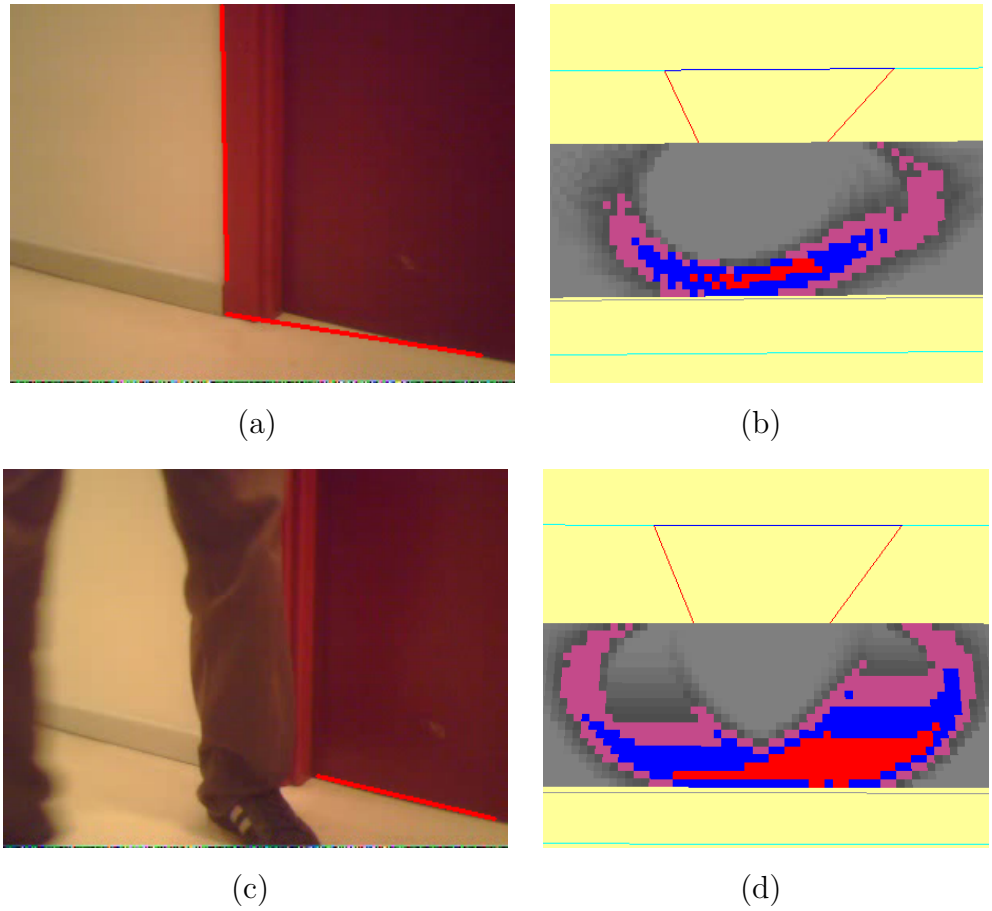
(a) (b)

Figura 5.26: Probabilidad calculada con modelo de observación para cualquier θ.

Puesto que el mundo está compuesto por múltiples puertas, no sería posible determinar frente a cuál se encuentra el robot y se produciría una simetría en cada puerta del mundo.

Efecto de las oclusiones

En caso de que se produzcan oclusiones, el modelo de observación no penaliza los elementos que no se ven en la imagen, por lo que el único inconveniente que se produce es que aumenta la probabilidad en algunas partes del mundo. Se muestra en la Figura 5.27 el efecto de las oclusiones cuando se producen frente a una de las puertas. Como se ve, el resultado es el previsto, aumentando las zonas con probabilidad alta (zona roja).



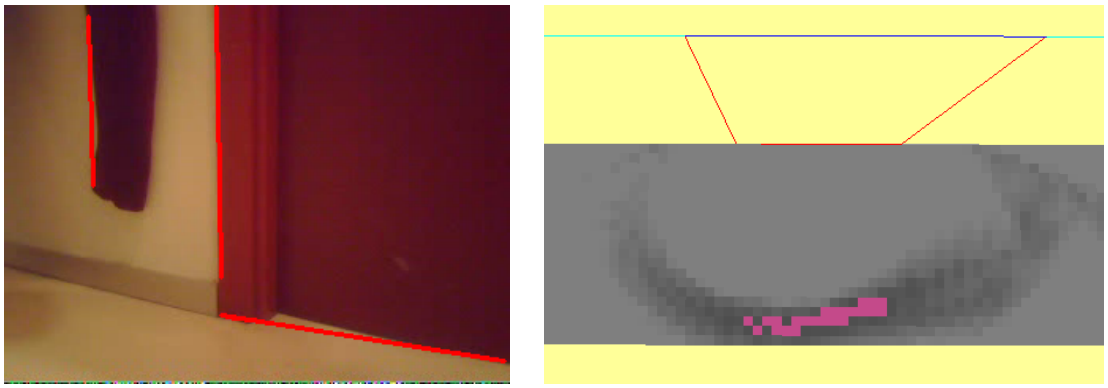
(a) (b)

(c) (d)

Figura 5.27: Efecto de oclusiones en el resultado del modelo de observación en entorno de oficinas. Resultado sin oclusiones (a,b) y con oclusiones (c,d) para cualquier θ.

Falsos positivos

Por último, se analiza el comportamiento del modelo de observación ante falsos positivos. Para ello, se ha añadido un nuevo elemento de un color similar al de las puertas, para ver qué sucedería si se detectase como falso positivo (Figura 5.28). En este caso, la función devuelve valores totalmente erróneos, perjudicando gravemente a la localización, por lo que deberán evitarse en la medida de lo posible este tipo de situaciones.



(a) (b)

Figura 5.28: Efecto de oclusiones en el resultado del modelo de observación en oficinas. Resultado obtenido para cualquier θ.

5.3.3. Precisión en el robot real

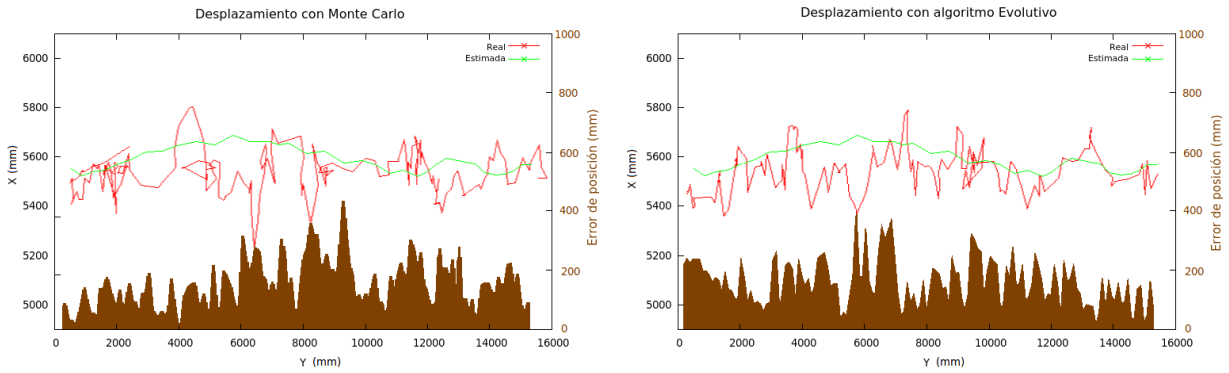
Los experimentos en el entorno de oficinas se han realizado en el escenario descrito en la sección 3.5, utilizando el robot *Nao* siempre en entornos reales. Al no poder contar con verdad absoluta en todo momento, se han realizado mediciones cada pocos centímetros de la posición (X, Y) del robot, que han servido para calcular el error respecto a la posición calculada por los algoritmos.

El primer experimento ha consistido en hacer que el robot *Nao* se desplace a lo largo de uno de los pasillos del entorno de oficinas. El movimiento seguido por el robot es en zigzag, recorriendo el pasillo de esta forma durante más de 15 metros.

Monte Carlo

Para el algoritmo de Monte Carlo la configuración utilizada ha sido de un número constante de 200 partículas. Inicialmente, las partículas han sido distribuidas en el mundo de forma aleatoria y en el resto de iteraciones han evolucionado utilizando un 10 % de partículas elitistas.

La Figura 5.29(a) muestra el resultado del experimento para el algoritmo de Monte Carlo. Puede verse cómo el algoritmo es capaz seguir correctamente el desplazamiento del robot, aunque se producen saltos en la estimación. El error medio del algoritmo ha sido de 15 cm.



(a) (b)

Figura 5.29: Desplazamiento con robot real en escenario de oficinas con Monte Carlo (a) y con algoritmo evolutivo (b). Trayectoria real (verde), calculada (rojo) y error resultante (marrón).

Algoritmo Evolutivo

El algoritmo evolutivo tiene un mayor número de parámetros de configuración. En este experimento se ha establecido el número de razas en 10, el número de explotadores por raza en 30 y el número de exploradores en 10. Además, los operadores genéticos utilizados han sido un 20 % de elitistas, un 20 % de individuos calculados mediante cruce y el resto mediante ruido térmico.

El número de razas debe ser alto en este experimento por el gran número de simetrías que puede haber, prácticamente una por cada puerta. Se ha seleccionado un valor con el cual la eficiencia del algoritmo y la precisión estuviesen equilibradas.

El resultado del experimento es el que se muestra en la Figura 5.29(b). De nuevo el algoritmo evolutivo es capaz de seguir la trayectoria del robot. En este caso la posición ha sido algo más estable que en el algoritmo de Monte Carlo y el error medio ha sido de 11.8 cm, menos que el obtenido con Monte Carlo.

5.3.4. Robustez ante secuestros en el robot real

El segundo experimento consiste en ver cómo se comportan los algoritmos ante simetrías y secuestros. Inicialmente, se ha situado al robot frente a una de las puertas, desde donde se ha desplazado para obtener más información de su entorno. Después, se han realizado dos secuestros siguiendo el procedimiento anterior.

Se expone el comportamiento de ambos algoritmos en la Figura 5.30, mostrando en azul el error del algoritmo de Monte Carlo y en verde el del algoritmo evolutivo. En el gráfico

se señalan también los momentos clave en los que se inicia el algoritmo (1.a) y cuando se realizan los secuestros (2.a y 3.a).

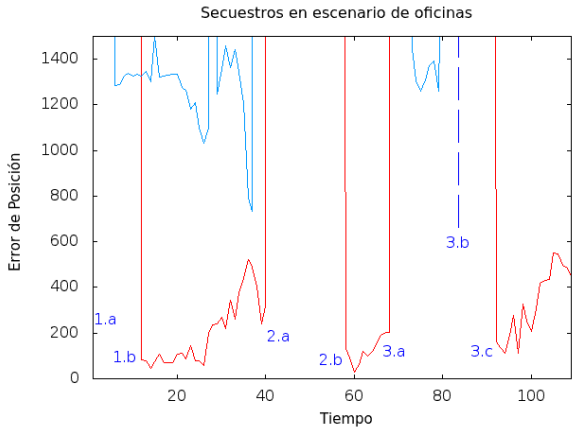


Figura 5.30: Error obtenido en secuestros con robot real en escenario de oficinas con Monte Carlo (azul) y algoritmo evolutivo (rojo).

Monte Carlo

Como puede verse en la Figura 5.30, el comportamiento del algoritmo de Monte Carlo ante este escenario con tantas simetrías es bastante pobre. Inicialmente (1.a) el algoritmo converge muy rápido hacia una posición situada relativamente cerca de la posición del robot, pero que no es del todo correcta. Al producirse los secuestros (2.a y 3.a) la reacción del algoritmo para cambiar su posición es rápida, pero selecciona en ambos casos localizaciones incorrectas debido a las simetrías.

El error medio del algoritmo en este experimento ha sido muy alto, puesto que en todos los casos se han elegido posiciones lejanas a la posición real, siendo su valor poco relevante.

También se ha medido el tiempo que tarda el algoritmo en cambiar de posición cuando se produce un secuestro (aunque sea errónea), situando este valor en tan solo 5 segundos. Esto indica que Monte Carlo cambia rápido de posición, aunque no siempre es recomendable.

Algoritmo Evolutivo

El comportamiento del algoritmo evolutivo ha sido muy distinto al de Monte Carlo. Inicialmente (1.a), se han creado varias razas frente a distintas puertas y la raza ganadora ha sido una de las que se encontraban en una posición incorrecta, de forma similar a lo sucedido en Monte Carlo. Sin embargo, al haber mantenido el resto de razas activas en otras

posiciones, cuando el robot ha obtenido más información del entorno ha podido recuperarse y ha seleccionado la posición correcta (1.b).

Al iniciarse el segundo secuestro (2.a), el algoritmo no cambia de posición rápidamente como sucedía con Monte Carlo, sino que tarda un tiempo considerable hasta que cambia a otra raza, que en este caso es la que está situada en la posición correcta (2.b). Esta tardanza en el cambio entre razas es debido a los mecanismos utilizados para seleccionar la raza ganadora, que evitan cambios bruscos entre razas.

En el tercer secuestro (3.a), se produce algo parecido a lo ocurrido inicialmente (1.a). Al principio se selecciona una localización incorrecta (3.b), pero tras obtener más información el algoritmo evolutivo cambia a la posición correcta (3.c).

El error medio del algoritmo, incluyendo el tiempo en el que la raza seleccionada no era la correcta, ha sido de 22,5 cm. En cuanto al tiempo de recuperación entre secuestros, el valor ha sido de 21 segundos, mayor que con Monte Carlo.

Con estos datos, se concluye que en escenarios complicados como el de oficinas, en el que el número de simetrías es muy grande, el algoritmo de Monte Carlo no tiene un comportamiento óptimo puesto que tiende a seleccionar posiciones similares pero incorrectas. El algoritmo evolutivo tiene en estos casos un comportamiento más favorable, al ser capaz de corregir su posición correctamente a lo largo del tiempo.

Capítulo 6

Localización visual con mapas desconocidos

En los capítulos anteriores se han descrito los algoritmos de localización visual para situar cámaras o robots en entornos con mapas conocidos. Este conocimiento a priori del escenario no es siempre posible y además cuenta con otras desventajas, como que es difícil adaptarlos a entornos que cambian dinámicamente.

En la sección 2.2 se presentó otro tipo de algoritmos llamados SLAM, que son capaces de calcular la posición de una cámara a la vez que se genera un mapa del entorno en el que se encuentra el robot. Estos algoritmos pueden utilizarse en la mayoría de entornos, sin necesidad de información a priori e incluso permiten que el entorno sea, hasta cierto punto, dinámico.

Tras haber estudiado y probado muchos de los algoritmos de localización visual en entornos desconocidos descritos en el capítulo 2 del estado del arte, se ha diseñado un algoritmo propio teniendo en cuenta los siguientes requisitos:

- No debe existir ningún tipo de información a priori sobre el entorno, por lo que el algoritmo será genérico y podrá ser utilizado en cualquier escenario. Esto significa que tiene que funcionar tanto en interiores como en exteriores.
- El entorno objetivo para el que está diseñado será principalmente estático, pero puede contener elementos dinámicos, como personas caminando u objetos desplazándose.
- El algoritmo debe funcionar en tiempo real en dispositivos con capacidad de cómputo limitada, como robots o dispositivos móviles.

El algoritmo que se ha desarrollado, implementado en *software* y validado experimentalmente se llama SDVL (siglas de *Semi-Direct Visual Localization*), y se ha inspirado en algunos de los algoritmos que se explicaron en el capítulo 2 del estado del arte, como MonoSLAM, PTAM, SVO o ORB-SLAM. En el capítulo 7 se comparará el algoritmo SDVL con las implementaciones originales de algunas de estas técnicas.

A lo largo de este capítulo se explican en detalle cada uno de los componentes del algoritmo, cotejándolos con los de otros algoritmos existentes.

6.1. Diseño general del algoritmo SDVL

El algoritmo SDVL sigue la filosofía de PTAM y divide en dos hilos independientes las dos tareas principales de los algoritmos de SLAM: por un lado calcula el desplazamiento de la cámara y por otro gestiona el mapa del entorno. Esta división se realiza puesto que el tiempo real solo es necesario para el seguimiento de la cámara en el entorno, mientras que el tiempo de cómputo para gestionar el mapa puede ser mayor sin perjudicar al resultado final del algoritmo.

En la Figura 6.1 se muestra el esquema general del algoritmo; éste tiene como entrada las imágenes de la cámara y como salida la estimación 3D a lo largo del tiempo.

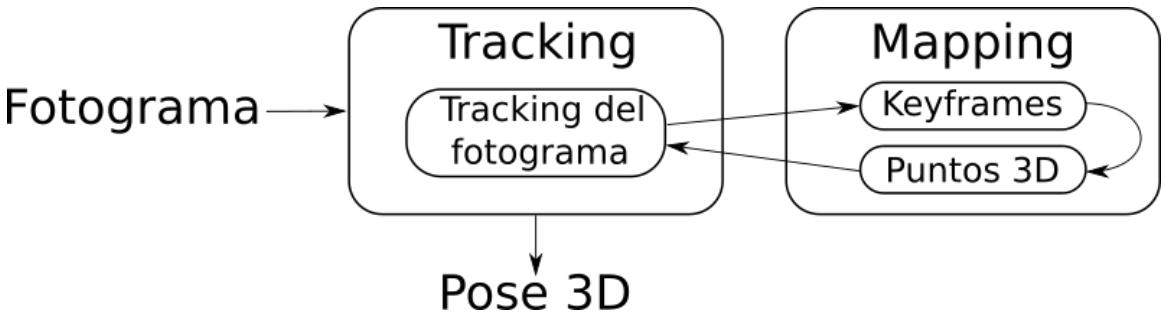


Figura 6.1: Diseño general del algoritmo SDVL.

El hilo de *Mapping* se encarga de crear, mantener y actualizar el mapa del entorno del robot. Para ello, se almacenan P puntos en 3D en K imágenes (observaciones) desde las que se observan estos puntos. No todas las observaciones pasan a formar parte del mapa, sino que solo los fotogramas que cumplan ciertas características se añaden al mapa (fotogramas claves o *Keyframes*).

El mapa inicial se genera partiendo de dos observaciones que tengan un desplazamiento suficiente entre ellas, que pasan a ser los dos primeros *Keyframes* del mapa. A partir de

entonces se añaden nuevos puntos en 3D en las zonas del mapa que no hayan sido exploradas anteriormente. Estos nuevos puntos en 3D no se calculan directamente por triangulación como sucedía en PTAM, sino que se inicializan teniendo en cuenta la incertidumbre en su profundidad y convergerán a medida que el robot se desplace y recoja nuevas observaciones.

El hilo de *Tracking* es el encargado de calcular el desplazamiento que ha sufrido la cámara entre dos observaciones utilizando el mapa 3D y las propias observaciones. Para calcular este desplazamiento se ha optado por utilizar una técnica híbrida basada tanto en píxeles característicos como en métodos directos.

Además, el sistema de *Tracking* cuenta con un sistema de rechazo de espurios (RdE) para eliminar puntos 3D mal posicionados o elementos en movimiento.

6.2. Fotogramas, píxeles de interés y puntos 3D

Se ha implementado el algoritmo SDVL en un programa en C++ con distintos componentes (clases) relacionados entre sí; es necesario conocer estas relaciones antes de explicar el algoritmo de forma detallada para comprender mejor su funcionamiento. En concreto se explican a continuación las relaciones que existen entre las observaciones (fotogramas), los píxeles en la imagen y los puntos 3D del mapa.

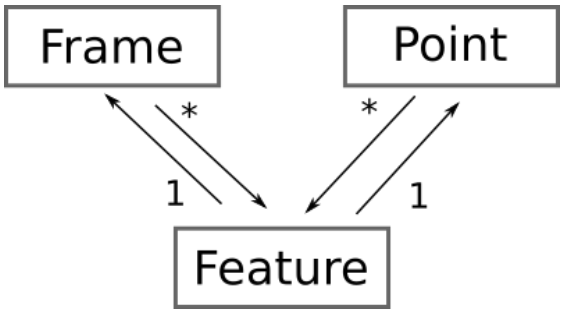


Figura 6.2: Referencias entre fotogramas (*Frames*), puntos 3D (*Points*) y píxeles 2D (*Features*) en SDVL.

6.2.1. Fotogramas

Un fotograma (*Frame*) representa una observación de la cámara, así como sus datos asociados. Cada fotograma cuenta con los siguientes elementos principales:

- Pose: Posición y orientación de la cámara cuando fue creado el fotograma.

- Pirámide de la imagen: Se almacena tanto la imagen original obtenida como subdivisiones de ésta hasta formar una pirámide de L niveles, siendo este último parámetro configurable.
- Píxeles característicos: Lista de píxeles de interés en cada nivel de la pirámide detectados mediante el algoritmo FAST, siguiendo el método descrito en la sección 6.4.
- Puntos 3D: Cuando un píxel característico se utiliza para crear un punto 3D se guarda en una lista de píxeles; a través de esta lista de píxeles se accede a los puntos 3D asociados siguiendo el esquema de la Figura 6.2.
- Fotogramas conectados: Cada *Keyframe* almacena una lista de referencias a otros *Keyframes* con los que comparte parte del campo de visión. Se considera que se comparte el campo de visión cuando varios puntos 3D han sido vistos desde ambos *Keyframes*.
- Los fotogramas cuentan también con un parámetro para saber si una observación ha sido elegida como *Keyframe*.

Así, cada vez que se recibe una nueva observación se instancia un objeto de la clase *Frame* que almacena toda la información descrita.

6.2.2. Píxeles de interés

Los píxeles en la imagen (*Features*) sirven para establecer una relación entre cada fotograma y cada punto 3D del mapa, tal como se ve en la Figura 6.2. La información que almacena cada píxel en la imagen es la siguiente:

- Fotograma: Observación a la que pertenece el píxel.
- Posición 2D: Coordenadas en píxeles de la imagen.
- Nivel: Nivel de la pirámide en el que se detectó, puesto que no tiene porqué haber sido detectado en la imagen original.
- Punto 3D: Punto 3D del mapa al que está asociado.
- Vector 3D: Vector unitario en 3D que representa el rayo de retroproyección del píxel en coordenadas relativas a la cámara.

6.2.3. Puntos 3D

Los puntos 3D (*Points*) del mapa no son almacenados directamente según sus coordenadas (X, Y, Z), sino que se ha utilizado una parametrización que permite tener en cuenta la incertidumbre en la profundidad de los puntos y que se detallará en la sección 6.3. Así, los parámetros principales de cada punto 3D son los siguientes:

- Píxel inicial: Píxel desde el que se creó el punto 3D la primera vez que fue detectado, utilizado para acceder a la observación inicial en la que se creó este punto siguiendo el esquema de la Figura 6.2.
- Píxeles: Lista con todos los píxeles en los que se ha detectado satisfactoriamente este punto en los distintos *Keyframes*.
- Proyecciones válidas: Número de veces que el punto 3D ha sido encontrado satisfactoriamente.
- Proyecciones inválidas: Número de veces que el punto 3D no ha sido encontrado al realizar el emparejamiento.
- Posicionamiento tridimensional del punto, parametrizado convenientemente (explicado en la sección 6.3).

6.3. Parametrización de puntos 3D

La parametrización más sencilla para representar un punto en 3 dimensiones es almacenar sus coordenadas (X, Y, Z) en el mundo. Esta parametrización es la utilizada por la mayoría de algoritmos de SLAM (como las primeras versiones de MonoSLAM o PTAM). Sin embargo, su defecto principal es que no permite representar cómodamente la incertidumbre en la posición del punto en el espacio, especialmente en su profundidad cuando se obtiene desde imágenes.

Dadas dos observaciones desde las cuales sea visible un mismo punto en 3D, es posible calcular la posición de este punto mediante triangulación siempre que el desplazamiento entre las observaciones respecto a la profundidad del punto sea suficiente. Dicho de otro modo, el ángulo que forma el punto 3D respecto a las dos observaciones (paralaje) debe ser suficientemente amplio para poder obtener una estimación de profundidad fiable.

Sin embargo, es posible que algunos puntos estén tan alejados que sea necesario realizar un gran desplazamiento para poder tener un paralaje suficiente o, en el peor de los casos, pueden existir objetos en el "infinito" para los cuales nunca se llegue a un paralaje suficiente (por ejemplo, al inicializar un punto con una estrella).

Todos estos puntos se perderían utilizando una parametrización clásica o incluso afectarían de manera negativa a la localización al realizar una estimación de la profundidad errónea. Por ello, diversos autores han propuesto otras representaciones de puntos en el mapa que tienen en cuenta la incertidumbre de su profundidad [Civera *et al.*, 2008] [Vogiatzis and Hernandez, 2011].

Para la implementación de SDVL se ha utilizado el método desarrollado por [Vogiatzis and Hernandez, 2011] con la adaptación propuesta por [Forster *et al.*, 2014] para representar puntos en el infinito. Así, la profundidad estimada de cada punto se modela mediante una distribución de probabilidad, utilizando cada nueva observación para actualizar la distribución mediante inferencia Bayesiana, de forma similar a como se vio en la Figura 2.10 del algoritmo SVO.

El método original toma como entrada un conjunto de medidas $x_1, \cdots x_n$ procedentes de un sensor de profundidad. Este sensor puede producir dos tipos de medidas: una medida válida (*inlier*), con probabilidad π normalmente distribuida alrededor de la profundidad real Z, y una medida errónea (*outlier*), con probabilidad $1-\pi$ resultante de una distribución uniforme en el intervalo $[Z_{min}, Z_{max}]$. Con estos datos, el siguiente modelo mixto *Gaussiano + Uniforme* describe la distribución de probabilidad de la medida x_n:

$$p(x_n, |Z, \pi) = \pi N(x_n|Z, \tau_n^2) + (1 - \pi)U(x_n|Z_{min}, Z_{max}) \tag{6.1}$$

siendo τ_n^2 la varianza de una medida válida, que puede ser calculada asumiendo una varianza de disparidad fotométrica de un píxel en el plano imagen.

La probabilidad a posteriori de este modelo se aproxima mediante el producto de una distribución Gaussiana para la profundidad Z y una distribución Beta para la probabilidad π:

$$q(Z, \pi|a_n, b_n, \mu_{Z_n}, \sigma_{Z_n}^2) := Beta(\pi|a_n, b_n)N(Z|\mu_{Z_n}, \sigma_{Z_n}^2) \tag{6.2}$$

donde a_n y b_n son contadores probabilísticos de cuántos *inliers* y *outliers* se han producido a lo largo de las observaciones, mientras que μ_{Z_n} y $\sigma_{Z_n}^2$ representan la media y la varianza de la Gaussiana de la profundidad estimada.

En la implementación del algoritmo SDVL, en lugar de utilizar directamente los valores de profundidad Z y σ_Z^2, se utiliza la inversa de la profundidad $\rho = \frac{1}{Z}$ y su varianza asociada σ_ρ^2. Esta parametrización inversa facilita la representación de puntos en el infinito, que se producirá cuando ρ sea 0.

Los valores iniciales de a y b se han fijado en 10, tal y como recomienda el artículo [Vogiatzis and Hernandez, 2011], mientras que ρ y σ_ρ toman de forma general el valor 1. Así, suponiendo una región de confianza del 95 %, la profundidad de cada punto se encontrará en el siguiente intervalo:

$$\left[\frac{1}{\rho - 2\sigma_\rho}, \frac{1}{\rho + 2\sigma_\rho}\right] \tag{6.3}$$

Con esta ecuación, la profundidad nunca podrá ser cero, pero se podrá representar el infinito cuando $\rho - 2\sigma_\rho = 0$. En concreto, para los valores iniciales dados, e ignorando los valores negativos que no son físicamente posibles, la profundidad del punto estará en el intervalo $[\frac{1}{3}, \infty]$.

6.3.1. Cálculo de la posición 3D

Desde la parametrización de puntos 3D con incertidumbre se puede realizar en todo momento una conversión para calcular la posición 3D cartesiana (X, Y, Z) en el espacio, aunque esta conversión podrá ser errónea cuando la incertidumbre sea alta.

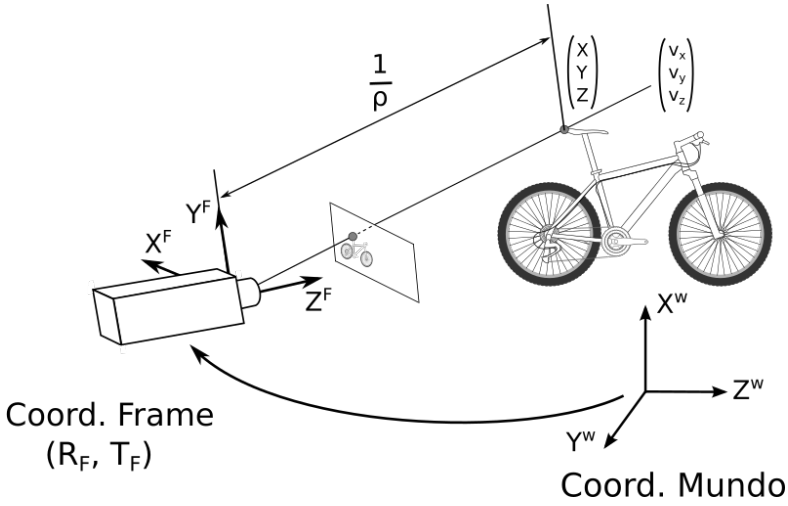


Figura 6.3: Parametrización de puntos 3D y conversión en coordenadas cartesianas.

Para calcular esta posición 3D, se utiliza la ecuación 6.4:

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = R_F \frac{1}{\rho} \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} + T_F \tag{6.4}$$

donde ρ es la inversa de la profundidad, T_F y R_F son la posición y orientación en el mundo del fotograma en el que se detectó el punto 3D y (v_x, v_y, v_z) es el rayo de retroproyección en coordenadas relativas a la cámara del píxel en la imagen donde fue detectado inicialmente el punto 3D (Figura 6.3).

6.4. Detección de píxeles de interés

Cada vez que se obtiene un nuevo fotograma se crean subdivisiones de la imagen original para generar una pirámide de L niveles. Analizar toda la información que existe en las imágenes no sería eficiente, por lo tanto, es necesario seleccionar un subconjunto de píxeles en cada uno de los niveles de la pirámide que sirvan como píxeles de interés para el resto de componentes del algoritmo.

Estos píxeles se han obtenido mediante el algoritmo de detección de esquinas FAST [Rosten and Drummond, 2006], algoritmo que destaca por su rapidez y repetibilidad. El umbral establecido para el algoritmo ha sido por defecto de 10, aunque este valor puede ser modificado en la configuración. En la Figura 6.4 se muestra un ejemplo de los píxeles de interés detectados con este umbral.

Figura 6.4: Píxeles seleccionados con FAST con umbral igual a 10 en secuencia F2D.

El algoritmo de FAST devuelve los píxeles detectados ordenados por filas y columnas, por lo que se calcula también una tabla que apunta al primer píxel detectado de cada fila de la imagen, lo que servirá para optimizar las búsquedas.

Los píxeles de interés se utilizan tanto en los emparejamientos entre imágenes como para inicializar nuevos puntos, como se verá a lo largo de este capítulo.

6.5. Emparejamiento de puntos entre imágenes

Uno de los elementos fundamentales de los algoritmos de SLAM basados en píxeles característicos es el emparejamiento de puntos entre dos observaciones. Dados dos fotogramas F_1 y F_2 y un punto X en el espacio 3D que es visible desde ambos fotogramas, el objetivo del emparejamiento es encontrar los pares de píxeles p_1 y p_2 en los que se observaría X desde cada uno de los fotogramas.

Si se conoce con exactitud la posición de los fotogramas y del punto 3D, basta con proyectar X en cada fotograma para obtener los píxeles p_1 y p_2. Sin embargo, debido a los errores en las observaciones y la incertidumbre en la posición 3D de X, es necesario realizar una búsqueda para tratar de encontrar el emparejamiento entre los píxeles.

En SDVL, un punto está siempre inicializado desde un fotograma F_1 y la posición 2D p_1 desde la que fue observado por primera vez se considera fija, por lo que la búsqueda se realiza en las imágenes de los fotogramas sucesivos (F_2) en los que X sea visible. Al inicializar un nuevo punto 3D, la profundidad no se conoce con exactitud, por lo que solo se puede inferir desde F_1 la retroproyección de p_1 que contendrá todos los puntos en el espacio $X_1, X_2, \cdots, X_n$ que proyecten en p_1.

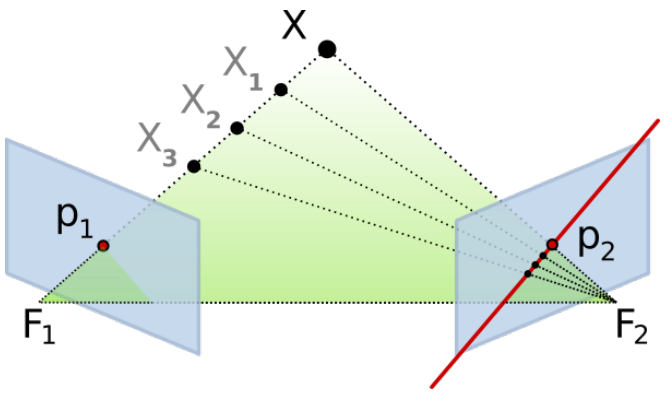


Figura 6.5: Recta epipolar obtenida al proyectar cada punto X_i en el plano imagen de F_2.

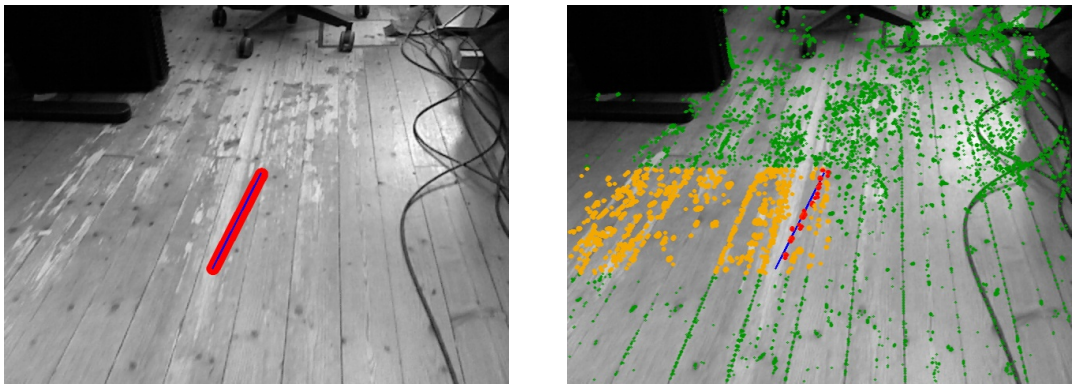
Algunos de estos puntos $X_1, X_2, \cdots, X_n$ serán visibles desde F_2 y la proyección de cada uno de esos puntos formará una línea recta en el plano imagen de F_2 que toma el nombre de recta epipolar (Figura 6.5).

Suponiendo una región de confianza del 95 % en la incertidumbre de la profundidad, la búsqueda solo se realiza en el tramo de la recta epipolar que se corresponda con una profundidad dentro del intervalo:

$$\left[\frac{1}{\rho - 2\sigma_\rho}, \frac{1}{\rho + 2\sigma_\rho}\right] \tag{6.5}$$

En caso de que $\rho - 2\sigma_\rho$ sea menor o igual que 0, se establecerá como valor mínimo un valor positivo suficientemente pequeño (por ejemplo 10^{-10}) para evitar divisiones por 0.

La búsqueda del punto 3D en la nueva observación se realiza a lo largo de la recta epipolar resultante suponiendo un margen de error alrededor de ésta (Figura 6.6(a)). El margen de error alrededor de la recta epipolar es configurable, pero su valor influirá en gran medida en el tiempo de cómputo final del algoritmo.



(a) (b)

Figura 6.6: Píxeles seleccionados para el emparejamiento de un punto 3D: Rango de búsqueda alrededor de la recta epipolar (a); píxeles característicos detectados (verde), comprobados (naranja) y utilizados en el emparejamiento (rojo) (b).

Dentro del rango de búsqueda calculado, se podría realizar una búsqueda exhaustiva comprobando cada píxel en su interior; sin embargo, el tiempo de cómputo sería muy alto y el algoritmo no funcionaría en tiempo real. Por ello, tan solo se utilizan en la práctica aquellos píxeles característicos detectados mediante el algoritmo de FAST.

Dado que los puntos obtenidos con el algoritmo FAST están ordenados por filas y columnas, no es necesario comprobar todos los puntos detectados, sino que es suficiente con un subconjunto de éstos. En la Figura 6.6(b) se muestra cómo, a partir de todos los puntos FAST detectados en la imagen (color verde), solo es necesario comprobar la posición en la imagen de aquellos puntos que comparten fila y columna con la recta epipolar (naranja).

De estos puntos, solo los puntos en rojo están situados dentro del rango de búsqueda y son los empleados en el emparejamiento.

Así pues, la comparación se realiza entre p_1 y todos aquellos puntos seleccionados en F_2. Para ello, primero se obtiene en F_1 un parche alrededor del p_1 con un ancho que por defecto se ha establecido en 8 píxeles. Además, se modifica el parche realizando una transformación afín [Berger, 1987] para tener en cuenta el desplazamiento realizado entre F_1 y F_2.

El parche obtenido se compara con un parche del mismo tamaño alrededor de cada punto candidato en F_2, evaluando cada emparejamiento mediante el algoritmo ZMSSD (*Zero-Mean Sum of Squared Differences*) [Martin and Crowley, 1995] y se selecciona aquel emparejamiento que tenga menor puntuación (lo que indicará que la diferencia entre parches es menor). El punto seleccionado será válido siempre que su valor sea menor que un umbral, considerando la búsqueda como fallida si resulta ser superior.

El emparejamiento de puntos no se realiza únicamente con la imagen original, sino que también se realiza con puntos que hayan sido detectados en otros niveles de la pirámide, adaptando en ese caso los márgenes de error al tamaño de la imagen.

En algunos casos, especialmente cuando el emparejamiento se efectúa en niveles altos de la pirámide (alejados de la imagen original), no es suficiente con obtener un píxel en la imagen como resultado del emparejamiento, sino que es necesario obtener un valor con precisión de subpíxel. Así, el píxel seleccionado mediante el algoritmo ZMSS se somete a un refinamiento a nivel de subpíxel utilizando el algoritmo de componente inversa [Baker and Matthews, 2004], que minimiza de forma iterativa el error fotométrico entre los parches.

En la Figura 6.7 se muestran los emparejamientos realizados correctamente entre dos imágenes consecutivas.

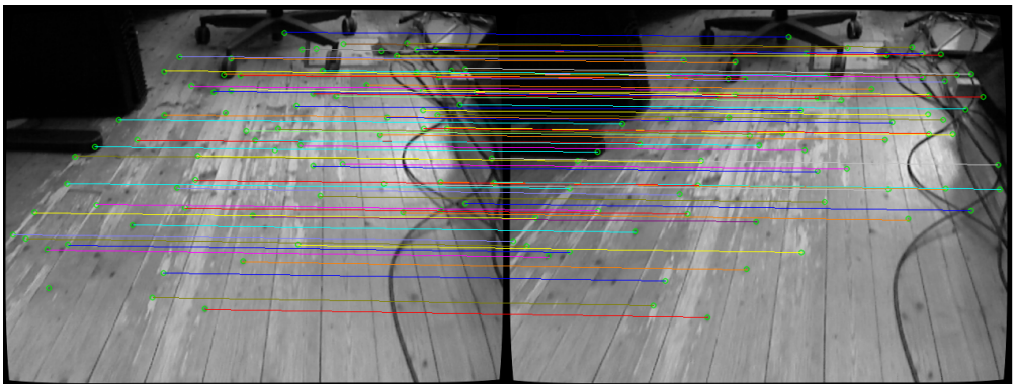


Figura 6.7: Emparejamiento de puntos mediante parches entre dos imágenes consecutivas.

6.5.1. Descriptores ORB

Como alternativa al proceso que se acaba de detallar, el algoritmo SDVL permite utilizar descriptores ORB [Rublee *et al.*, 2011] para realizar el emparejamiento de puntos, en lugar de comparar parches entre imágenes. ORB ha supuesto una alternativa eficiente y libre a los descriptores clásicos SIFT o SURF, utilizando puntos FAST orientados con un descriptor asociado de 256 bits.

La extracción de descriptores ORB es más costosa que la obtención de parches, pero proporciona más fiabilidad en los emparejamientos. Para evitar que el tiempo de cómputo se incremente demasiado, solo se calculan los descriptores ORB de aquellos puntos FAST que se van a emparejar y se almacenan en memoria para evitar que se calcule más de una vez el descriptor ORB del mismo punto. El tamaño del parche que se ha empleado para calcular los descriptores ORB ha sido de 31 píxeles.

Una vez que se han obtenidos los descriptores ORB de dos píxeles en dos imágenes, estos se comparan calculando la distancia de Hamming [Hamming, 1950] de ambos descriptores, validando los emparejamientos cuya distancia sea menor que un umbral; este umbral se ha establecido experimentalmente en 100.

6.6. *Mapping*

El hilo de *Mapping* es el encargado de gestionar el mapa del entorno en el que se encuentra el robot. Este mapa se actualiza con cada observación y sirve de base para el cálculo del seguimiento de la cámara.

En caso de que se almacenasen en el mapa todas las observaciones obtenidas, se generaría una lista de fotogramas con mucha información repetida y cuyo tratamiento sería muy poco eficiente. Por ello, el mapa solo almacena un subconjunto de observaciones (*Keyframes*) que se consideran suficientes para representar el entorno.

Para determinar cuándo un fotograma se almacena en el mapa como *Keyframe* se comprueban dos condiciones:

1. Número de iteraciones transcurridas: En caso de que se cree un *Keyframe* habrá que esperar un número de iteraciones predeterminado antes de poder crear otro. Por defecto este valor se ha establecido en 30 iteraciones (1 por segundo).

2. Número de puntos perdidos: Se compara el número de puntos 3D visibles respecto a los que se veían cuando se creó el último *Keyframe*. En caso de que este número sea menor significa que se han perdido puntos y se procederá a crear un nuevo *Keyframe* cuando se pierdan al menos un 10 % de los puntos.

Para crear un nuevo *Keyframe* deben cumplirse simultáneamente estas dos condiciones. Sin embargo, cuando se producen desplazamientos muy pronunciados (especialmente en rotaciones), los puntos se pueden perder rápidamente y este procedimiento puede llevar a que el robot se pierda. Por ello, cuando el número de puntos perdidos desde el último *Keyframe* sea mayor que el 30 %, se creará un nuevo *Keyframe* incluso aunque no hayan transcurrido suficientes iteraciones desde el último.

El número de *Keyframes* a almacenar también puede limitarse para que el *Mapping* sea más eficiente. En caso alcanzar el número máximo de *Keyframes*, se elimina del mapa aquel que se encuentre más alejado de la posición actual, ya que es menos probable que se regrese a una posición más alejada que a otras posiciones cercanas a la actual.

Los puntos 3D de los que se compone el mapa son aquellos que han sido detectados por uno o varios de los *Keyframes* actuales y se accede a su posición a través de cada *Keyframe* siguiendo el esquema de la Figura 6.2.

Además, el mapa también cuenta con una lista de *puntos candidatos*, que son aquellos puntos 3D que todavía tienen una incertidumbre en su profundidad demasiado alta y por tanto no han convergido. El hilo de *Mapping* se encarga de actualizar estos candidatos con cada nuevo fotograma obtenido hasta que converjan.

6.6.1. Inicialización del mapa

Para inicializar el mapa, el algoritmo necesita obtener dos observaciones que compartan una parte del mundo en su campo de visión y que exista un desplazamiento suficiente entre ellas. El mecanismo desarrollado es automático y no necesita por tanto intervención externa.

Primero, se obtienen los píxeles característicos de cada nueva observación mediante el algoritmo FAST. Para poder utilizar la observación se requiere que al menos se detecten 50 puntos en la imagen, aunque normalmente el algoritmo suele obtener más de 300.

La primera observación que cumpla esta condición se almacena como el primer *Keyframe* del algoritmo y sirve como base para generar el mapa inicial. En las observaciones sucesivas,

se calcula el desplazamiento de cada punto detectado en el primer *Keyframe* mediante flujo óptico de Lucas-Kanade [Lucas and Kanade, 1981].

Este procedimiento se repite hasta que la distancia media de todos los puntos detectados supere un umbral, momento en el cual se considera que existe un desplazamiento suficiente entre las imágenes como para calcular el mapa inicial y se almacena la última observación como el segundo *Keyframe* del mapa.

En el caso de que se pierda un gran número de puntos durante el seguimiento de puntos por flujo óptico, se entiende que la inicialización ha sido fallida y se reinicia el proceso utilizando la última observación como primer *Keyframe*.

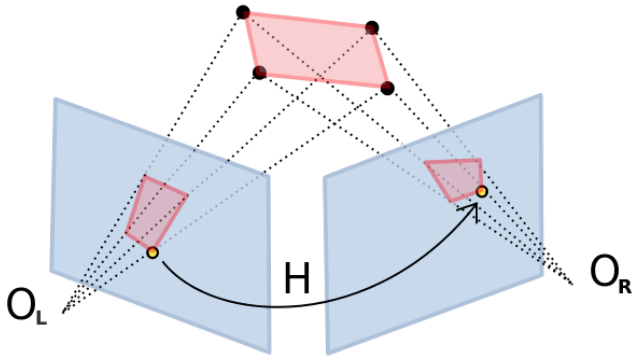


Figura 6.8: Homografía entre dos imágenes que observan el mismo plano.

Una vez que se dispone de los dos primeros *Keyframes*, se calcula el mapa inicial utilizando el algoritmo de 5 puntos [Stewenius *et al.*, 2006]. Suponiendo que ambas observaciones contienen un plano dominante y utilizando el algoritmo RANSAC [Fischler and Bolles, 1981], se obtiene la homografía H que determina la transformación de perspectiva entre ambas imágenes (Figura 6.8), de forma que el error de reproyección de cada par de puntos se minimice (ecuación 6.6):

$$\begin{pmatrix} x'_i \\ y'_i \\ 1 \end{pmatrix} \sim H \begin{pmatrix} x_i \\ y_i \\ 1 \end{pmatrix} \tag{6.6}$$

siendo (x_i, y_i) las coordenadas en píxeles de cada punto en el primer *Keyframe*, (x'_i, y'_i) las coordenadas equivalentes en el segundo *Keyframe* y H la matriz de homografía de 3x3 que mejor aproxima cada par de puntos.

Una vez obtenida la homografía que mejor aproxima los puntos en 2D entre las dos observaciones, se descompone H en sus valores singulares [Faugeras and F., 1988]. Así, se

obtiene la matriz de rotación R, el vector de traslación t y el vector normal n.

La descomposición en valores singulares devuelve hasta 8 soluciones matemáticamente posibles para (R, t, n), de las cuales solo 2 cumplirán las restricciones físicas del problema. Para elegir la mejor solución de entre esas dos, se calcula cuál de ellas tiene un menor error geométrico entre los píxeles dados utilizando el error de Sampson [Sampson, 1982].

Una vez seleccionada la mejor solución, se utilizan los valores de R y t para establecer el desplazamiento entre los dos primeros *Keyframes* y para calcular la posición de los puntos del mapa inicial mediante triangulación. Tras calcular la posición de todos los puntos se reescala el mapa para que la profundidad media de los puntos sea la deseada (típicamente 1.0); el mapa generado sería perfectamente compatible con cualquier otra escala, ya que con una sola cámara no hay referencia de profundidad exacta.

Con este procedimiento se obtiene un mapa con dos *Keyframes* y F puntos en 3D visibles desde estos dos *Keyframes*, lo que sirve como punto de partida para el *Tracking* y para ampliar el mapa progresivamente.

6.6.2. Inicialización de puntos 3D

Una vez calculado el mapa inicial, éste se actualiza y se amplia a medida que el robot se desplaza por zonas del entorno que no hayan sido exploradas previamente.

En algoritmos como PTAM, se crean nuevos puntos en 3D mediante triangulación a partir de dos observaciones del mismo punto en dos *Keyframes* distintos. Esto cuenta con la ventaja de que genera nuevos puntos de forma muy rápida y eficiente, pero genera demasiados puntos espurios por dos razones:

- El punto solo ha sido observado desde dos fotogramas y puede que se hayan producido errores en el emparejamiento, por lo que la posición 3D calculada contendrá errores y el punto será difícil de encontrar en el resto de observaciones.
- El punto puede estar demasiado lejos de la escena actual y no contar con paralaje suficiente como para realizar la triangulación con garantías, siendo la profundidad final del punto poco realista.

Como se ha explicado, en SDVL se utilizan puntos 3D con incertidumbre en la profundidad, por lo que cada punto es observado desde múltiples observaciones antes de converger y calcular su posición final. Esto asegura que las posiciones finales calculadas sean más fiables y existan menos puntos espurios.

El algoritmo de inicialización de puntos solo se pone en marcha en aquellos fotogramas que hayan sido catalogados como *Keyframes*. Primero se seleccionan los píxeles de la imagen que pasarán a ser candidatos, sabiendo que es necesario explorar las zonas de la imagen que no hayan sido previamente analizadas. Para ello, se divide la imagen en celdas que inicialmente aparecen como no exploradas. Las celdas que ya cuentan con puntos conocidos (que habrán sido detectadas durante el *Tracking*) quedan descartadas, mientras que en las restantes se crea como máximo un nuevo candidato por celda. El tamaño de cada celda es configurable, pero debe mantener un equilibrio para que el algoritmo sea eficiente y el mapa suficientemente denso.

Para determinar qué puntos son inicializados, se obtienen primero los píxeles característicos de cada celda no explorada mediante FAST (en cada nivel de la pirámide de la imagen) y entre ellos se selecciona el que tenga una mayor puntuación según el algoritmo de Shi-Tomasi [Shi, 1994].

Para inicializar cada punto se necesita una lista de *Keyframes* que compartan el campo visual con el *Keyframe* actual, que son almacenados generando un grafo de *Keyframes* conectados por su campo de visión. Para decidir qué *Keyframes* se almacenan, se utilizan los puntos en la imagen que se detectaron en este fotograma durante el *Tracking* y se contabilizan los puntos 3D visibles desde el fotograma actual que también son visibles desde el resto de *Keyframes*. Solo se añaden al grafo los *Keyframes* que compartan varios puntos 3D con el fotograma actual (por defecto 10).

Los puntos candidatos para ser inicializados se buscan en la lista de *Keyframes* conectados, siendo solo necesario que se encuentre en uno de los *Keyframes*. El emparejamiento de puntos se realiza con el método explicado en la sección 6.5 y, en caso de que el emparejamiento sea válido en alguno de los *Keyframes* conectados, se triangula la posición 3D para obtener la profundidad inicial del punto 3D. Como ya se ha comentado, esta profundidad inicial no tiene porqué ser la profundidad real, por lo que se añaden estos puntos a la lista de candidatos del mapa y su profundidad real se calculará a medida que se obtengan nuevas observaciones.

6.6.3. Actualización de la incertidumbre en profundidad

A medida que se obtienen nuevas observaciones, el valor de los parámetros de incertidumbre de cada punto se modifica en función del desplazamiento de la cámara y de la profundidad real del punto.

Para los puntos más cercanos a la cámara el desplazamiento hará que se alcance un

paralaje suficiente, disminuyendo así la incertidumbre en la profundidad y convergiendo el valor de la profundidad inversa ρ hacia su posición real. La fase de actualización de cada punto también es costosa y no merece la pena seguir realizándola una vez que la incertidumbre sea suficientemente pequeña. Para determinar cuándo un punto ha convergido en su posición 3D real y, por tanto, ya no necesita seguir siendo actualizado, se utiliza la condición de convergencia propuesta en [Civera *et al.*, 2008], calculando el índice de linealidad L mediante las ecuaciones 6.7 y 6.8:

$$\sigma_d = \frac{\sigma_\rho}{\rho^2} \tag{6.7}$$

$$L = \frac{4\sigma_d}{d}|cos\alpha| \tag{6.8}$$

siendo d la distancia desde el punto en 3D a la posición actual de la cámara y α el paralaje del punto 3D con respecto a la primera y última observación.

Con esta ecuación, cuando el paralaje sea bajo, la desviación típica σ_ρ será alta y el $|cos\alpha|$ será cercano a uno, obteniendo un valor de L alto. Mientras que a medida que el paralaje aumente, tanto σ_ρ como $|cos\alpha|$ disminuirán y como consecuencia el valor de L será bajo. Se considera que un punto ha convergido cuando el valor de L sea menor que 0.1, valor que ha sido fijado experimentalmente.

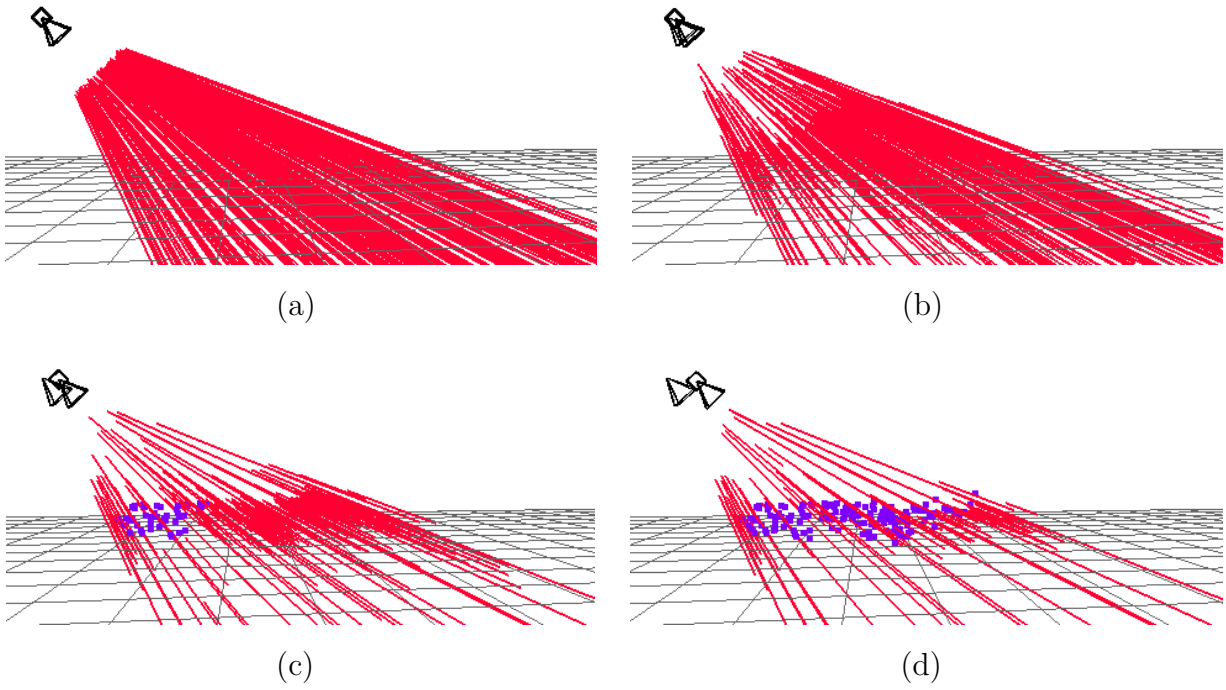


Figura 6.9: Convergencia de puntos 3D tras 1 (a) 5 (b), 10 (c) y 15 (d) iteraciones.

En la Figura 6.9 se muestra un ejemplo de cómo los puntos 3D convergen con el paso de las iteraciones. Inicialmente, todos los puntos inicializados tienen una alta incertidumbre y su profundidad es por tanto desconocida (rectas rojas), pero una vez que el desplazamiento de la cámara es suficiente, algunos de los puntos cumplen la condición de convergencia y fijan su profundidad (puntos azules).

Cada vez que se recibe un nuevo fotograma (sea o no *Keyframe*), se recorre la lista de candidatos del mapa y se intentan actualizar su posiciones. Primero, se comprueba si el punto es visible desde la posición actual de la cámara, en cuyo caso se procede a buscar el punto según el método visto en la sección 6.5. Si el punto se encuentra, se triangula su posición en el espacio utilizando el fotograma actual y el *Keyframe* original en el que se detectó por primera vez este candidato. Esto da lugar a una nueva profundidad que sirve para actualizar la distribución de probabilidad del punto 3D. Además, se comprueba si el punto ha convergido para proceder en ese caso a su eliminación de la lista de candidatos.

En caso de que el punto no se encuentre, se incrementa su contador de número de proyecciones inválidas, lo que puede llevar a su eliminación completa si no se encuentra durante varios fotogramas consecutivos.

6.6.4. Optimización del mapa

De forma opcional, el algoritmo permite realizar una optimización del mapa mediante el algoritmo de ajuste de haces (*Bundle Adjustment*, BA). Este algoritmo fue por primera vez utilizado en algoritmos de SLAM en tiempo real en el algoritmo de PTAM. Como ya se explicó en el capítulo 2 del estado del arte, el objetivo de este algoritmo es minimizar el error de reproyección de N puntos 3D que son visibles desde M posiciones de la cámara.

Dado un punto 3D X_i, una posición de la cámara x_j y el píxel en la imagen donde fue emparejado inicialmente $p_{i,j}$, el error de reproyección $e_{i,j}$ se define mediante la ecuación 6.9:

$$e_{i,j} = x_{i,j} - f_p(x_j, X_i) \tag{6.9}$$

siendo $f_p(x_j, X_i)$ la función de proyección del punto X_i en la imagen desde x_j.

La minimización por mínimos cuadrados del error de reproyección en el algoritmo de *Bundle Adjustment* se realiza de forma iterativa mediante el algoritmo de Levenberg-Marquardt [Nocedal and Wright, 2000] [Hartley and Zisserman, 2003]. En concreto, en

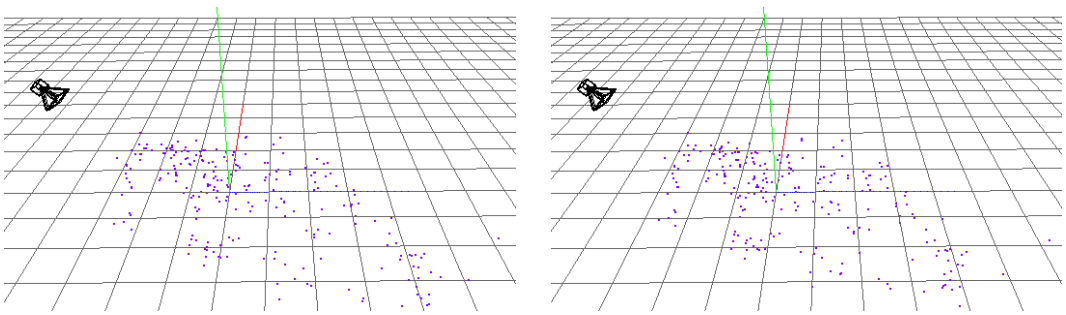
el algoritmo SDVL se ha utilizado la implementación de *Bundle Adjustment* de la librería *g2o* [Kümmerle *et al.*, 2011].

El BA en el algoritmo SDVL es opcional, puesto que incrementa el tiempo de cómputo del *Mapping* y, dependiendo de la capacidad de cómputo del dispositivo que se utilice, esto podría llevar a que se dejasen de procesar fotogramas en el *Mapping* cuando el algoritmo tardase demasiado.

En caso de que se utilice el BA, se realiza la optimización después de generar el mapa inicial y cada vez que se añada un nuevo *Keyframe*. Para determinar qué *Keyframes* y puntos 3D se optimizan, se seleccionan los *Keyframes* conectados al *Keyframe* actual y se utilizan todos los puntos 3D que sean visibles desde estos *Keyframes* que ya hayan convergido en una posición final. No se tienen en cuenta los puntos con una incertidumbre de la profundidad alta porque la posición 3D estimada de estos puntos puede afectar al resultado del BA si la profundidad es errónea.

Además, habrá otros *Keyframes* que también observen a los puntos elegidos pero que no hayan sido seleccionados todavía. Estos *Keyframes* también se añaden al algoritmo de BA, pero su posición se establece como fija y el algoritmo no puede modificarla.

El algoritmo de BA se ejecuta en dos fases. Primero se minimiza el error de reproyección realizando 5 iteraciones del algoritmo de Levenberg-Marquardt con los puntos y *Keyframes* seleccionados. Tras estas 5 iteraciones, se descartan todos los puntos cuyo error ha sido demasiado alto, es decir, que estén fuera del intervalo de confianza del 95 % de una distribución χ^2 con dos grados de libertad. Tras descartar estos puntos, se realizan otras 10 iteraciones del algoritmo con los puntos restantes. Por último, se actualizan las posiciones de los puntos 3D y de los *Keyframes* (no fijos) con los valores calculados por el algoritmo.



(a) (b)

Figura 6.10: Mapa generado antes (a) y después (b) de ser optimizado con BA.

En la Figura 6.10 se muestra un ejemplo de cómo se modifica ligeramente el mapa cuando se optimiza mediante BA.

6.7. *Tracking*

El otro hilo de ejecución fundamental del algoritmo SDVL es el encargado del *Tracking*. Este hilo utiliza la información almacenada en el mapa y las observaciones de la cámara para calcular el desplazamiento del robot desde la última iteración, es decir, incrementalmente. Además, deberá realizar esta tarea cumpliendo las restricciones de eficiencia temporal para funcionar en tiempo real.

Partiendo de la posición y orientación de la cámara en el instante $t-1$ (x_{t-1}), el cálculo de la posición final en el instante t (x_t) se realiza en cuatro pasos diferenciados:

1. Suponiendo un modelo de velocidad constante, se inicializa la posición del fotograma actual x_{t_i} teniendo en cuenta la posición del fotograma anterior x_{t-1} y la velocidad a la que se desplazaba la cámara en la última iteración.

2. Se alinea la imagen actual con la imagen del fotograma anterior utilizando *métodos directos* para obtener una primera posición estimada del fotograma actual x_{t_e}.

3. Se proyectan los puntos disponibles en el mapa 3D y se realiza un emparejamiento de algunos de estos puntos en la imagen actual.

4. Por último, se tiene en cuenta el resultado del emparejamiento de puntos para ajustar la posición final del fotograma actual x_t .

Siguiendo este procedimiento, se obtiene un seguimiento del desplazamiento de la cámara preciso y con una eficiencia temporal mayor que la de otros algoritmos del estado del arte.

A continuación se explican en detalle los pasos 2, 3 y 4 para estimar la posición de un fotograma en el hilo de *Tracking*.

6.7.1. Alineamiento de la imagen

Una vez aplicado el modelo de movimiento al fotograma actual, se utiliza la información de intensidad de la imagen para estimar una primera posición del fotograma x_{t_e}, que será posteriormente refinada.

A partir de dos imágenes F_t y F_{t-1} que observan el mismo punto 3D X, se define el residuo de intensidad δF como la diferencia fotométrica entre los píxeles de cada fotograma en los que se observa X. La posición 2D en cada fotograma vendrá determinada por la

proyección de X en cada imagen, teniendo en cuenta la profundidad estimada que tenga el punto 3D en ese momento (Figura 6.11).

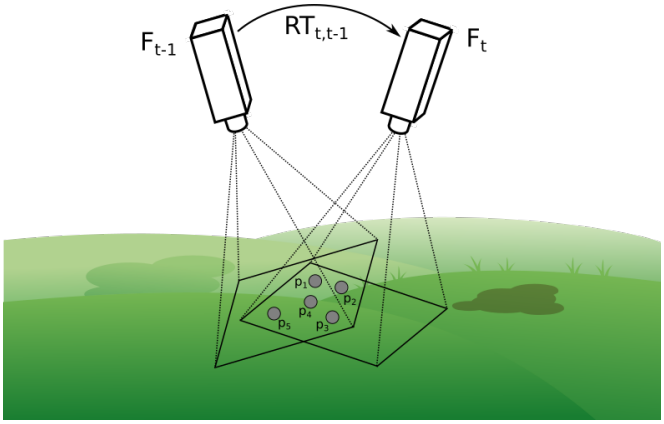


Figura 6.11: Alineamiento de imágenes basado en puntos.

Así, partiendo de la posición inicial del fotograma actual x_{t_i}, se calcula la posición estimada del fotograma x_{t_e} minimizando los residuos de intensidad δF de cada punto p_i detectado en el fotograma anterior (ecuación 6.10):

$$x_{t_e} = \underset{x_{t_i}}{argmin} \frac{1}{2} \sum_{i}^{n} ||\delta F(x_{t_i}, p_i)||^2 \tag{6.10}$$

Se trata de un problema no lineal de mínimos cuadrados que puede ser resuelto de forma eficiente mediante el algoritmo iterativo de Gauss-Newton [Kelley, 1999].

El residuo de intensidad δF se calcula mediante un parche de 4x4 píxeles alrededor de los píxeles en los que haya proyectado cada punto 3D, siguiendo la formulación del algoritmo de componente inversa [Baker and Matthews, 2004]. En este caso, los parches no se transforman antes de ser comparados, puesto que se supone que el desplazamiento entre cada fotograma será pequeño.

Como se deduce de lo anterior, no se intentan alinear las imágenes utilizando todos sus píxeles, sino que, con el propósito de aumentar la eficiencia temporal, solo se utilizan aquellos píxeles del fotograma anterior que estén asociados a un punto 3D y sean visibles desde el fotograma actual.

El proceso anterior se realiza en varios niveles de la pirámide para poder tolerar grandes desplazamientos. Así, primero se intenta converger con el nivel de la pirámide más alto (donde la imagen es más pequeña) y posteriormente se realiza el mismo procedimiento con niveles de la pirámide inferiores. En general, los niveles de la pirámide más bajos no suelen

aportar más precisión, por lo que se puede prescindir de su utilización para ahorrar tiempo de cómputo.

El resultado del alineamiento de la imagen será una posición estimada x_{t_e} que servirá como punto de partida para el resto de pasos del *Tracking*.

6.7.2. Alineamiento de puntos 3D

El paso anterior ha permitido alinear la cámara con respecto al fotograma anterior, estimando una posición del fotograma actual que será próxima a su posición real. Con esta nueva posición estimada, los puntos del mundo 3D proyectan en la imagen actual en una posición 2D cercana a su posición real; sin embargo, debido a inexactitudes en la posición de los puntos 3D y los fotogramas, estas proyecciones no son totalmente correctas.

En otros algoritmos basados solo en puntos (como PTAM), la posición del fotograma actual tendría una incertidumbre mucho mayor y el rango de búsqueda sería por tanto mayor. Para mejorar la eficiencia temporal, estos algoritmos suelen realizar una búsqueda de puntos 3D en dos fases:

- Búsqueda de grano grueso: Se selecciona un reducido subconjunto de puntos que se tratan de emparejar en la imagen actual con un rango de búsqueda grande. Los puntos emparejados sirven para realizar una nueva estimación tentativa de la posición del fotograma.
- Búsqueda de grano fino: Una vez que la incertidumbre en la posición del fotograma es menor, se utiliza un mayor número de puntos para realizar una búsqueda con un rango menor, que determinan la posición final del fotograma.

A pesar de realizar esta búsqueda en dos fases, los tiempos de cómputo son muy altos debido a que las búsquedas de grano grueso son muy costosas computacionalmente. En el caso de SDVL, el alineamiento de la imagen permite obtener una primera estimación de la posición del fotograma actual, por lo que solo es necesario realizar una búsqueda de grano fino y el tiempo de cómputo es menor.

El primer paso consiste en proyectar en el fotograma actual todos los puntos 3D que eran visibles en el fotograma anterior, guardando la posición candidata en la que han proyectado. La búsqueda de cada punto en la imagen se realiza utilizando la técnica de emparejamiento de la sección 6.5. Con el fin de aumentar la eficiencia temporal del algoritmo, no todos los puntos visibles se intentan emparejar, sino que solo se utiliza un subconjunto de éstos.

Al igual que sucedía en la inicialización de puntos 3D del *Mapping*, se ha dividido la imagen en una rejilla de celdas que sirve para clasificar los puntos 3D proyectados según su posición, asignando a cada punto proyectado la celda que le corresponda. Así, una vez que se encuentra un punto 3D en una celda, se ignoran el resto de puntos 3D asignados a esa celda. A su vez, también existe un número máximo de puntos encontrados que, en caso de alcanzarse, hace que se ignoren todos los puntos restantes.

Dentro de cada celda se trata de dar preferencia a aquellos puntos más fiables, es decir, aquellos que hayan sido vistos un mayor número de veces, para lo cual cada punto 3D almacena un contador que se incrementa cada vez que se empareja correctamente un punto 3D.

Cabe destacar que los puntos 3D se utilizan incluso aunque no hayan convergido y la incertidumbre en su profundidad sea alta. Esto se debe a que pueden existir puntos 3D alejados que no converjan porque no se alcance un paralaje suficiente; estos puntos no aportarán mucha información sobre la traslación de la cámara, pero pueden ser fundamentales a la hora de calcular su orientación.

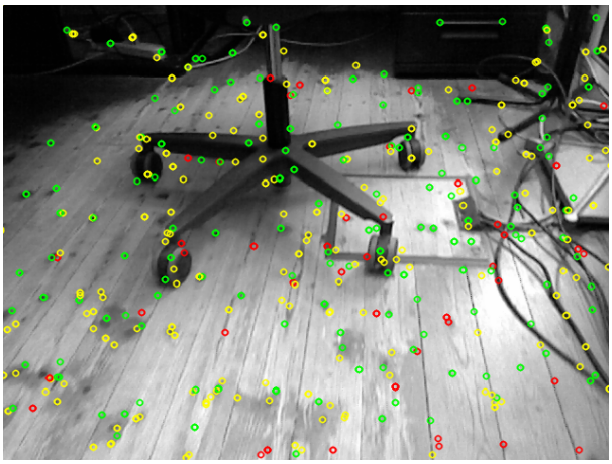


Figura 6.12: Puntos visibles durante el *Tracking*. Puntos correctamente emparejados (verde), no encontrados (rojo) y no utilizados (amarillo).

En la Figura 6.12 se muestra el resultado del procedimiento que se acaba de explicar. En la imagen aparecen todos los puntos del mapa que son visibles desde un fotograma de ejemplo. Algunos de los puntos son emparejados con éxito (en verde), mientras que otros no se encuentran (rojo) y, por tanto, se busca otro punto dentro de su misma celda. El resto de puntos (amarillo) no son utilizados por el *Tracking*, bien porque ya se dispone de otro punto encontrado en su misma celda o porque se alcanza el número máximo de puntos a utilizar.

6.7.3. Ajuste de la posición final

El último paso de algoritmo consiste en tener en cuenta los emparejamiento realizados en el paso anterior y la posición estimada de la cámara para calcular su posición final en la observación actual.

A partir de los emparejamientos obtenidos, se puede calcular el desplazamiento de la cámara x_t partiendo de la posición estimada x_{t_e}, minimizando el error de reproyección con la ecuación 6.11:

$$x_t = \underset{x_{t_e}}{argmin} \frac{1}{2} \sum_{i}^{n} ||p_i - f_p(x_{t_e}, X_i)||^2 \tag{6.11}$$

siendo $f_p(x_{t_e}, X_i)$ la función de proyección del punto 3D X_i en la imagen y p_i el píxel en el que se ha encontrado X_i durante el emparejamiento.

De nuevo, se trata de un problema no lineal de mínimos cuadrados que se resuelve de forma iterativa mediante Gauss-Newton. En nuestra implementación, en lugar de tratar de minimizar directamente el error cuadrático de reproyección, se utiliza como función objetivo el estimador robusto biponderado de Tukey [Huber, 1981], puesto que presenta un mejor comportamiento ante *outliers* y es más eficiente a la hora de estimar la posición final.

6.8. Rechazo de espurios

Uno de los elementos principales que afecta a la precisión el algoritmo, e incluso puede llevar a que el robot se pierda, es que existan en el mapa puntos 3D mal situados (espurios). Estos puntos pueden ser fruto de una mala triangulación o de puntos que se han desplazado, por ejemplo cuando una persona pasa por delante de la cámara del robot.

El primero de los casos, en el que la posición del punto no ha sido calculada correctamente, podría producirse por un mal funcionamiento del algoritmo de emparejamiento o porque la profundidad del punto 3D se haya calculado con poco paralaje.

Se han intentado eliminar los espurios mediante los siguientes mecanismos:

- Cada punto 3D calcula su posición a partir de su detección en varias imágenes, por lo que es difícil que el emparejamiento falle en todas ellas, aunque podría llegar a suceder en entornos con poca textura o texturas repetitivas.

- El alineamiento de imágenes utilizado hace que el algoritmo converja hacia su posición real incluso aunque las texturas sean poco favorables, facilitando el trabajo al mecanismo de emparejamiento.

- Por la propia parametrización de los puntos, la profundidad es variable a medida que las observaciones aportan más información y en todo momento se tiene en cuenta la incertidumbre en la profundidad de cada punto.

En cualquier caso, aunque existan espurios, éstos no afectan a la precisión del algoritmo siempre que sean casos aislados, puesto que el resto de puntos válidos compensarían esta situación.

Peores serían las consecuencias en el segundo caso, en el que uno o varios elementos de la escena se muevan. Si estos elementos son pequeños en comparación al tamaño de la escena que se observa, su existencia podría no tener consecuencias en la localización calculada, pero si ocupasen gran parte de la imagen podrían llevar a una localización poco precisa.

Por ello, se ha implementado un algoritmo de rechazo de espurios (RdE) para mejorar la localización en estos casos. El algoritmo desarrollado está basado en el trabajo realizado por [Civera *et al.*, 2010], en el cual se proponía un método para eliminar espurios en los algoritmos de SLAM con filtros de Kalman.

En el caso del algoritmo SDVL, el mecanismo de detección de puntos tiene lugar durante el alineamiento de puntos, de forma que solo un subconjunto de puntos 3D se utilizan para calcular el desplazamiento de la cámara desde el último fotograma. Una vez realizado el emparejamiento, la selección de puntos se realiza iterativamente mediante RANSAC siguiendo el siguiente procedimiento:

1. Se seleccionan m puntos aleatoriamente y se actualiza la posición de la cámara tan solo con éstos. El valor de m por defecto es 5, puesto que se considera el número mínimo de puntos necesarios para calcular el desplazamiento entre dos fotogramas [Nistér, 2004].

2. Con la nueva posición de la cámara, se comprueban cuántos puntos emparejados validan la posición estimada (seguidores). Para ello, se proyecta cada punto 3D en el fotograma actual y se comprueba el error de reproyección, guardando como válidos (*inliners*) los puntos cuyo error esté por debajo de un umbral, mientras que el resto se almacenan como inválidos (*outliers*).

3. Si el número de seguidores es el mayor encontrado hasta ese momento se almacenan los *inliners* y la posición calculada hasta el final del algoritmo.

El algoritmo de RANSAC seguirá hipotetizando nuevas posiciones hasta que se alcance el número máximo de hipótesis n_h. Este valor se actualiza en cada iteración para asegurar que se ha seleccionado al menos una hipótesis sin espurios con probabilidad p (por defecto $p = 99\,\%$), utilizando la ecuación 6.12:

$$n_h = \frac{log(1-p)}{log(1-(1-\epsilon)^m)} \tag{6.12}$$

donde ϵ es el ratio del número de *outliers* en la mejor hipótesis respecto al número total de puntos. Por ejemplo, en el caso de que un $10\,\%$ de los puntos fuesen *outliers*, el número de iteraciones realizado por el algoritmo sería de 5. Asimismo, se ha establecido un máximo de 100 iteraciones para evitar que aumente excesivamente el tiempo de cómputo del algoritmo.

Una vez actualizada la posición de la cámara con los puntos catalogados como *inliers*, se aumenta el umbral establecido del error de reproyección y se comprueba si alguno de los *outliers* está dentro de los márgenes y puede ser rescatado, lo que suele suceder con puntos cercanos a la cámara. Cuando alguno de los *outliers* es rescatado, se añade a la lista de *inliers* y se recalcula la posición de la cámara teniendo en cuenta los puntos rescatados.

En la Figura 6.13 se muestra un ejemplo de un fotograma en el que las partes móviles (la persona y la silla) han sido etiquetadas como *outliers* mientras que el resto de puntos se mantienen como *inliers*

Figura 6.13: Rechazo de espurios en entornos con elementos dinámicos: *inliers* en verde y *outliers* en rojo.

6.9. Relocalización

En caso de que el seguimiento se pierda, lo que puede suceder en caso de oclusiones o desplazamientos muy bruscos de la cámara, el algoritmo debe ser capaz de volver a localizarse cuando se vuelva a una zona conocida del mapa.

Se entiende que el robot se ha perdido si la localización es "mala" durante 3 observaciones consecutivas (ver sección 6.10); si esto sucede, se pone en marcha el mecanismo de relocalización. Solo se activa el mecanismo en esas condiciones para no ralentizar el algoritmo en situaciones normales de seguimiento correcto.

Para intentar la relocalización del robot, se recorren todos los *Keyframes* almacenados comenzando por el más reciente hasta que se encuentre uno que encaje con el fotograma actual. Primero, se trata de alinear la imagen actual con la del *Keyframe* mediante el método directo explicado en 6.7.1 y, en caso de que el error resultante sea pequeño, se intenta realizar el emparejamiento de puntos 3D.

Si se halla algún *Keyframe* que tras este procedimiento encuentre al menos 20 emparejamientos válidos, éste se toma como referencia para continuar con el *Tracking* a partir de su posición.

El tiempo de ejecución del procedimiento de relocalización puede ser muy alto y no funciona en tiempo real, por lo que podrían llegar a perderse observaciones en el proceso.

6.10. Fiabilidad de la localización

El algoritmo desarrollado guarda en todo momento una estimación de la fiabilidad de la localización, catalogándola como "buena", "insuficiente" o "mala" en función de los emparejamiento realizados.

La localización será "buena" siempre que el ratio entre puntos emparejados e intentos de emparejamiento sea mayor del 20 %. Se cataloga la localización como "mala" si este ratio es menor del 20 % y además el número de puntos emparejados está por debajo de un umbral (por defecto 20). En cualquier otro caso, la localización se etiqueta como "insuficiente".

Una localización "insuficiente" solo implica que el fotograma actual no puede añadirse al mapa como Keyframe, pero se sigue usando para actualizar los puntos del mapa y servirá como referencia para el siguiente fotograma. Por contra, una localización "mala" hace que el fotograma actual sea ignorado y se elimine sin ser utilizado.

6.11. Comparativa con otros algoritmos de SLAM

A continuación se muestra de manera esquemática el funcionamiento del algoritmo SDVL y del resto de algoritmos de SLAM que se describieron en el capítulo 2 del estado del arte, para ver de forma general sus similitudes.

Funcionalidad	Mono-SLAM	PTAM	SVO	LSD-SLAM	ORB-SLAM	SDVL
Probabilístico	Sí	No	No	No	No	No
Hilos de ejecución	1	2	2	3	3	2
Emparejamiento	Parches	Parches	Híbrido	Métodos directos	ORB	Híbrido
Puntos 3D con incertidumbre	No	No	Sí	Sí	No	Sí
Mapa inicial	Dado	Homo-grafía	Homo-grafía	Incerti-dumbre	Homog. / Matriz F.	Homo-grafía
Keyframes	No	Sí	Sí	Sí	Sí	Sí
Puntos en el mapa	Cientos	Miles	Miles	Miles	Miles	Miles
Mapa denso	No	No	No	Sí	No	No
Gestión de mapas grandes	No	No	No	Sí	Sí	No
Relocalización	No	Sí	Sí	Sí	Sí	Sí
Rechazo de espurios	No	No	No	Sí	Sí	Sí
Cierre de bucle	No	No	No	Sí	Sí	No

Capítulo 7

Experimentos con mapas desconocidos

En este capítulo se analiza el funcionamiento del algoritmo SDVL explicado en el capítulo anterior, comparándolo con otros algoritmos existentes en el estado del arte que han demostrado precisión y robustez. En concreto, se comparan los resultados de SDVL con los obtenidos en PTAM, SVO, LSD-SLAM y ORB-SLAM, que fueron descritos en detalle en el capítulo 2 del estado del arte.

Para llevar a cabo esta evaluación se van a utilizar tanto bases de datos internacionales como vídeos propios realizados para esta tesis. El procesador utilizado en todos los experimentos ha sido un *Intel(R) Core(TM) i5-3330 CPU @ 3.00GHz*.

Se han elegido varios escenarios de los descritos en el capítulo 3, que son representativos para validar experimentalmente las prestaciones de este algoritmo en términos de precisión, robustez y eficiencia temporal.

7.1. Análisis experimental del algoritmo SDVL

Con estos experimentos se busca caracterizar objetivamente la calidad de la autolocalización visual 3D con el algoritmo SDVL analizando, entre otros, su robustez, precisión y eficiencia temporal.

El algoritmo SDVL cuenta con múltiples parámetros de configuración que influyen en el resultado final. Los valores por defecto han sido establecidos experimentalmente para que funcionen en la mayoría de los escenarios. Los parámetros más importantes son:

1. Desplazamiento inicial: Desplazamiento mínimo en píxeles necesario para calcular el mapa inicial. Un valor demasiado pequeño puede hacer que no exista suficiente paralaje y el mapa calculado sea erróneo; normalmente será suficiente con establecer valores de entre 10 y 20 píxeles.

2. Número de *Keyframes*: Número máximo de *Keyframes* que almacenará el mapa, lo que afectará a la eficiencia temporal del algoritmo. Si este número es alto, el mapa generado contendrá más puntos 3D y la eficiencia temporal será menor tanto en el *Tracking* como en el *Mapping*.

3. Ratio de puntos perdidos: Porcentaje máximo de puntos perdidos desde el último *Keyframe*. Si este valor se supera, se creará un nuevo *Keyframe* incluso aunque no hayan transcurrido suficientes iteraciones desde que se añadió el último. Se analizará en profundidad su comportamiento en la sección 7.1.1.

4. Número de emparejamientos: Número máximo de emparejamiento utilizados en el *Tracking*. Un número alto mejorará la precisión pero empeorará la eficiencia temporal en la estimación instantánea de la posición 3D, lo que puede deteriorar globalmente la calidad de la estimación.

5. Rechazo de espurios (RdE): Activa o desactiva el rechazo de espurios durante el *Tracking*.

6. *Bundle Adjustment* (BA): Realiza una optimización local con *Bundle Adjustment* cada vez que se añade un nuevo *Keyframe*.

7. Descriptores ORB: Uso de descriptores ORB, en lugar de parches, para realizar el emparejamiento de puntos.

Los cuatro primeros parámetros deberán configurarse según las características del escenario, mientras que los tres últimos modificarán el comportamiento general del algoritmo.

A continuación se va a analizar el comportamiento del algoritmo SDVL en función de los valores de estos tres últimos parámetros, que dan como resultado ocho configuraciones distintas:

Configuración	ORB	BA	RdE
Config1	No	No	No
Config2	No	No	Sí
Config3	No	Sí	No
Config4	No	Sí	Sí
Config5	Sí	No	No
Config6	Sí	No	Sí
Config7	Sí	Sí	No
Config8	Sí	Sí	Sí

7.1.1. Precisión y eficiencia temporal en habitación con cámara libre

Para este experimento se utiliza la secuencia *Office Room 3* (OR3) del banco de pruebas ICL-NUIM, que fue descrita en el capítulo 3. Este vídeo está realizado con una cámara libre que realiza un movimiento circular alrededor de una habitación. El entorno se caracteriza por tener un tamaño reducido y con pocas texturas, por lo que el número de puntos del mapa no será muy alto.

La precisión de los algoritmos ha sido evaluada utilizando el error absoluto de trayectoria (ATE), que mide la diferencia entre cada posición estimada y la verdad absoluta teniendo en cuenta los sellos de tiempo; para ello, primero se alinean las trayectorias realizando una descomposición en valores singulares (SVD) y posteriormente se calcula la distancia entre cada par de puntos.

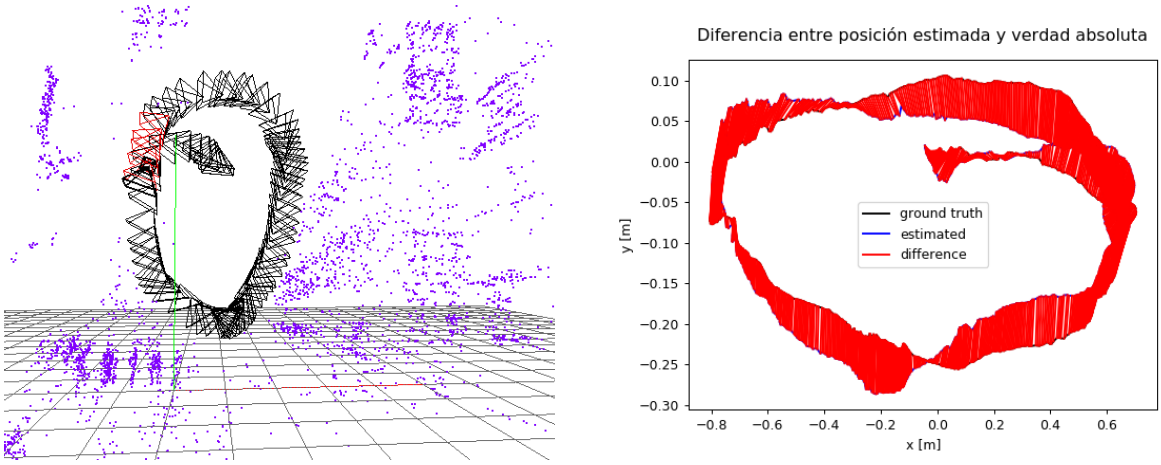
El valor que se tomará como referencia para determinar qué algoritmo es más preciso será la raíz del error cuadrático medio (RMSE), puesto que penaliza más los errores altos que otros estadísticos (como la media ponderada).

A continuación, se muestran los resultados de precisión (en metros) obtenidos con cada una de las ocho configuraciones del algoritmo SDVL. Los resultados se han obtenido ejecutando el algoritmo con cada configuración en tres ocasiones, mostrando en la tabla la más precisa. Los resultados pueden variar ligeramente en cada ejecución del algoritmo debido a que los puntos seleccionados para realizar el *Tracking* cuentan con cierta aleatoriedad.

Configuración	RMSE	Media	Mediana	σ
Config1	0.067	0.065	0.065	0.016
Config2	0.068	0.063	0.053	0.027
Config3	0.081	0.073	0.067	0.035
Config4	0.066	0.062	0.058	0.023
Config5	0.110	0.100	0.086	0.046
Config6	0.089	0.082	0.075	0.035
Config7	0.055	0.053	0.050	0.015
Config8	0.068	0.064	0.063	0.019

Los resultados obtenidos son similares con todas las configuraciones, variando en un rango de entre 5.5 y 11 cm. Los mejores resultados se han logrado con la configuración 7 (con ORB, con BA y sin RdE) y los peores con la 5 (con ORB, sin BA y sin RdE), aunque las diferencias son mínimas y no permiten sacar conclusiones.

En la Figura 7.1 se muestra el mapa generado y el error obtenido respecto a la verdad absoluta con la primera configuración.



(a) (b)

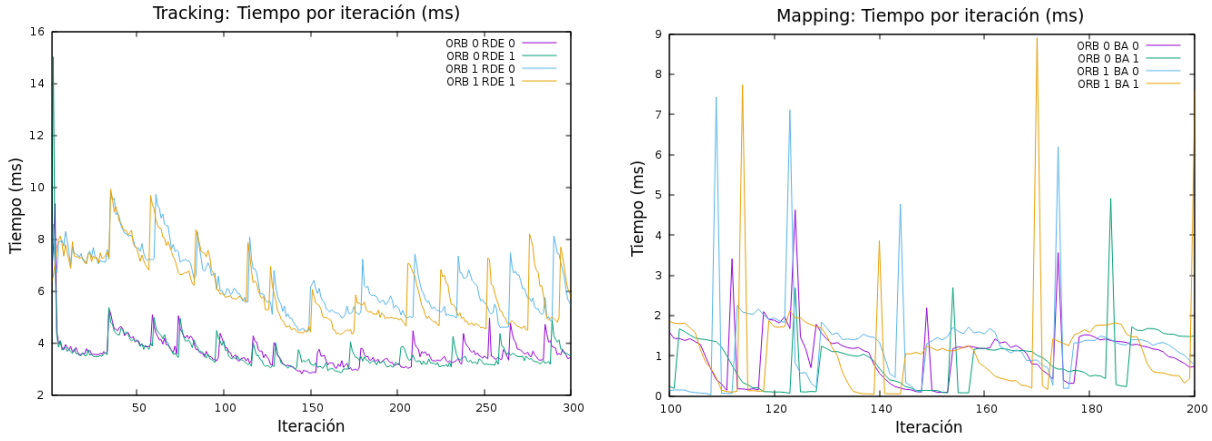
Figura 7.1: Mapa generado (a) y error obtenido respecto a la verdad absoluta (b) en secuencia OR3 (Config1).

Por su parte, la eficiencia temporal (en ms) de cada una de estas configuraciones se muestra en la siguiente tabla:

	Tracking			*Mapping*		
Configuración	Media	Mediana	σ	Media	Mediana	σ
Config1	3.84	3.55	1.04	3.50	2.46	3.43
Config2	3.72	3.45	0.98	3.46	2.50	3.32
Config3	3.88	3.56	1.09	3.88	2.28	5.23
Config4	3.69	3.45	0.94	3.84	2.49	4.66
Config5	6.05	5.59	1.99	3.88	2.60	4.48
Config6	5.81	5.43	1.78	3.67	2.65	3.71
Config7	5.87	5.48	1.87	4.74	2.65	3.98
Config8	5.92	5.52	1.79	3.92	2.70	4.44

Como se ve en la tabla anterior, en el hilo de *Tracking* los tiempo de cómputo son algo mayores al utilizar descriptores ORB que al emparejar mediante parches. Por su parte, el rechazo de espurios no ha variado significativamente los tiempos de cómputo, debido a que el mapa generado tenía un número de puntos reducido y pocos puntos espurios. Por último, el uso de *Bundle Adjustment* no es relevante para el *Tracking*, puesto que solo se utiliza en el hilo de *Mapping*.

En la Figura 7.2(a) se muestra el tiempo de cómputo por iteración con las cuatro configuraciones sin BA (Config1, Config2, Config5 y Config6); los cambios bruscos del tiempo de ejecución se deben al aumento del número de puntos que se produce tras añadir un nuevo *Keyframe*.



(a) (b)

Figura 7.2: Tiempos de ejecución de los hilos de *Tracking* (sin BA) (a) y *Mapping* (sin RdE) (b) en secuencia OR3.

En el hilo de *Mapping* los tiempos de ejecución son ligeramente superiores cuando se utilizan descriptores ORB. El RdE no afecta directamente al *Mapping*, por lo que no influye en la eficiencia temporal. Por su parte, el *Bundle Adjustment* tampoco afecta demasiado al tiempo de cómputo medio debido al tamaño reducido del mapa.

En la Figura 7.2(b) se muestran los tiempos de ejecución sin RdE (Config1, Config3, Config5 y Config7); en este caso, los picos se producen al añadirse nuevos *Keyframes*, ya que se procede a buscar nuevos puntos 3D en la imagen y se optimiza con BA (en Config3 y Config7).

Este experimento demuestra que el algoritmo SDVL tiene una alta precisión y una alta eficiencia con todas las configuraciones analizadas en entornos de pequeñas dimensiones.

Ratio de puntos perdidos: robustez y eficiencia temporal

Uno de los factores fundamentales que afectan a la eficiencia temporal del algoritmo es la frecuencia con la que se crean los *Keyframes*, puesto que, como se vio en la Figura 7.2, añadir un nuevo *Keyframe* al mapa hace aumentar el tiempo de ejecución de los hilos de *Tracking* y *Mapping*.

El ratio de puntos perdidos r_p se calcula mediante la ecuación 7.1:

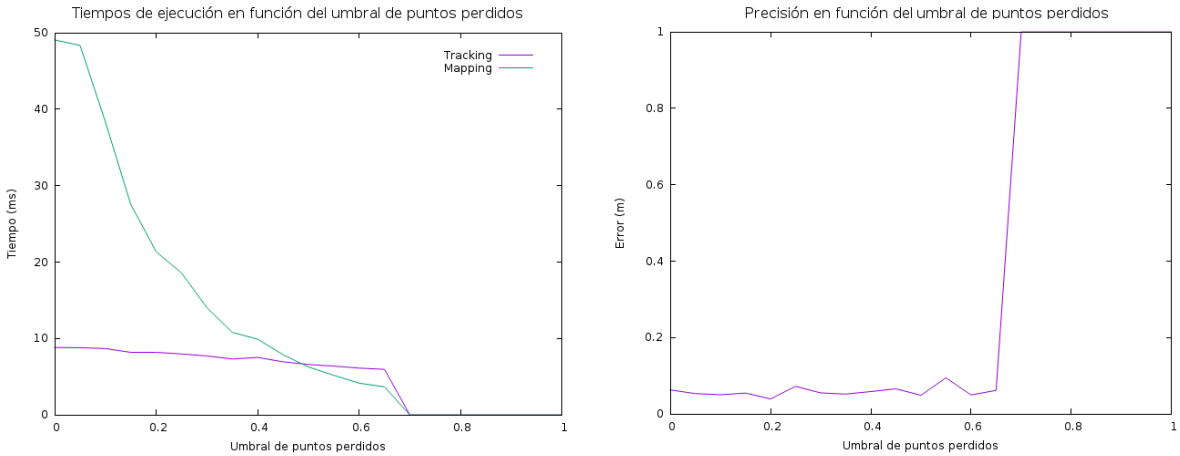
$$r_p = \frac{NP_f}{NP_{Kf}} \tag{7.1}$$

donde NP_f es el número de puntos 3D visibles en el fotograma actual y NP_{Kf} el número de puntos visibles en el último *Keyframe*. Si el valor de r_p supera un umbral, se añade un nuevo *Keyframe* al mapa independientemente de cuántas iteraciones hayan transcurrido desde que se creó el último.

Por una parte, interesa que el umbral establecido sea bajo para que se añadan *Keyframes* al mapa rápidamente cuando se produzcan desplazamientos bruscos, manteniendo así alto el número de puntos disponibles para el *Tracking*. Por otro lado, el umbral debería ser alto para que se creen menos *Keyframes*, lo que beneficiaría a la eficiencia del algoritmo.

En la Figura 7.3(a) se muestra cómo evoluciona el tiempo de cómputo de los hilos de *Tracking* y *Mapping* en función del umbral establecido. Con umbrales muy bajos, el tiempo de ejecución aumenta exponencialmente para el *Mapping* y de forma lineal para el *Tracking*, mientras que para valores altos (mayores a 0.7) el algoritmo se pierde y no es capaz de completar la secuencia.

Cabe destacar que, como se desprende de la Figura 7.3(b), un mayor tiempo de cómputo no tiene porqué significar una mayor precisión. Una vez que se establece un umbral suficientemente bajo como para que el robot no se pierda, la precisión es similar para todos los umbrales establecidos.



(a) (b)

Figura 7.3: Análisis del umbral de puntos perdidos. Eficiencia temporal (a) y precisión (b) en secuencia OR3 (con BA, RdE y descriptores ORB).

Por defecto, el valor establecido para el umbral ha sido de 0.3, aumentándolo en aquellos escenarios donde no se produzcan desplazamientos bruscos para mejorar la eficiencia temporal.

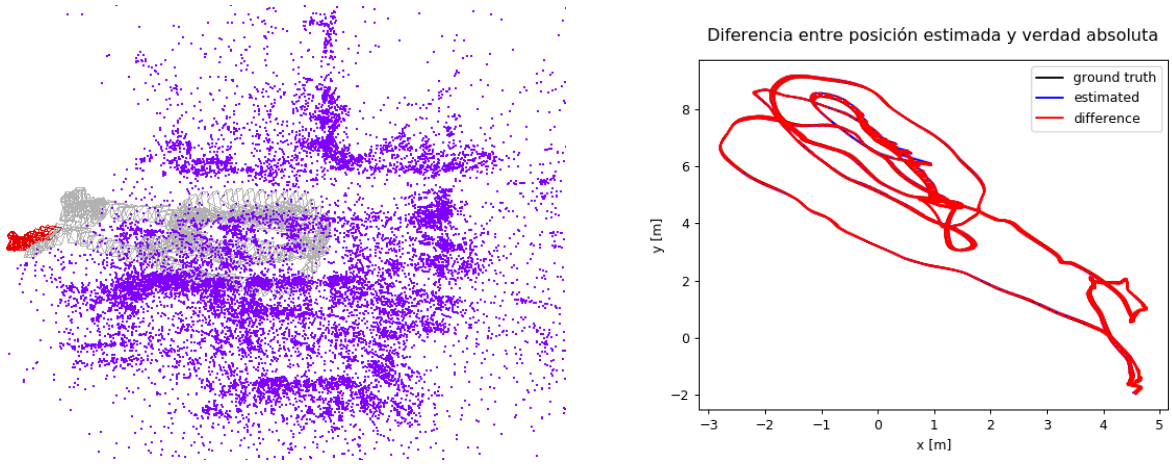
7.1.2. Precisión y eficiencia temporal en nave industrial con drones

Con este experimento se pretende validar el funcionamiento del algoritmo SDVL en escenarios más amplios y con una mayor complejidad que en el del experimento anterior, utilizando la secuencia *Machine Hall 01* (MH01) de la base de datos EUROC MAV, que fue descrita en el capítulo 3. Este escenario destaca por sus amplias dimensiones (una nave industrial de 400 m^2) y porque la escena ha sido grabada con un drone con seis grados de libertad.

A continuación se muestra la precisión obtenida con las ocho configuraciones del algoritmo SDVL analizadas en el experimento anterior:

Configuración	RMSE	Media	Mediana	σ
Config1	0.98	0.72	0.38	0.65
Config2	1.05	0.85	0.62	0.62
Config3	0.15	0.12	0.09	0.09
Config4	0.14	0.12	0.10	0.08
Config5	0.73	0.58	0.46	0.45
Config6	0.49	0.41	0.34	0.27
Config7	0.28	0.25	0.21	0.07
Config8	0.11	0.09	0.08	0.05

En este experimento sí que se aprecian diferencias importantes en función de los parámetros elegidos, siendo la mejor precisión de 11 cm (Config8, con ORB, BA y RdE) y alcanzando hasta más de un metro de error en el peor de los casos (Config2, sin ORB, sin BA y con RdE). En la Figura 7.4 se muestra el mapa generado y el error en la trayectoria estimada para la mejor configuración (Config8).



(a) (b)

Figura 7.4: Mapa generado (a) y error obtenido respecto a la verdad absoluta (b) en secuencia MH01 (Config8).

En escenarios grandes como el de este experimento, pequeños errores en la localización se acumulan con el paso de las iteraciones y llevan a una *deriva en la escala del mapa* que hace que la precisión empeore. Esta deriva se debe principalmente a dos factores: puntos 3D espurios y errores de emparejamiento.

A tenor de los resultados, el uso de *Bundle Adjustment* aumenta considerablemente la precisión del algoritmo, puesto que las cuatro configuraciones con mayor precisión han sido aquellas en las que se ha utilizado este mecanismo (Config3, Config4, Config7 y

Config8). Por tanto, la utilización de BA es un factor fundamental en escenarios de grandes dimensiones.

Por otra parte, tanto el rechazo de espurios como el emparejamiento con descriptores ORB aumentan levemente la precisión del algoritmo en este experimento, aunque el resultado no es tan claro como en el caso del BA.

Los tiempos de ejecución de los hilos de *Tracking* y *Mapping* en este escenario han sido:

	Tracking			*Mapping*		
Configuración	Media	Mediana	σ	Media	Mediana	σ
Config1	10.43	10.36	2.16	31.42	24.68	24.92
Config2	10.03	9.88	2.28	32.28	24.40	26.07
Config3	10.51	10.45	2.27	85.89	25.41	119.28
Config4	10.14	10.05	2.19	89.61	26.21	128.50
Config5	17.12	16.52	5.10	21.06	14.59	18.50
Config6	17.40	16.51	6.80	20.94	14.62	18.65
Config7	17.26	16.73	5.16	42.21	13.94	71.93
Config8	17.25	16.63	5.19	42.31	14.71	66.45

Como se puede comprobar, la eficiencia temporal ha sido menor que en el experimento anterior, tanto en el *Tracking* como en el *Mapping*, debido a que en este escenario el mapa generado cuenta con más puntos 3D y más *Keyframes*.

En el *Tracking*, el tiempo de ejecución aumenta considerablemente cuando se utilizan descriptores ORB, ya que su extracción es más costosa computacionalmente que los parches. Por su parte, el rechazo de espurios no afecta en gran medida a los tiempos de ejecución, lo que significa que el número de espurios ha sido pequeño.

En el hilo de *Mapping*, el tiempo aumenta notablemente al utilizar BA, como era previsible, puesto que debe optimizar un gran número de puntos 3D. Por otro lado, la eficiencia temporal es mayor al utilizar descriptores ORB; aunque esto puede parecer contradictorio con lo que sucede en el *Tracking*, se debe a que el emparejamiento con descriptores ORB es más excluyente que el emparejamiento por parches, haciendo que el número de puntos del mapa sea menor.

Análisis temporal desglosado

A continuación, se muestran en detalle los tiempo de ejecución de los hilos de *Tracking* y *Mapping* desglosados en componentes con las dos configuraciones que han sido más precisas

en el experimento anterior (Config4 y Config8):

		Config4 (sin ORB)			Config8 (con ORB)		
Hilo	Operación	Media	Mediana	σ	Media	Mediana	σ
Tracking	Iniciar Fotograma	4.73	4.69	0.84	4.67	4.69	0.77
	Alinear Imagen	1.95	1.72	0.80	1.10	1.04	0.45
	Alinear Puntos 3D	2.95	2.93	0.82	11.12	10.51	4.29
	Ajustar Posición	0.54	0.52	0.23	0.32	0.28	0.20
	Total	10.14	10.05	2.19	17.25	16.63	5.19
Mapping	Actualizar Puntos	23.18	20.49	10.64	13.82	12.99	5.78
	Iniciar Puntos	33.73	22.00	29.92	35.35	29.25	22.52
	Bundle Adjustment	230.62	216.33	114.43	105.62	85.75	73.44
	Total	290.01	279.53	109.03	163.67	146.16	78.82

Para el *Tracking*, los componentes evaluados han sido:

1. Iniciar Fotograma: Se genera la pirámide de la imagen para el fotograma actual y se obtienen los píxeles característicos FAST en cada nivel de la pirámide.

2. Alinear Imagen: Se calcula la posición del fotograma minimizando el error fotométrico respecto a la observación anterior.

3. Alinear Puntos 3D: Se proyectan en la imagen los puntos del mapa visibles desde la posición actual y se realiza el emparejamiento en 2D. Posteriormente, se seleccionan los *inliers* con el mecanismo de rechazo de espurios.

4. Ajustar Posición: Se calcula la posición final a partir de los emparejamientos seleccionados. Además, se tratan de rescatar *outliers* y, si es necesario, se recalcula la posición de la cámara.

Respecto al hilo de *Mapping*, solo se han tenido en cuenta las iteraciones en las que se añaden nuevos *Keyframes*; debido a esto, los tiempos obtenidos son mayores que los mostrados anteriormente. Los componentes evaluados se detallan a continuación:

1. Actualizar Puntos: Se actualizan los puntos 3D que no hayan convergido (candidatos) buscándolos en el fotograma actual.

2. Iniciar Puntos: Se inicializan nuevos puntos 3D buscándolos en otros *Keyframes* almacenados.

3. *Bundle Adjustment*: Se optimiza el mapa con los puntos 3D visibles desde el fotograma actual y desde los *Keyframes* a los que está conectado.

Tanto en el *Tracking* como en el *Mapping* existen otros mecanismos menores cuyo tiempo es despreciable y que no se han evaluado por separado, como por ejemplo la eliminación de puntos 3D no encontrados o *Keyframes* obsoletos.

En el *Tracking*, la diferencia principal ente las dos configuraciones del algoritmo SDVL analizadas es el alineamiento de puntos. Esto se debe a que la extracción de descriptores ORB es más costosa computacionalmente que la obtención de parches; en concreto, cada descriptor tarda alrededor de tres veces más de tiempo en ser extraído y comparado con otro descriptor que en el caso de los parches.

Otro de los elementos que más afecta al tiempo final del *Tracking* es la inicialización del fotograma. Este tiempo es bastante estable y depende principalmente del tamaño de las imágenes, que en este caso era de 752x480 píxeles.

En el hilo de *Mapping*, el tiempo de cómputo es mucho mayor cuando se realiza el emparejamiento con parches, debido a que este emparejamiento es menos excluyente que el de los descriptores ORB. Esto se traduce en un número mayor de emparejamientos y un número mayor de puntos 3D en el mapa. Esto influye en la actualización de puntos (hay que actualizar más puntos) y, especialmente, en el *Bundle Adjustment* (hay más puntos para optimizar). En este experimento, el número medio de puntos 3D optimizados con *Bundle Adjustment* utilizando parches ha sido de 4323, mientras que con descriptores ORB la media ha sido de 1632; esto explica por qué el tiempo es mucho mayor en el primer caso.

Por otra parte, la inicialización de nuevos puntos tarda algo más cuando se utilizan descriptores ORB. Aunque la extracción de los descriptores es más costosa, en este caso, la diferencia no es muy grande porque muchos de estos descriptores ya habrán sido calculados en iteraciones anteriores y estarán almacenados en memoria.

7.1.3. Robustez en entornos con elementos dinámicos

El rechazo de espurios cumple un papel fundamental cuando existen elementos dinámicos en el entorno. A continuación se muestran dos experimentos en los que algunos de los elementos del mundo son dinámicos, lo que permitirá analizar cómo influye la utilización del RdE. Las configuraciones del algoritmo que se van a utilizar en este análisis son Config1 y Config2, cuya única diferencia es que la segunda utiliza RdE y la primera no.

Vuelo sobre una calzada

El primero de los experimentos se ha realizado utilizando la secuencia URJC4, explicada en el capítulo 3, que ha sido grabada con un drone *HoverWasp* sobre una calzada. La mayoría de la escena es estática (el suelo y los árboles), mientras que algunos elementos (personas y coches) se desplazan por debajo del drone. En este caso no se dispone de verdad absoluta precisa, por lo que solo se comparan las trayectorias estimadas con y sin RdE. Las trayectorias seguidas son similares con las dos configuraciones (Figura 7.5), con un error medio entre ellas de tan solo 0.008 m, de manera que los elementos dinámicos no han influido significativamente en el algoritmo.

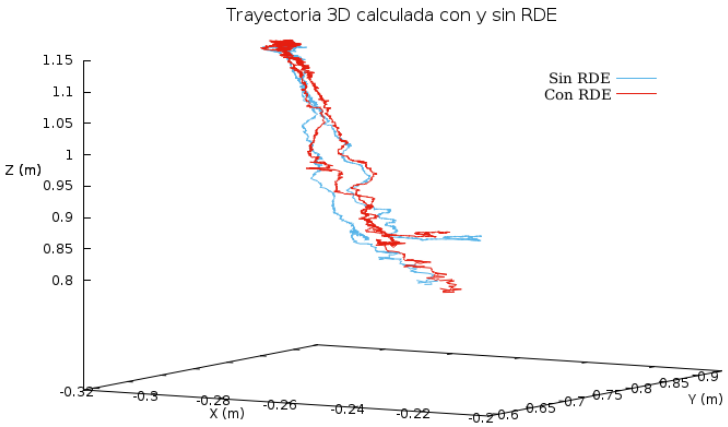


Figura 7.5: Trayectorias estimadas con y sin RdE en secuencia URJC4 con elementos dinámicos.

Los tiempos de cómputo también son similares, al no existir un número suficientemente grande de espurios en el mapa:

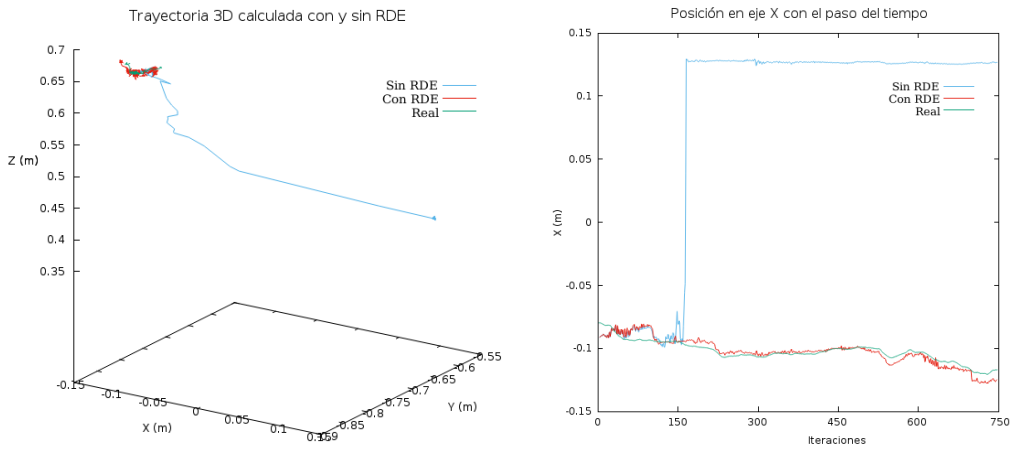
Configuración	Media	Mediana	σ
Config1 (sin RdE)	8.53	8.18	1.71
Config2 (con RdE)	8.43	8.09	1.67

Escenario con personas moviéndose

El segundo experimento se ha realizado utilizando la secuencia *Freiburg3 Walking Static* (F3W), que fue explicada en el capítulo 3. Al contrario de lo que sucedía en el experimento anterior, los elementos que se desplazan en la escena (personas) ocupan gran parte de la pantalla, por lo que resulta fácil que el algoritmo se pierda.

En la Figura 7.6 se muestra el resultado del experimento. En el caso de no utilizar RdE, la posición estimada se desplaza en un momento dado cuando una de las personas pasa

por delante de la cámara, haciendo que la posición calculada esté alejada de la realidad a partir de ese momento. Cuando se utiliza RdE esto no sucede y la posición estimada se mantiene estable en todo momento con un error medio respecto a la verdad absoluta de 0.012 metros.



(a) (b)

Figura 7.6: Trayectorias estimadas con y sin RdE en secuencia F3W con elementos dinámicos (a). Posición en eje X con el transcurso de las iteraciones (b).

Los tiempos de ejecución del hilo de *Tracking* son algo mayores cuando se utiliza RdE puesto que, al existir espurios, el algoritmo de RANSAC debe realizar varias hipótesis antes de encontrar un conjunto de puntos sin espurios:

Configuración	Media	Mediana	σ
Config1 (sin RdE)	4.82	4.72	0.91
Config2 (con RdE)	5.79	5.56	1.34

7.1.4. Robustez ante secuestros

Para este experimento se ha utilizado la secuencia *Freiburg2 360 Kidnap* (F2K) de la base de datos TUM. Aquí, un robot con ruedas se desplaza siguiendo una trayectoria circular por la escena y tras un periodo de tiempo se tapa la cámara durante unos segundos, simulando así un secuestro del robot. Posteriormente, el robot prosigue su trayectoria hasta que cierra el bucle.

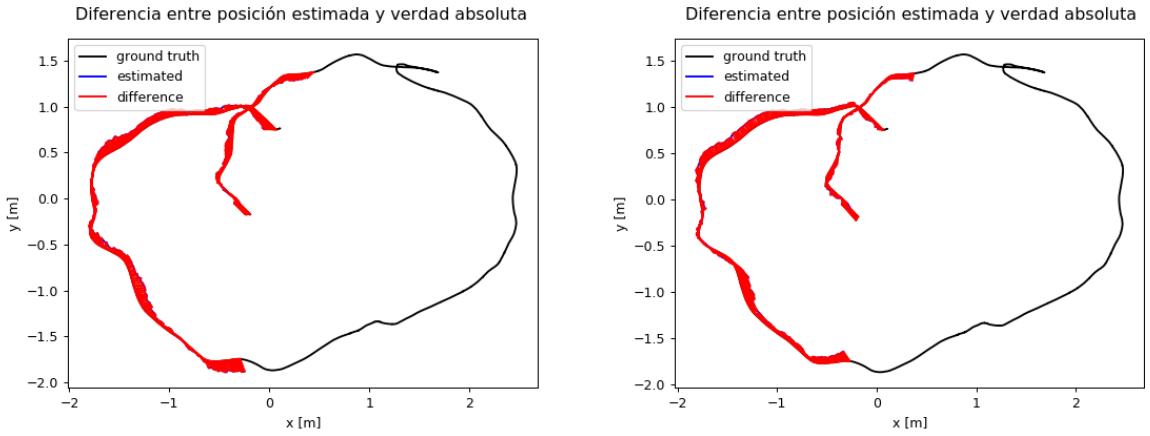
En este caso se comparan las configuraciones Config4 y Config8, que se diferencian tan solo en el uso de descriptores ORB para el emparejamiento de puntos. La precisión del algoritmo para estas dos configuraciones ha sido la siguiente:

Configuración	RMSE	Media	Mediana	σ
Config4 (sin ORB)	0.14	0.12	0.10	0.08
Config8 (con ORB)	0.11	0.09	0.08	0.05

Los tiempos de ejecución medios del hilo de *Tracking* mientras el robot estaba localizado han sido de 5.01 ms para la configuración sin ORB y de 8.21 ms para la configuración con ORB, debido al mayor tiempo de cómputo a la hora de extraer los descriptores.

Durante la relocalización, el tiempo de ejecución ha aumentado hasta 99.75 ms en el primer caso y a 43.89 en el segundo, dejando de funcionar en tiempo real en ambos casos. Al igual que sucedía con los tiempos de cómputo del hilo de *Mapping*, el tiempo de ejecución es mayor al utilizar parches porque el número de puntos que se almacenan en el mapa es mayor (los descriptores ORB son más excluyentes).

En la Figura 7.7 se muestran las trayectorias calculadas por las dos configuraciones de SDVL y la diferencia respecto a la verdad absoluta:



(a) (b)

Figura 7.7: Trayectoria estimada y verdad absoluta en secuencia F2K con secuestros con algoritmo SDVL: Config4 (a) y Config8 (b).

7.1.5. Casos de error

El algoritmo SDVL desarrollado funciona correctamente en la mayoría de los escenarios en los que ha sido probado. Sin embargo, se han encontrado determinadas situaciones en las que el algoritmo no se comporta correctamente, ya sea porque se pierde o porque la trayectoria calculada se aleja notablemente de la verdad absoluta. A continuación se resumen estos escenarios:

- Escenarios sin textura: En escenarios con poca textura, la técnica desarrollada no es capaz de realizar suficientes emparejamientos en las observaciones, por lo que el *Tracking* no tiene suficientes puntos en los que apoyarse y el algoritmo se pierde. Además, el robot no podrá relocalizarse a menos que vuelva a una zona del mapa anterior con textura suficiente.

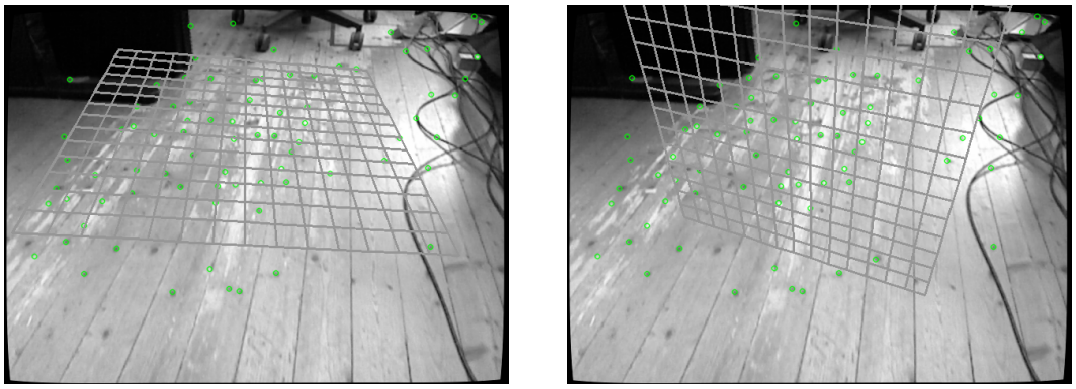
- Desplazamientos muy rápidos: Los desplazamientos muy rápidos producen errores en el emparejamiento principalmente por dos razones: los puntos 3D pueden salirse de los rangos de búsqueda en la imagen y las imágenes no suelen tener suficiente nitidez debido a este tipo de desplazamientos, ocasionando en ambos casos que no se emparejen correctamente los puntos y el algoritmo se pierda. Además, esto también dificulta la creación de nuevos puntos, al no ser observados en un número suficiente de fotogramas como para ser inicializados con garantías.

- Rotación sin traslación: Las rotaciones sin traslación hacen que los nuevos puntos 3D creados tengan una alta incertidumbre y solo sirvan como guía para estimar la orientación del robot. Esto ocasiona que las traslaciones no se calculen correctamente y el mapa pierda coherencia con la realidad.

- Escenarios con puntos muy alejados: En los escenarios donde todos los puntos detectados estén alejados, se producirá algo similar al caso anterior; los puntos del mapa tendrán una gran incertidumbre y la traslación no se calculará correctamente, lo que llevará a una localización errónea.

- Secuestros a nuevas zonas del mapa: Si el robot se pierde o se produce un secuestro, solo podrá relocalizarse si vuelve a una posición anterior conocida; si esto no se produce, el robot se perderá indefinidamente.

- Mapa inicial mal calculado: La generación del mapa inicial mediante homografía afecta en gran medida a la precisión del algoritmo, contaminando todas las subsiguientes estimaciones de posición. Para empezar, la inicialización por homografía parte de la base de que existe un plano dominante en la escena donde se sitúan una parte de los puntos detectados, lo que no tiene porqué producirse. Incluso en el caso de que esto suceda, la selección de puntos inicial puede llevar a realizar un mal cálculo del plano inicial (Figura 7.8), lo que hará que el algoritmo se apoye inicialmente en puntos 3D cuya posición es errónea.



(a) (b)

Figura 7.8: Homografía inicial bien (a) y mal (b) calculada en secuencia F1F utilizando distintos umbrales del detector FAST.

7.1.6. Valoración global del algoritmo SDVL

Con los experimentos realizados, se ha podido comprobar cómo el comportamiento del algoritmo depende de los parámetros de configuración que se utilicen. Las conclusiones que se pueden extraer de los tres parámetros de configuración analizados son:

- Rechazo de espurios: El RdE aumenta ligeramente la precisión del algoritmo en entornos estáticos sin perjudicar a su eficiencia, por lo que se recomienda su uso en este tipo de escenas. Además, juega un papel clave en entornos dinámicos, aumentando la robustez y la precisión del algoritmo a costa de disminuir ligeramente la eficiencia temporal.

- *Bundle Adjustment*: A pesar de la pérdida de eficiencia temporal que se produce en el hilo de *Mapping* al utilizar BA, su uso es un requisito indispensable si se necesita una alta precisión en escenarios de grandes dimensiones. En cualquier caso, no afecta a la eficiencia del *Tracking*, que es el hilo que debe funcionar en tiempo real.

- Descriptores ORB: Los descriptores ORB aumentan ligeramente la precisión del algoritmo, pero disminuyen la eficiencia temporal del hilo de *Tracking*, lo que podría afectar al seguimiento en tiempo real dependiendo de la capacidad de cómputo del robot.

En resumen, tanto el BA como el RdE han demostrado ser indispensables si se quiere disponer de un algoritmo con alta precisión y robustez. Por su parte, la utilización de descriptores ORB tiene ventajas (más precisión) e inconvenientes (menos eficiencia en el

Tracking), por lo que su utilización dependerá de la aplicación práctica que se quiera dar al algoritmo SDVL.

La valoración global del algoritmo es positiva. La precisión obtenida con algunas de las configuraciones analizadas (especialmente Config4 y Config8) ha sido de pocos centímetros de error, suficiente para los escenarios que se han analizado. La eficiencia temporal ha sido buena, con unos tiempos de cómputo del hilo de *Tracking* que dejan margen para funcionar en tiempo real en robots con poca capacidad de cómputo. Por último, el algoritmo ha demostrado ser robusto ante secuestros y entornos dinámicos, por lo que puede utilizarse en todo tipo de situaciones.

7.2. Comparativa con otros algoritmos de SLAM

Una vez validado experimentalmente y caracterizado objetivamente el funcionamiento del algoritmo SDVL propuesto en términos de precisión, robustez y eficiencia temporal, se compara ahora con otros algoritmos de SLAM del estado del arte.

De las ocho configuraciones que se analizaron en la sección anterior, se van a utilizar para la comparativa solo dos de ellas, las que presentaron mayor precisión y robustez en los experimentos realizados: la Config2 (sin ORB, con BA y con RdE) a la que se denotará "SDVL" y la Config8 (con ORB, con BA y con RdE) a la que se llamará "ORB-SDVL".

Las técnicas con las que se ha comparado son: PTAM, SVO, LSD-SLAM y ORB-SLAM, que se consideran las más ilustrativas e interesantes y se presentaron con detalle en el capítulo 2 del estado del arte. Se han utilizado las implementaciones de referencia de los autores de estos algoritmos, con algunas adaptaciones para suministrarles los mismos datos de entrada y recoger de manera homogénea sus resultados de salida. Los parámetros de configuración de estos algoritmos se han modificado en cada caso para obtener los mejores resultados posibles y los resultados que se muestran son los que han obtenido una mayor precisión tras 3 ejecuciones.

Las escenas utilizadas en la comparativa son algunas de las que se estudiaron en la sección anterior (OR3 y MH01) y otras cuatro escenas que fueron descritas en el capítulo 3 (MH02, F1F, F2D y ZU). La motivación para su elección es que representan distintos escenarios de experimentación con retos diferentes: interiores y exteriores, escenarios amplios y reducidos, robots aéreos y cámaras libres, etc.

7.2.1. Comparativa de precisión

La precisión obtenida (RMSE en metros) en cada uno de los experimentos realizados son los que se muestran a continuación:

Secuencia	SDVL	ORB-SDVL	PTAM	SVO	LSD-SLAM	ORB-SLAM
OR3	0.066	0.068	0.044	0.094*	0.092	0.085
MH01	0.147	0.112	X	0.135	0.254	0.044
MH02	0.115	0.077	X	0.052	1.395	0.035
F1F	0.026	0.041	0.069*	0.018*	0.366	0.013
F2D	0.054	0.034	X	X	0.103	0.027
ZU	1.112	0.548	X	0.331	0.508	0.429

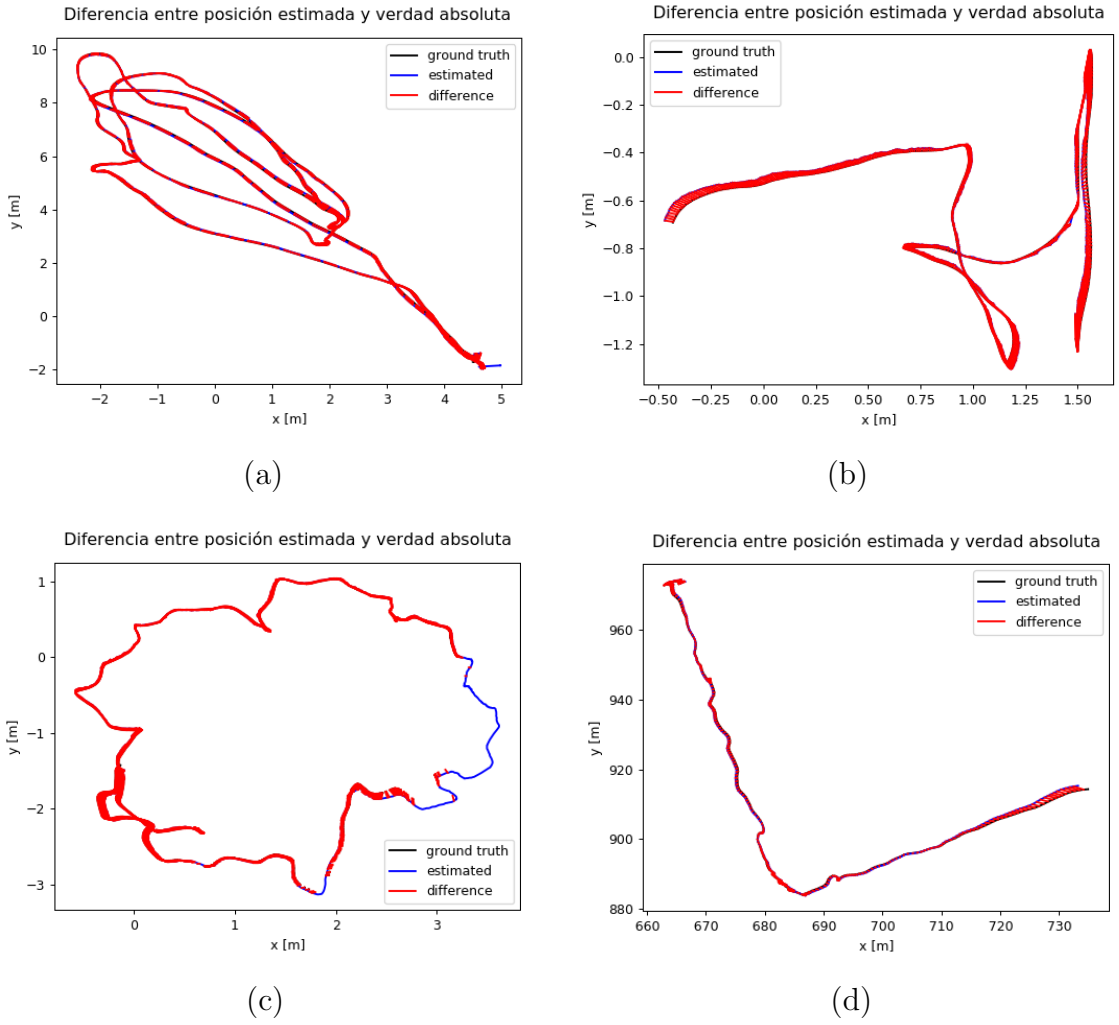


Figura 7.9: Trayectorias estimadas y verdad absoluta en secuencias MH02 (a), F1F (b), F2D (c) y ZU (d) con ORB-SDVL.

Los experimentos con una "X" son aquellos que no han sido capaces de realizar el seguimiento del robot durante al menos un 30 % del tiempo, mientras que los resultados con asterisco son los que han realizado el seguimiento parcialmente (entre el 30 y el 70 %). Por su parte, se señalan en rojo los experimentos que no han completado la secuencia y en azul los que han obtenido mejor resultado en cada una de ellas.

El algoritmo con mayor precisión en la mayoría de los experimentos ha sido ORB-SLAM, especialmente en escenarios con trayectorias amplias. Por su parte, SVO es también muy preciso en algunos escenarios, pero ha terminado perdiéndose en 3 de los 6 experimentos. LSD-SLAM y PTAM han sido los menos precisos en esta comparativa, el primero porque presenta el mayor error de localización en la mayoría de experimentos y el segundo porque solo ha logrado terminar la secuencia en uno de ellos.

En cuanto al algoritmo desarrollado en esta tesis, la precisión obtenida se ha situado en una zona intermedia respecto al resto de algoritmos, siendo más precisa en casi todos los experimentos la versión del algoritmo con descriptores ORB. En la Figura 7.9 se muestran las trayectorias estimadas por el algoritmo ORB-SDVL y el error respecto a la verdad absoluta de los cuatro últimos escenarios.

7.2.2. Comparativa de eficiencia temporal

Los tiempos de ejecución (en ms) del hilo de *Tracking* de los experimentos analizados en el apartado anterior han sido los siguientes:

Secuencia	SDVL	ORB-SDVL	PTAM	SVO	LSD-SLAM	ORB-SLAM
OR3	3.69	5.92	4.25	1.50*	7.91	20.27
MH01	10.14	17.25	X	3.18	16.63	34.72
MH02	9.93	16.99	X	3.14	16.87	30.62
F1F	6.06	8.49	5.15*	3.29*	14.53	19.39
F2D	6.15	10.32	X	X	12.46	27.07
ZU	8.47	16.34	X	2.36	17.47	26.16

En este caso el algoritmo con mayor eficiencia temporal ha sido SVO, con unos resultados excelentes de pocos milisegundos. Por otro lado, ORB-SLAM ha sido el que mayores tiempos de cómputo ha tenido, estando en algunos casos ligeramente por encima de los valores necesarios para funcionar en tiempo real. En cuanto a LSD-SLAM, los tiempos han estado en un valor intermedio entre los dos algoritmos anteriores, mientras que en

PTAM es difícil sacar conclusiones puesto que se ha perdido rápidamente en los escenarios más complejos.

El algoritmo propuesto ha tenido unos tiempos reducidos, situándose entre la eficiencia temporal de SVO y la de LSD-SLAM. En este caso, la eficiencia de la configuración sin ORB ha sido mucho mejor que con ORB, con tiempos cerca de un 40 % inferiores.

7.2.3. Comparativa de robustez

Con los resultados vistos anteriormente también se pueden extraer conclusiones sobre la robustez de los algoritmos.

PTAM ha demostrado funcionar tan solo en escenarios reducidos, perdiéndose rápidamente cuando la trayectoria seguida por la cámara se sale del mapa inicial. SVO también se pierde en escenarios donde se realizan giros en la trayectoria de la cámara, tanto rápidos (F1F), como lentos (OR3, F2D), debido a que su implementación original está pensada para ser utilizada con cámaras que apunten hacia abajo.

Tanto los algoritmos ORB-SLAM y LSD-SLAM como las dos versiones de SDVL no se han perdido en los experimentos realizados, por lo que se han mostrado como los más robustos ante este tipo de escenarios.

Es interesante resaltar que tanto ORB-SLAM como LSD-SLAM cuentan con mecanismos para manejar mapas de grandes dimensiones, como por ejemplo cierres de bucle, por lo que a priori su funcionamiento debe ser más robusto en este tipo de escenarios que SDVL.

También es necesario señalar que SDVL cuenta específicamente con un mecanismo de rechazo de espurios que ha mostrado ser de gran utilidad en escenarios con elementos dinámicos, algo que destaca respecto al resto de algoritmos.

7.2.4. Valoración global de la comparativa

Una vez se han comparado los algoritmos en cuanto a precisión, eficiencia temporal y robustez, se realiza en esta sección una valoración global de los algoritmos comparados a la luz de estas tres características.

Por su precisión y robustez, ORB-SLAM es el algoritmo que mejores resultados ha ofrecido, aunque también es el de menor eficiencia temporal; por ello, su uso en tiempo real está limitado a situaciones en las que se disponga de suficiente capacidad de cómputo.

El algoritmo SVO también ofrece grandes resultados en escenarios sin giros, especialmente en aquellos en los que la cámara apunte hacia el suelo, destacando por su excelente eficiencia temporal.

Tanto PTAM como LSD-SLAM no han destacado positivamente, debido a que PTAM es poco robusto en trayectos largos y que LSD-SLAM tiene una mala relación de precisión-eficiencia temporal.

Por su parte, el algoritmo SDVL, tanto con como sin descriptores ORB, guarda un equilibrio entre eficiencia temporal y precisión. No es el algoritmo con mayor precisión pero la relación de precisión-eficiencia temporal es buena, siendo indicado para plataformas con capacidad de cómputo reducida en las que se busque robustez. Entre las dos versiones del algoritmo, SDVL sin ORB cuenta con una mayor eficiencia, mientras que ORB-SDVL tiene una mayor precisión, por lo que la configuración a utilizar dependerá de los requerimientos que se necesiten en la localización.

Capítulo 8

Conclusiones

En los capítulos anteriores se ha visto en detalle el trabajo realizado en esta tesis doctoral, cuyo campo de investigación principal es la localización visual con robots. La localización es una funcionalidad útil para los robots autónomos, puesto que les ayuda a decidir cuál debe ser su comportamiento. La localización visual es especialmente útil, porque las cámaras son un sensor disponible en muchos robots y dispositivos, y además cuentan con múltiples aplicaciones prácticas: navegación autónoma, generación de mapas 3D, realidad aumentada, etc. Las soluciones a la localización visual se caracterizan por su robustez, precisión y eficiencia computacional.

En el capítulo 2 se han presentado las técnicas propuestas más ilustrativas en la literatura en este campo, dividiéndolas en dos grupos: los algoritmos para localización visual en entornos con mapas conocidos y los algoritmos para entornos desconocidos. Se han detallado en profundidad las que han servido como referencias directas para la realización de esta tesis.

En el capítulo 3 se han descrito las herramientas y los escenarios utilizados para validar la implementación *software* de los algoritmos desarrollados en este trabajo. Por un lado, se han visto los medios empleados para la validación: simuladores, robots reales y bases de datos internacionales; por otro, se han descrito los distintos escenarios de experimentación utilizados: el escenario de la RoboCup, el entorno de oficinas y el entorno de robótica aérea y cámaras libres.

Los capítulos 4 y 5 han servido para mostrar los aportes realizados en la localización visual en entornos conocidos. En el capítulo 4 se detalla el algoritmo evolutivo propuesto, cuyo desarrollo ha estado motivado por la necesidad de disponer de un algoritmo especialmente diseñado para entornos con múltiples simetrías, muy robusto y que

funcionara en tiempo real, ya que otros algoritmos clásicos no funcionan correctamente de manera sostenida con múltiples hipótesis en el largo plazo. En el capítulo 5 se ha validado experimentalmente el algoritmo evolutivo en dos escenarios distintos, el entorno de la RoboCup y un edificio de oficinas, comparando los resultados con diferentes implementaciones del algoritmo de Monte Carlo.

En los capítulos 6 y 7 se ha presentado el algoritmo propio SDVL, un algoritmo novedoso para localización visual en entornos desconocidos y especialmente diseñado para servir como método eficiente de localización visual en tiempo real en robots y dispositivos sin excesiva capacidad de cómputo. Este *software* se ha validado experimentalmente y se ha comparado con otros algoritmos de SLAM de contrastada utilidad, recurriendo a bases de datos internacionales con verdad absoluta para poder medir fehacientemente la precisión, robustez y eficiencia temporal de los algoritmos.

En este capítulo final se realiza un repaso del trabajo realizado, poniendo el énfasis en los resultados obtenidos y las contribuciones principales. Además, se revisarán los objetivos que fueron establecidos en el capítulo 1 para verificar su grado de cumplimiento. El final de este capítulo se centra en aquellas líneas de investigación y trabajos futuros que podrían derivarse del trabajo aquí realizado.

8.1. Cumplimiento de objetivos

El objetivo principal de esta tesis era ofrecer aportes en la autolocalización visual monocular de robots, tanto en entornos conocidos como en desconocidos. Este objetivo se ha logrado con el diseño e implementación de dos algoritmos originales propios: el algoritmo evolutivo para localización en entornos con mapa conocido y el algoritmo SDVL para entornos de los que no se dispone de información previa. Ambos han sido diseñados para funcionar con una única cámara RGB, pudiendo así ser utilizados en un gran número de robots, dispositivos móviles, etc. Además, han sido validados experimentalmente, caracterizados de modo objetivo y comparados con otros algoritmos de referencia.

8.1.1. Localización visual en entornos con mapa conocido

Los algoritmos clásicos empleados en entornos con mapas conocidos no son capaces de manejar correctamente entornos con simetrías, ya sea porque no mantienen prolongadamente en el tiempo múltiples hipótesis (caso de Monte Carlo, como se estudió en el experimento sintético de la sección 5.1) o porque su eficiencia no es la adecuada para

algoritmos en tiempo real (caso de los modelos de Markov). Debido a ello, en este trabajo se ha diseñado y desarrollado un algoritmo evolutivo especialmente preparado para manejar múltiples hipótesis en tiempo real, mejorando los resultados de otros algoritmos clásicos en escenarios con simetrías, y que además mantiene acotado el coste computacional para su funcionamiento en tiempo real.

El algoritmo evolutivo toma como entrada las observaciones de la cámara del robot. Estas observaciones son analizadas para extraer información relevante del mundo que rodea al robot, como píxeles o líneas, a partir de su geometría y sus colores. El análisis concreto de las observaciones depende del tipo de escenario en el que se encuentre el robot; en el escenario de la RoboCup se extraen de las imágenes píxeles que pertenecen a líneas y porterías, mientras que en el escenario de oficinas se buscan puertas y rodapiés. Además, cabe destacar que el algoritmo propuesto es lo suficientemente genérico como para que pueda ser adaptado a otro tipo de entornos.

Los píxeles extraídos de las observaciones sirven para evaluar cada posible posición del robot mediante un modelo de observación (subobjetivo a1), que se encarga de comprobar la concordancia de los datos de entrada con el mapa disponible para dar a cada posición una probabilidad. Si varias posiciones del mundo tienen una alta probabilidad y no se conoce con certeza la posición del robot, el algoritmo evolutivo es capaz de mantener cuanto sea necesario múltiples hipótesis en cada una de estas posiciones a través de sus razas, en espera de recibir más información que permita discriminar la posición correcta (subobjetivo a2).

Cada raza cuenta con un conjunto de explotadores encargados de analizar en profundidad una posición y sus alrededores en busca de la mejor solución local, los cuales evolucionan de una iteración a otra mediante operadores genéticos. Además, el algoritmo cuenta con una serie de exploradores que buscan nuevas posiciones del mundo que concuerden con los datos extraídos de las imágenes, lo que es de especial utilidad cuando se producen secuestros o en entornos con múltiples simetrías.

El algoritmo ha sido implementado en C++ y se ha integrado en el software del jugador del equipo de la URJC en la RoboCup. Además, ha sido validado experimentalmente en dos escenarios reales distintos (subobjetivo a3), el entorno de la RoboCup y un escenario de oficinas. Asimismo, se han comparado los resultados obtenidos con distintas variantes del algoritmo de Monte Carlo (filtro de partículas), que se ha tomado como referencia por su aceptación. Estas variantes se han implementado dentro de esta tesis para alimentarlos con exactamente los mismos datos que al algoritmo evolutivo. El algoritmo evolutivo ha demostrado su capacidad para funcionar en tiempo real, ofreciendo un comportamiento robusto a la hora de manejar múltiples hipótesis y ante secuestros.

8.1.2. Localización visual en entornos con mapa desconocido

Los algoritmos de SLAM con una sola cámara presentados en el estado del arte se caracterizan por su eficiencia temporal (SVO) o por su robustez (ORB-SLAM), de forma que la mejora en una de estas dos características suele ir en detrimento de la otra. Esto implica que muchos de estos algoritmos no puedan ser utilizados en robots o dispositivos con poca capacidad de cómputo, debido a su baja robustez o a sus requerimientos de *hardware*.

En esta tesis se ha diseñado y desarrollado un algoritmo propio de localización en entornos sin mapa conocido especialmente orientado a funcionar en robots y dispositivos con procesadores modestos. En él se ha buscado un equilibrio entre precisión y eficiencia temporal. A este algoritmo se le ha dado el nombre de SDVL, siglas de *Semi-Direct Visual Localization*.

El algoritmo SDVL analiza las imágenes de la cámara para obtener una lista de píxeles característicos mediante el algoritmo FAST, que sirven tanto para generar dinámicamente un mapa del entorno que rodea al robot como para calcular el desplazamiento de la cámara.

Inicialmente, el algoritmo utiliza dos imágenes que cuenten con desplazamiento suficiente entre ellas para calcular un mapa inicial mediante homografía. A partir de ese momento, el algoritmo añade nuevos puntos 3D al mapa a medida que se desplaza por el entorno (subobjetivo b1). Cada punto 3D tiene inicialmente una incertidumbre alta sobre su profundidad, que se reduce a medida que el punto es observados desde distintas posiciones, pudiendo converger en una posición 3D si se alcanza un paralaje suficiente entre las observaciones. El mapa es optimizado con frecuencia mediante *Bundle Adjustment* para que el error de reproyección de los puntos 3D en las observaciones se reduzca y aumente así la precisión y la robustez del algoritmo.

Utilizando el mapa de puntos 3D generado y las observaciones de la cámara, el algoritmo SDVL calcula su localización en el entorno utilizando tanto métodos directos (minimizando el error fotométrico entre píxeles) como píxeles característicos (minimizando el error de reproyección), lo que aumenta la eficiencia y la precisión del algoritmo (subobjetivo b2). Para emparejar los píxeles entre las imágenes se han realizado dos implementaciones distintas, una mediante parches en la imagen y otra con descriptores ORB. Además, se ha implementado un mecanismo de rechazo de espurios que aumenta la precisión y la robustez del algoritmo, especialmente ante entornos dinámicos.

En caso de que el robot se pierda o se produzca un secuestro, el algoritmo es capaz de detectar esta situación e intentará relocalizarse dentro del mapa que ha generado. Esta

relocalización solo será posible cuando el robot vuelva a una zona del entorno visitada previamente.

El algoritmo se ha implementado en C++ y ha sido validado experimentalmente analizando exhaustivamente las distintas configuraciones del algoritmo (subobjetivo b3), haciendo una valoración de precisión, eficiencia temporal y robustez. De estas configuraciones, se han seleccionado dos (etiquetadas como SDVL y ORB-SDVL) y se han comparado con otros algoritmos de SLAM disponibles en el estado del arte: PTAM, SVO, LSD-SLAM y ORB-SLAM. Fruto de esta comparativa, se ha comprobado como el algoritmo desarrollado guarda un equilibrio entre eficiencia temporal y precisión, lo que hace que sea adecuado para plataformas con poca capacidad de cómputo en las que se busque robustez.

8.2. Contribuciones

Algunas de las contribuciones en esta tesis son:

- Algoritmo evolutivo diseñado especialmente para autolocalizarse de forma eficiente en entornos con mapas con múltiples simetrías. Este algoritmo se utilizó en el equipo de fútbol robótico SpiTeam, que participó en la liga de la plataforma estándar de la RoboCup (SPL) hasta el año 2013.
- Algoritmo SDVL con alta precisión, eficiencia temporal y robustez, que está siendo integrando en el proyecto de *software* libre JdeRobot para su uso con drones, financiado parcialmente por Google.
- Mecanismo de rechazo de espurios: Técnica que ya había sido utilizada en algoritmos de SLAM con filtros de Kalman y que ha sido implementada de forma novedosa en el algoritmo SDVL, demostrando ser fundamental para obtener una localización precisa y robusta en entornos con elementos dinámicos.

Durante el desarrollo de esta tesis se han publicado diversos artículos en revistas internacionales con parte de este trabajo, entre los que destacan los siguientes:

- *LineSLAM: Visual Real Time Localization Using Lines and UKF*. E. Perdices, L. M. López y J. M. Cañas. ROBOT2013: First Iberian Robotics (pp. 663-678). Springer International Publishing, 2014.

En este artículo se detalla el algoritmo de SLAM desarrollado utilizando líneas en lugar de píxeles característicos y que sirvió como paso previo al algoritmo SDVL explicado en esta tesis.

- *Robot Evolutionary Localization Based on Attentive Visual Short-Term Memory.* J. Vega, E. Perdices y J. M. Cañas. Sensors, 2013 (**Revista JCR**).

  Artículo en el que se utiliza el algoritmo evolutivo desarrollado en esta tesis para localizar a un robot con información procedente de algoritmos de memoria visual.

- *Behavior-based Iterative Component Architecture for soccer applications with the Nao humanoid.* C. E. Agüero, J. M. Cañas, F. Martín y E. Perdices. En 5th Workshop on Humanoids Soccer Robots. Nashville, TN, USA. 2010.

  Se describe el comportamiento de la arquitectura BICA y las relaciones entre sus distintos componentes, entre los que se encuentra el algoritmo evolutivo desarrollado.

Además, actualmente está en proceso de aceptación otro artículo en el que se describe el algoritmo SDVL que se ha explicado en esta tesis.

8.3. Líneas futuras

Esta sección trata algunas líneas de investigación que pueden derivarse directamente de este trabajo y podrían ser una buena continuación:

- Localización híbrida: Los algoritmos desarrollados en esta tesis son independientes entre sí y cada uno está enfocado a un tipo de problema distinto. Por una parte, los algoritmos vistos en entornos conocidos son muy robustos ante falsos positivos y oclusiones al basarse en la acumulación de observaciones, mientras que su precisión es menor que en el caso de los algoritmos de SLAM.

  Podría estudiarse el uso de ambos algoritmos de forma simultánea como solución completa que integre las ventajas de las dos aproximaciones. Así, la localización global sería precisa y el mapa inicial se podría enriquecer con nueva información.

- Uso de cámaras RGBD: Las cámaras RGBD no pueden ser utilizadas en todos los escenarios por sus restricciones de luminosidad y distancia; sin embargo, en muchos de los entornos de experimentación empleados en esta tesis sí podrían haber sido

empleadas. Esto podría permitir mejorar tanto los modelos de observación en el caso de entornos conocidos, como aumentar la precisión de los algoritmos de SLAM.

Resulta interesante medir con el mismo tipo de algoritmo la ganancia que aporta el sensor RGBD frente al RGB en términos de robustez, precisión y tiempo de cómputo.

- Mapas densos en SDVL: Uno de los puntos débiles del algoritmo SDVL, así como de otros muchos basados total o parcialmente en píxeles característicos, es la calidad de los mapas generados. Mientras que estos mapas son muy útiles para calcular la localización del robot en ellos, no suelen ser un reflejo fidedigno de la realidad tridimensional.

  Podrían utilizarse técnicas como las desarrolladas en [Engel *et al.*, 2014] [Zia *et al.*, 2016] para mejorar la calidad de los mapas generados.

- Eliminación de *Keyframes* redundantes: Generar un mayor número de *Keyframes* evita que el robot se pierda con facilidad y mejora la precisión de los algoritmos; sin embargo, también empeora la eficiencia temporal de éstos cuando el mapa es muy grande, ya que almacena mucha información redundante.

  Una alternativa que ha sido implementada con éxito por otros autores [Tan *et al.*, 2013] [Mur-Artal *et al.*, 2015] consiste en generar *Keyframes* con mucha frecuencia y posteriormente eliminar los que sean redundantes sin perjudicar al *Tracking*. Esta técnica podría implementarse en el algoritmo SDVL para mejorar la robustez sin empeorar la eficiencia temporal.

- Uso de otras primitivas: Además de píxeles, podrían utilizarse otras primitivas en los algoritmos de SLAM, como líneas, planos o patrones 3D. Ya existen propuestas en este sentido realizadas por otros investigadores [Salas-Moreno *et al.*, 2013] [Salas-Moreno *et al.*, 2014] [Concha and Civera, 2014].

- Empleo con drones en vivo: Se pretende utilizar el algoritmo SDVL desarrollado a bordo de drones. Esto mejoraría el comportamiento de los robots aéreos; por ejemplo, el *AR.Drone* de Parrot cuenta con un estabilizador de vuelo basado en altura con sensores de ultrasonido que falla cuando se sitúa un objeto debajo, lo que no sucedería si basase su comportamiento en la localización puramente visual.

Bibliografía

[Agüero *et al.*, 2010] Carlos E. Agüero, José María Cañas, Francisco Martín, and Eduardo Perdices. Behavior-based iterative component architecture for soccer applications with the nao humanoid. In *5th Workshop on Humanoids Soccer Robots. Nashville, TN, USA*, 2010.

[Baker and Matthews, 2004] Simon Baker and Iain Matthews. Lucas-kanade 20 years on: A unifying framework. *International journal of computer vision*, 56(3):221–255, 2004.

[Baltzakis and Trahanias, 2002] Haris Baltzakis and Panos Trahanias. Hybrid mobile robot localization using switching state-space models. In *Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on*, volume 1, pages 366–373. IEEE, 2002.

[Barber *et al.*, 1996] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software (TOMS)*, 22(4):469–483, 1996.

[Berger, 1987] Marcel Berger. Geometry (vols. 1-2), 1987.

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

[Burchardt *et al.*, 2011] Armin Burchardt, Tim Laue, and Thomas Röfer. Optimizing particle filter parameters for self-localization. In *RoboCup 2010: Robot Soccer World Cup XIV*, pages 145–156. Springer, 2011.

[Burri *et al.*, 2016] Michael Burri, Janosch Nikolic, Pascal Gohl, Thomas Schneider, Joern Rehder, Sammy Omari, Markus W. Achtelik, and Roland Siegwart. The euroc micro aerial vehicle datasets. *The International Journal of Robotics Research*, 35(10):1157–1163, 2016.

[Canny, 1986] John Canny. A computational approach to edge detection. *IEEE Transactions on pattern analysis and machine intelligence*, (6):679–698, 1986.

[Cano, 2016] Jorge Cano. Building of an uav: from the hardware to the driver and autonomous applications. *Trabajo Fin de Grado. Universidad Rey Juan Carlos, Madrid, España*, 2016.

[Castle *et al.*, 2010] Robert O. Castle, Georg Klein, and David W. Murray. Combining monoslam with object recognition for scene augmentation using a wearable camera. *Image and Vision Computing*, 28(11):1548–1556, 2010.

[Chli and Davison, 2008] Margarita Chli and Andrew Davison. Active matching. *Computer Vision–ECCV 2008*, pages 72–85, 2008.

[Chli and Davison, 2009] Margarita Chli and Andrew J. Davison. Automatically and efficiently inferring the hierarchical structure of visual maps. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pages 387–394. IEEE, 2009.

[Civera *et al.*, 2008] Javier Civera, Andrew J. Davison, and José María Martínez Montiel. Inverse depth parametrization for monocular slam. *IEEE transactions on robotics*, 24(5):932–945, 2008.

[Civera *et al.*, 2009] Javier Civera, Andrew J. Davison, Juan A. Magallón, and José María Martínez Montiel. Drift-free real-time sequential mosaicing. *International Journal of Computer Vision*, 81(2):128–137, 2009.

[Civera *et al.*, 2010] Javier Civera, Oscar G. Grasa, Andrew J. Davison, and José María Martínez Montiel. 1-point ransac for extended kalman filtering: Application to real-time structure from motion and visual odometry. *Journal of Field Robotics*, 27(5):609–631, 2010.

[Clemente *et al.*, 2007] Laura A. Clemente, Andrew J. Davison, Ian D. Reid, José Neira, and Juan D. Tardós. Mapping large loops with a single hand-held camera. In *Robotics: Science and Systems*, volume 2, 2007.

[Concha and Civera, 2014] Alejo Concha and Javier Civera. Using superpixels in monocular slam. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pages 365–372. IEEE, 2014.

[Davison, 2002] Andrew J. Davison. Slam with a single camera. In *Workshop on Concurrent Mapping and Localization for Autonomous Mobile Robots, in conjunction with ICRA*, 2002.

[Davison, 2003] Andrew J. Davison. Real-time simultaneous localisation and mapping with a single camera. In *ICCV*, volume 3, pages 1403–1410, 2003.

[Dellaert *et al.*, 1999] Frank Dellaert, Wolfram Burgard, Dieter Fox, and Sebastian Thrun. Using the condensation algorithm for robust, vision-based mobile robot localization. In *Computer Vision and Pattern Recognition, 1999. IEEE Computer Society Conference on.*, volume 2, pages 588–594. IEEE, 1999.

[Duda and Hart, 1972] Richard O. Duda and Peter E. Hart. Use of the hough transformation to detect lines and curves in pictures. *Communications of the ACM*, 15(1):11–15, 1972.

[Engel *et al.*, 2013] Jakob Engel, Jurgen Sturm, and Daniel Cremers. Semi-dense visual odometry for a monocular camera. In *Proceedings of the IEEE international conference on computer vision*, pages 1449–1456, 2013.

[Engel *et al.*, 2014] Jakob Engel, Thomas Schöps, and Daniel Cremers. Lsd-slam: Large-scale direct monocular slam. In *European Conference on Computer Vision*, pages 834–849. Springer, 2014.

[Faugeras and Lustman, 1988] Olivier D. Faugeras and Francis Lustman. Motion and structure from motion in a piecewise planar environment. *International Journal of Pattern Recognition and Artificial Intelligence*, 2(03):485–508, 1988.

[Fischler and Bolles, 1981] Martin A. Fischler and Robert C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.

[Forster *et al.*, 2014] Christian Forster, Matia Pizzoli, and Davide Scaramuzza. Svo: Fast semi-direct monocular visual odometry. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pages 15–22. IEEE, 2014.

[Fox *et al.*, 1999a] Dieter Fox, Wolfram Burgard, Frank Dellaert, and Sebastian Thrun. Monte carlo localization: Efficient position estimation for mobile robots. *AAAI/IAAI*, 1999(343-349):2–2, 1999.

[Fox *et al.*, 1999b] Dieter Fox, Wolfram Burgard, and Sebastian Thrun. Markov localization for mobile robots in dynamic environments. *Journal of Artificial Intelligence Research*, 11:391–427, 1999.

[Fox, 2001] Dieter Fox. Kld-sampling: Adaptive particle filters. In *NIPS*, volume 14, pages 713–720, 2001.

[Gálvez-López and Tardos, 2012] Dorian Gálvez-López and Juan D. Tardos. Bags of binary words for fast place recognition in image sequences. *IEEE Transactions on Robotics*, 28(5):1188–1197, 2012.

[Gilks and Berzuini, 2001] Walter R. Gilks and Carlo Berzuini. Following a moving target—monte carlo inference for dynamic bayesian models. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 63(1):127–146, 2001.

[Gutmann, 2002] J. S. Gutmann. Markov-kalman localization for mobile robots. In *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, volume 2, pages 601–604. IEEE, 2002.

[Hahnel *et al.*, 2003] Dirk Hahnel, Wolfram Burgard, Dieter Fox, and Sebastian Thrun. An efficient fastslam algorithm for generating maps of large-scale cyclic environments from raw laser range measurements. In *Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, volume 1, pages 206–211. IEEE, 2003.

[Hamming, 1950] Richard W. Hamming. Error detecting and error correcting codes. *Bell Labs Technical Journal*, 29(2):147–160, 1950.

[Handa *et al.*, 2010] Ankur Handa, Margarita Chli, Hauke Strasdat, and Andrew J. Davison. Scalable active matching. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 1546–1553. IEEE, 2010.

[Handa *et al.*, 2014] Ankur Handa, Thomas Whelan, John McDonald, and Andrew J. Davison. A benchmark for rgb-d visual odometry, 3d reconstruction and slam. In *Robotics and automation (ICRA), 2014 IEEE international conference on*, pages 1524–1531. IEEE, 2014.

[Haralick *et al.*, 1994] Bert M. Haralick, Chung-Nan Lee, Karsten Ottenberg, and Michael Nölle. Review and analysis of solutions of the three point perspective pose estimation problem. *International journal of computer vision*, 13(3):331–356, 1994.

[Harris and Pike, 1988] Christopher G. Harris and JM Pike. 3d positional integration from image sequences. *Image and Vision Computing*, 6(2):87–90, 1988.

[Hartley and Zisserman, 2003] Richard Hartley and Andrew Zisserman. *Multiple view geometry in computer vision*. Cambridge university press, 2003.

[Hecht and Zajac, 1974] Eugene Hecht and A. Zajac. Optics addison-wesley. *Reading, Mass*, 19872:350–351, 1974.

[Heinemann *et al.*, 2006] Patrick Heinemann, Juergen Haase, and Andreas Zell. A combined monte-carlo localization and tracking algorithm for robocup. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, pages 1535–1540. IEEE, 2006.

[Hesch and Roumeliotis, 2011] Joel A. Hesch and Stergios I. Roumeliotis. A direct least-squares (dls) method for pnp. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 383–390. IEEE, 2011.

[Holland, 1975] John H. Holland. Adaptation in natural and artificial systems. an introductory analysis with applications to biology, control and artificial intelligence. *Ann Arbor: University of Michigan Press, 1975*, 1, 1975.

[Holland, 1992] John H. Holland. Genetic algorithms. 1992.

[Holmes *et al.*, 2008] Steven Holmes, Georg Klein, and David W. Murray. A square root unscented kalman filter for visual monoslam. In *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, pages 3710–3716. IEEE, 2008.

[Horaud *et al.*, 1989] Radu Horaud, Bernard Conio, Olivier Leboulleux, and Bernard Lacolle. An analytic solution for the perspective 4-point problem. *Computer Vision, Graphics, and Image Processing*, 47(1):33–44, 1989.

[Huber and Ronchetti, 1981] Peter J. Huber and E. M. Ronchetti. Robust statistics wiley. *New York*, 1981.

[Jazwinski, 1970] Andrew H. Jazwinski. Stochastic processes and filtering theory, 1970.

[Jochmann *et al.*, 2012] Gregor Jochmann, Sören Kerner, Stefan Tasse, and Oliver Urbann. Efficient multi-hypotheses unscented kalman filtering for robust localization. *RoboCup 2011: Robot Soccer World Cup XV*, pages 222–233, 2012.

[Kalman and Bucy, 1961] Rudolph E. Kalman and Richard S. Bucy. New results in linear filtering and prediction theory. *Journal of basic engineering*, 83(3):95–108, 1961.

[Kalman, 1960] Rudolph E. Kalman. A new approach to linear filtering and prediction problems. *Journal of basic Engineering*, 82(1):35–45, 1960.

[Kelley, 1999] Carl T. Kelley. *Iterative methods for optimization*. SIAM, 1999.

[Kiriy and Buehler, 2002] Evgeni Kiriy and Martin Buehler. Three-state extended kalman filter for mobile robot localization. *McGill University., Montreal, Canada, Tech. Rep. TR-CIM*, 5:23, 2002.

[Klein and Murray, 2007] Georg Klein and David Murray. Parallel tracking and mapping for small ar workspaces. In *Mixed and Augmented Reality, 2007. ISMAR 2007. 6th IEEE and ACM International Symposium on*, pages 225–234. IEEE, 2007.

[Kröse *et al.*, 2001] Ben J. A. Kröse, Nikos Vlassis, Roland Bunschoten, and Yoichi Motomura. A probabilistic model for appearance-based robot localization. *Image and Vision Computing*, 19(6):381–391, 2001.

[Kullback and Leibler, 1951] Solomon Kullback and Richard A. Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951.

[Kümmerle *et al.*, 2011] Rainer Kümmerle, Giorgio Grisetti, Hauke Strasdat, Kurt Konolige, and Wolfram Burgard. g 2 o: A general framework for graph optimization. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 3607–3613. IEEE, 2011.

[Labbe and Michaud, 2014] Mathieu Labbe and François Michaud. Online global loop closure detection for large-scale multi-session graph-based slam. In *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, pages 2661–2666. IEEE, 2014.

[Lepetit *et al.*, 2009] Vincent Lepetit, Francesc Moreno-Noguer, and Pascal Fua. Epnp: An accurate o (n) solution to the pnp problem. *International journal of computer vision*, 81(2):155–166, 2009.

[Li *et al.*, 2012] Shiqi Li, Chi Xu, and Ming Xie. A robust o (n) solution to the perspective-n-point problem. *IEEE transactions on pattern analysis and machine intelligence*, 34(7):1444–1450, 2012.

[Lin and Wang, 2010] Kuen-Han Lin and Chieh-Chih Wang. Stereo-based simultaneous localization, mapping and moving object tracking. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 3975–3980. IEEE, 2010.

[Longuet-Higgins, 1987] H. Christopher Longuet-Higgins. A computer algorithm for reconstructing a scene from two projections. *Readings in Computer Vision: Issues, Problems, Principles, and Paradigms, M. A. Fischler and O. Firschein, eds*, pages 61–62, 1987.

[Lovegrove and Davison, 2010] Steven Lovegrove and Andrew Davison. Real-time spherical mosaicing using whole image alignment. *Computer Vision–ECCV 2010*, pages 73–86, 2010.

[Lovegrove *et al.*, 2011] Steven Lovegrove, Andrew J. Davison, and Javier Ibanez-Guzmán. Accurate visual odometry from a rear parking camera. In *Intelligent Vehicles Symposium (IV), 2011 IEEE*, pages 788–793. IEEE, 2011.

[Lucas and Kanade, 1981] Bruce D. Lucas and Takeo Kanade. An iterative image registration technique with an application to stereo vision. 1981.

[Majdik *et al.*, 2017] András L. Majdik, Charles Till, and Davide Scaramuzza. The zurich urban micro aerial vehicle dataset. *The International Journal of Robotics Research*, 2017.

[Martin and Crowley, 1995] Jerome Martin and James L. Crowley. Comparison of correlation techniques. In *Intelligent Autonomous Systems*, pages 86–93, 1995.

[Martín *et al.*, 2007] Francisco Martín, Vicente Matellán, Pablo Barrera, and José María Cañas. Localization of legged robots combining a fuzzy-markov method and a population of extended kalman filters. *Robotics and Autonomous Systems*, 55(12):870–880, 2007.

[Matthies and Shafer, 1987] Larry Matthies and STEVENA Shafer. Error modeling in stereo navigation. *IEEE Journal on Robotics and Automation*, 3(3):239–248, 1987.

[McCann, 2015] Shawn McCann. 3d reconstruction from multiple images. 2015.

[Mei *et al.*, 2009] Christopher Mei, Gabe Sibley, Mark Cummins, Paul M. Newman, and Ian D. Reid. A constant-time efficient stereo slam system. In *BMVC*, pages 1–11, 2009.

[Milford and Wyeth, 2008] Michael J. Milford and Gordon F. Wyeth. Single camera vision-only slam on a suburban road network. In *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, pages 3684–3689. IEEE, 2008.

[Montiel and Davison, 2006] José María Martínez Montiel and Andrew J. Davison. A visual compass based on slam. In *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on*, pages 1917–1922. IEEE, 2006.

[Moreno *et al.*, 2002] Luis Moreno, José María Armingol, Santiago Garrido, Arturo De La Escalera, and Miguel A. Salichs. A genetic algorithm for mobile robot localization using ultrasonic sensors. *Journal of Intelligent and Robotic Systems*, 34(2):135–154, 2002.

[Mouragnon *et al.*, 2006] Etienne Mouragnon, Maxime Lhuillier, Michel Dhome, Fabien Dekeyser, and Patrick Sayd. Real time localization and 3d reconstruction. In *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, volume 1, pages 363–370. IEEE, 2006.

[Mur-Artal and Tardos, 2016] Raul Mur-Artal and Juan D. Tardos. Orb-slam2: an open-source slam system for monocular, stereo and rgb-d cameras. *arXiv preprint arXiv:1610.06475*, 2016.

[Mur-Artal *et al.*, 2015] Raul Mur-Artal, José María Martínez Montiel, and Juan D. Tardos. Orb-slam: a versatile and accurate monocular slam system. *IEEE Transactions on Robotics*, 31(5):1147–1163, 2015.

[Newcombe *et al.*, 2011] Richard A. Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J. Davison, Pushmeet Kohi, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. Kinectfusion: Real-time dense surface mapping and tracking. In *Mixed and augmented reality (ISMAR), 2011 10th IEEE international symposium on*, pages 127–136. IEEE, 2011.

[Newman *et al.*, 2005] Paul M. Newman, John J. Leonard, and Richard J. Rikoski. Towards constant-time slam on an autonomous underwater vehicle using synthetic aperture sonar. *Robotics Research*, pages 409–420, 2005.

[Nistér *et al.*, 2004] David Nistér, Oleg Naroditsky, and James Bergen. Visual odometry. In *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*, volume 1, pages I–I. Ieee, 2004.

[Nistér, 2004] David Nistér. An efficient solution to the five-point relative pose problem. *IEEE transactions on pattern analysis and machine intelligence*, 26(6):756–770, 2004.

[Nützi *et al.*, 2011] Gabriel Nützi, Stephan Weiss, Davide Scaramuzza, and Roland Siegwart. Fusion of imu and vision for absolute scale estimation in monocular slam. *Journal of intelligent & robotic systems*, 61(1):287–299, 2011.

[Ojanen, 1999] Harri Ojanen. Automatic correction of lens distortion by using digital image processing. *Rutgers University, Dept. of Mathematics technical report*, 1999.

[Olson *et al.*, 2000] Clark F. Olson, Larry H. Matthies, H. Schoppers, and Mark W. Maimone. Robust stereo ego-motion for long distance navigation. In *Computer Vision and Pattern Recognition, 2000. Proceedings. IEEE Conference on*, volume 2, pages 453–458. IEEE, 2000.

[Quinlan and Middleton, 2009] Michael J. Quinlan and Richard H. Middleton. Multiple model kalman filters: A localization technique for robocup soccer. In *Robot Soccer World Cup*, pages 276–287. Springer, 2009.

[Röfer and Jüngel, 2003] Thomas Röfer and Matthias Jüngel. Vision-based fast and reactive monte-carlo localization. In *Robotics and Automation, 2003. Proceedings. ICRA'03. IEEE International Conference on*, volume 1, pages 856–861. IEEE, 2003.

[Rosten and Drummond, 2006] Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. *Computer vision–ECCV 2006*, pages 430–443, 2006.

[Rublee *et al.*, 2011] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 2564–2571. IEEE, 2011.

[Rudoy and Wolfe, 2006] Daniel Rudoy and Patrick J. Wolfe. Monte carlo methods for multi-modal distributions. In *Signals, Systems and Computers, 2006. ACSSC'06. Fortieth Asilomar Conference on*, pages 2019–2023. IEEE, 2006.

[Salas-Moreno *et al.*, 2013] Renato F. Salas-Moreno, Richard A. Newcombe, Hauke Strasdat, Paul H. J. Kelly, and Andrew J. Davison. Slam++: Simultaneous localisation and mapping at the level of objects. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1352–1359, 2013.

[Salas-Moreno *et al.*, 2014] Renato F. Salas-Moreno, Ben Glocken, Paul H. J. Kelly, and Andrew J. Davison. Dense planar slam. In *Mixed and Augmented Reality (ISMAR), 2014 IEEE International Symposium on*, pages 157–164. IEEE, 2014.

[Sampson, 1982] Paul D. Sampson. Fitting conic sections to "very scattered" data: An iterative refinement of the bookstein algorithm. *Computer graphics and image processing*, 18(1):97–108, 1982.

[Scaramuzza and Siegwart, 2008] Davide Scaramuzza and Roland Siegwart. Appearance-guided monocular omnidirectional visual odometry for outdoor ground vehicles. *IEEE transactions on robotics*, 24(5):1015–1026, 2008.

[Shi, 1994] Jianbo Shi. Good features to track. In *Computer Vision and Pattern Recognition, 1994. Proceedings CVPR'94., 1994 IEEE Computer Society Conference on*, pages 593–600. IEEE, 1994.

[Simmons and Koenig, 1995] Reid Simmons and Sven Koenig. Probabilistic robot navigation in partially observable environments. In *IJCAI*, volume 95, pages 1080–1087, 1995.

[Smith *et al.*, 2006] Paul Smith, Ian D. Reid, and Andrew J. Davison. Real-time monocular slam with straight lines. 2006.

[Solis *et al.*, 2009] Andres Solis, Amiya Nayak, Milos Stojmenovic, and Nejib Zaguia. Robust line extraction based on repeated segment directions on image contours. In *Computational Intelligence for Security and Defense Applications, 2009. CISDA 2009. IEEE Symposium on*, pages 1–7. IEEE, 2009.

[Stewenius *et al.*, 2006] Henrik Stewenius, Christopher Engels, and David Nistér. Recent developments on direct relative orientation. *ISPRS Journal of Photogrammetry and Remote Sensing*, 60(4):284–294, 2006.

[Strasdat *et al.*, 2011] Hauke Strasdat, Andrew J. Davison, José María Martínez Montiel, and Kurt Konolige. Double window optimisation for constant time visual slam. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 2352–2359. IEEE, 2011.

[Strasdat *et al.*, 2012] Hauke Strasdat, José María Martínez Montiel, and Andrew J. Davison. Visual slam: why filter? *Image and Vision Computing*, 30(2):65–77, 2012.

[Sturm *et al.*, 2012] Jürgen Sturm, Nikolas Engelhard, Felix Endres, Wolfram Burgard, and Daniel Cremers. A benchmark for the evaluation of rgb-d slam systems. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 573–580. IEEE, 2012.

[Sunderhauf *et al.*, 2007] Niko Sunderhauf, Sven Lange, and Peter Protzel. Using the unscented kalman filter in mono-slam with inverse depth parametrization for autonomous airship control. In *Safety, Security and Rescue Robotics, 2007. SSRR 2007. IEEE International Workshop on*, pages 1–6. IEEE, 2007.

[Tan *et al.*, 2013] Wei Tan, Haomin Liu, Zilong Dong, Guofeng Zhang, and Hujun Bao. Robust monocular slam in dynamic environments. In *Mixed and Augmented Reality (ISMAR), 2013 IEEE International Symposium on*, pages 209–218. IEEE, 2013.

[Tardif *et al.*, 2008] Jean-Philippe Tardif, Yanis Pavlidis, and Kostas Daniilidis. Monocular visual odometry in urban environments using an omnidirectional camera. In *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, pages 2531–2538. IEEE, 2008.

[Thrun *et al.*, 2001] Sebastian Thrun, Dieter Fox, Wolfram Burgard, and Frank Dellaert. Robust monte carlo localization for mobile robots. *Artificial intelligence*, 128(1-2):99–141, 2001.

[Triggs *et al.*, 1999] Bill Triggs, Philip F. McLauchlan, Richard I. Hartley, and Andrew W. Fitzgibbon. Bundle adjustment—a modern synthesis. In *International workshop on vision algorithms*, pages 298–372. Springer, 1999.

[Vahdat *et al.*, 2007] Ali R. Vahdat, Naser NourAshrafoddin, and Saeed Shiry Ghidary. Mobile robot global localization using differential evolution and particle swarm optimization. In *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*, pages 1527–1534. IEEE, 2007.

[Vogiatzis and Hernández, 2011] George Vogiatzis and Carlos Hernández. Video-based, real-time multi-view stereo. *Image and Vision Computing*, 29(7):434–441, 2011.

[Wan and Van Der Merwe, 2000] Eric A. Wan and Rudolph Van Der Merwe. The unscented kalman filter for nonlinear estimation. In *Adaptive Systems for Signal Processing, Communications, and Control Symposium 2000. AS-SPCC. The IEEE 2000*, pages 153–158. IEEE, 2000.

[Wang, 1988] C. Ming Wang. Location estimation and uncertainty analysis for mobile robots. In *Robotics and Automation, 1988. Proceedings., 1988 IEEE International Conference on*, pages 1231–1235. IEEE, 1988.

[Wolf *et al.*, 2005] Jürgen Wolf, Wolfram Burgard, and Hans Burkhardt. Robust vision-based localization by combining an image-retrieval system with monte carlo localization. *IEEE Transactions on Robotics*, 21(2):208–216, 2005.

[Wright and Nocedal, 1999] Stephen Wright and Jorge Nocedal. Numerical optimization. *Springer Science*, 35:67–68, 1999.

[Zia *et al.*, 2016] M. Zeeshan Zia, Luigi Nardi, Andrew Jack, Emanuele Vespa, Bruno Bodin, Paul H. J. Kelly, and Andrew J. Davison. Comparative design space exploration of dense and semi-dense slam. In *Robotics and Automation (ICRA), 2016 IEEE International Conference on*, pages 1292–1299. IEEE, 2016.

[Zienkiewicz *et al.*, 2016] Jacek Zienkiewicz, Andrew Davison, and Stefan Leutenegger. Real-time height map fusion using differentiable rendering. In *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on*, pages 4280–4287. IEEE, 2016.