Navegación visual en robots móviles.

Luis J. Manso Fernández-Argüelles

Este documento se distribuye bajo licencia “Reconocimiento-Compartir bajo la misma licencia 3.0 España” de Creative Commons

Resumen

La navegación autónoma es la habilidad que diferencia a los robots móviles del resto. Uno de los requisitos más esenciales para desplazarse es la detección de obstáculos. El proyecto tiene como objetivo desarrollar un sistema visual de detección de obstáculos alternativo a los existentes.

A pesar de que ya existen enfoques basados en láser que dan buenos resultados para la navegación autónoma, sus precios son elevados y carecen de algunas características deseables. Debido a esto, resulta interesante disponer de una alternativa de bajo coste que siga manteniendo los requisitos de eficacia y eficiencia.

La solución a la que se llega es a un sistema de detección de obstáculos basado en la fusión de múltiples clasificaciones, una que se basa en información geométrica y otra basada en información cromática.

La memoria del proyecto, además de detallar la solución desarrollada, la compara con otras opciones previas, indica los problemas encarados y cómo han sido resueltos, muestra los resultados de los experimentos, e introduce RobEx y RoboComp, las plataformas hardware y software usadas. Finalmente, se describe la puesta en marcha del sistema en un robot RobEx.

Además, para demostrar la bondad del sistema desarrollado, el capítulo 5 se dedica a la descripción de los múltiples experimentos realizados y los resultados y conclusiones que se han extraido de la ejecución de los mismos.

El código del proyecto de fin de carrera se ha incorporado a RoboComp (repositorio de componentes software para robótica). Además, también se puede encontrar en el disco que acompaña a la memoria del proyecto, junto con videos y fotografías tomadas durante la fase de experimentación.

Agradecimientos

A todos aquellos a los que les habría hecho ilusión o esperaban ver su nombre en esta frase.

Índice general

1. Introducción **11**

1.1. Motivaciones 12

1.2. Clasificación de detectores de obstáculos 14

1.3. Objetivos 16

2. Punto de partida: RoboComp y RobEx **19**

2.1. El robot RobEx 20

2.2. Programación orientada a componentes 24

2.3. RoboComp 26

2.4. Entorno de desarrollo 27

3. Diseño del sistema **33**

3.1. Descripción global del sistema 34

3.2. baseComp 36

3.3. camMotionComp 39

3.4. camaraComp 41

3.5. floorColorCueComp y floorPPSComp 43

3.6. floorComp 44

3.7. localNavigatorComp 45

3.8. Diseño genérico de un componente 46

4. Algoritmos e implementación **49**

4.1. floorPPSComp 49

4.2. floorColorCueComp 58

4.3. floorComp 62

4.4. localNavigatorComp 67

5. Experimentos **69**

5.1. Calibración evolutiva de la homografía 69

5.2. Navegación 70

5.2.1. Guiado por geometría 71

5.2.2. Guiado por apariencia 72

5.2.3. Fusión 73

5.3. Comportamiento en suelos rugosos 73

5.4. Acceso compartido a la torreta 75

5.5. Calibración algebraica de la homografía 76

6. Instalación y puesta en marcha **79**

6.1. Instalación de requisitos 79

6.2. Desgarga 83

6.3. Compilación 83

6.4. Configuración 84

6.5. Ejecución 88

7. Conclusiones **93**

A. Modelo de base robótica diferencial **95**

B. La relación de homografía **99**

C. Configuración de OpenSSH para RoboComp **103**

Capítulo 1

Introducción

Una vez resuelto el problema del control de la mecánica del robot y la locomoción, la detección de obstáculos es la única barrera para poder disponer de un sistema de navegación local autónomo.

Para cumplir los objetivos de coste, en vez de un dispositivo láser, se propone un par estéreo de cámaras a cuyas imágenes se le aplican diferentes análisis. En ellos se usa información geométrica del robot y de la configuración de sus cámaras y estadísticos de la gama de color del suelo.

Todo lo propuesto a lo largo del proyecto se ha probado sobre una base robótica móvil, el robot RobEx. El sistema de visión está compuesto por dos cámaras USB colocadas en una torreta estéreo motorizada, montada a su vez sobre la base robótica. En la figura 1.1 se puede ver el robot usado. A pesar de que la implementación y las pruebas han sido llevadas a cabo sobre el robot RobEx usando la arquitectura de componentes software RoboComp, los conceptos y algoritmos son genéricos y se pueden usar en culaquier otra plataforma robótica que disponga de un par de cámaras.

RobEx es una base robótica cuyo diseño se distribuye como *hardware libre* bajo una licencia de tipo *Creative Commons Share-Alike*. La plataforma de software robótico RoboComp se distribuye bajo la licencia *GPL*. Así pues, ambos proyectos, desarollados en el Laboratorio de Robotica de la Universidad de Extremadura, son libres y se puede acceder a ellos mediante sus correspondientes páginas web [12, 13]). Dado que los resultados del proyecto de fin de carrera se han incluido dentro de RoboComp, el software desarrollado en él se distribuye

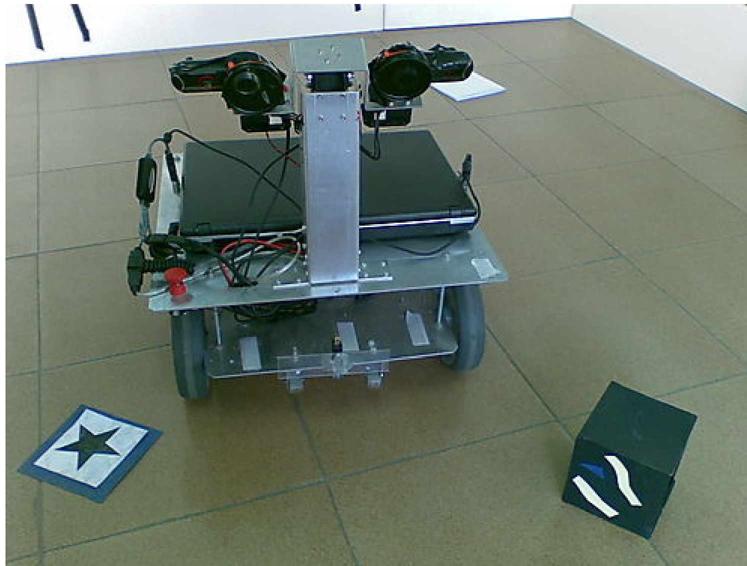

Figura 1.1: Robot RobEx equipado con una torreta estéreo.

bajo la misma licencia.

El software desarrollado es capaz de trabajar a 20Hz en un ordenador portátil común. Además, está preparado para distribuirse en varios ordenadores e interactuar con otros componentes software que, a su vez, pueden situarse en otros ordenadores.

1.1. Motivaciones

De la capacidad de navegación de un robot dependen radicalmente las aplicaciones que se le puedan dar, y por tanto, también el desarrollo de muchas de las areas en las que se espera que la robótica tenga presencia en los próximos años:

- Asistencia en entornos de oficinas.
- Transporte automatizado de humanos.
- Asistencia a personas discapacitadas.

Las técnicas de navegación autónoma se suelen dividir en base a su granularidad: navegación local y global. El nivel local, en el cual se centra el proyecto,

trata los aspectos necesarios para realizar pequeños trayectos procesando las entradas de los sensores y despl´azandose por un entorno cambiante sin chocar, y así, llegar a objetivos cercanos.

La navegación global trata los aspectos necesarios para que, partiendo de la base de que hay una buena navegación local, el robot llegue a un destino deseado, estando éste arbitrariamente lejos. Mientras la navegación local suele ser predominantemente reactiva, la navegación global es primordialmente deliberativa. Por tanto, la navegación global necesita disponer de un modelo del mundo suficientemente potente y de un mecanismo de localización del robot en dicho modelo. Con estos requisitos, la navegación global se resuelve encontrando una serie de subobjetivos cercanos que lleven hasta el objetivo final. Estos dos niveles suelen organizarse jerárquicamente en mayor o menor medida, por lo que la efectividad de la navegación global depende de la de los mecanismos de navegación local. Precisamente por la dependencia de la navegación global en la local es por lo que el proyecto se centra en esta última.

Debido a la importancia del campo, la comunidad científica ha hecho un gran esfuerzo para desarrollar técnicas que capacitan a los robots para navegar. En la mayoría de las soluciones suficientemente fiables se ha recurrido al uso de dispositivos láser. Sin embargo, a pesar de ser válidos en la mayoría de los casos, presentan las siguientes desventajas:

- Coste elevado. La relación es, aproximadamente, de 1 a 20.
- Limitaciones físicas en la detección. Dado que sólo barren un plano, y este suele ser paralelo al suelo[1]:
  - No detectan obstáculos por encima de ese plano (obstáculos altos, ver 1).
  - No detectan obstáculos bajos ni "*precipicios*".

[1]En caso de no situar el plano de barrido paralelo al suelo, la distancia máxima útil se vería reducida drásticamente. Sin embargo, de girar el ángulo normal al plano positivamente respecto al eje X lo suficiente, se pueden detectar precipicios. De hacerlo negativamente, se podrían detectar obstáculos altos. Alternativamente, los sensores láser se pueden motorizar para conseguir un barrido en dos dimensiones, pero encarece, complica y ralentiza el sistema.

- Consumo de energía mayor[2].
- Mediciones erróneas con algunos materiales. Por ejemplo, en paredes alicatadas.

Tipo	Nombre	Fabricante	Consumo
Cámara	Unibrain	Fire-i	0.9W
	FireFly MV	Point Grey	1W
	Flea2	Point Grey	2.4W
	DFW-VL500	Sony	4W
	PL-A741	PixeLINK	4.2W
Laser	URG-04LX	Hokuyo	4.5W
	UTM-30LX	Hokuyo	8.4W
	LMS200	Sick	20W
	LMS400	Sick	25W

Cuadro 1.1: Consumos de distintos dispositivos láser y cámaras.

Las cámaras tienen una doble ventaja respecto al precio ya que, además de la desigual relación entre sus costes, las cámaras tienen otros usos. Sin embargo, los láseres realizan medidas directas del entorno y no estimaciones, por lo que suelen tener un comportamiento más preciso.

Por tanto, no se sugiere el uso de técnicas de visión artificial por cuestiones de precisión, sino por su coste y la mayor riqueza perceptiva que pueden aportar.

1.2. Clasificación de detectores de obstáculos

Para la detección de obstáculos se han usado muchas técnicas y tecnologías, desde un dispositivo *fin de carrera*, hasta caros y complejos dispositivos LIDAR (del inglés, "LIght Detection And Ranging") de barrido en dos dimensiones. En esta sección se presentan las técnicas más conocidas.

[2] La tabla 1.1 ilustra los consumos de los dispositivos láser y cámaras usadas más comúnmente en robótica.

Las técnicas más simples se basan en el contacto físico del robot con los obstáculos. Se denominan generalmente *bumpers*, y son dispositivos con pulsadores u otros mecanismos de detección de presión montados sobre un parachoques. Evidentemente, con estos dispositivos, la notificación del obstáculo sucede una vez ha ocurrido el choque, por lo que se suelen usar únicamente como medida de emergencia.

Un siguiente paso pordrían ser los detectores por *ultrasonidos*. Éstos son dispositivos que emiten un sonido por encima del rango audible y producen una estimación del objeto más cercano en función del tiempo que tarda en volver la señal sonora. No se suelen instalar individualmente debido a que el ángulo cubierto por un solo sensor suele ser insuficiente. Si bien la precisión no es la característica más ventajosa de estos sensores, sí marcan una gran diferencia con respecto a los bumpers porque no es necesario el contacto físico. Son capaces de detectar obstáculos a algo más de un metro de distancia. Los detectores por *infrarojos*, a pesar de trabajar con otro tipo de ondas, tienen características similares.

La tecnología *láser* es la más común de las opciones usadas. Gracias a que sus mediciones son considerablemente precisas en la mayoría de las ocasiones, no sólo se usa para navegación local, sino que también se usa para la navegación global. Los lásers de gama alta diseñados para robótica autónoma alcanzan rangos de hasta treinta metros. Además, tienen la precisión suficiente como para que su uso sea apropiado para distinguir aproximadamente diferentes lugares. La técnica comunmente utilizada para esto se denomina SLAM (Simultaneous localization and mapping). Se puede consultar [16, 9] para más información sobre técnicas SLAM con láser.

A pesar de los problemas vistos en 1.1, los lásers son los dispositivos más populares debido a que la información que aportan no necesita ser procesada de ninguna forma para poder usarla. Esto hace que sea sencillo sacarles partido.

Las cámaras tienen un precio mucho más asequible que el láser. A pesar de que es dificil conseguir estimaciones tan precisas como las del láser con ellas, tienen propiedades de las que el láser carece:

- Hacen un muestreo en dos dimensiones.

- No tienen límites en cuanto a alcance.
- Los datos capturados pueden ser usados a la vez para otro tipo de problemas, no sólo para detectar obstáculos.

Como contrapartida, el láser tiene la ventaja de que puede funcionar de noche (aplicaciones en seguridad) y su proceso suele ser computacionalmente mucho menos costoso.

Hasta aquí se han visto los dispositivos más comunmente usados. Hay otros, de diferente aceptación, como lidars que incorporan información de reflectancia junto con la de profundidad. Éstos tienen las mismas aplicaciónes que tendría un par láser-cámara calibrado. También se usan otros que no se pueden catalogar como sensores de obstáculos pero que se usan para la navegación, como dispositivos GPS o las interfaces de redes inalámbricas IEEE 802.11. Estos últimos suponen que la posición de los puntos de acceso es estática y usan la potencia de señal de los puntos de acceso para localizarse.

1.3. Objetivos

El objetivo planteado en el proyecto es desarrollar un sistema visual de detección de obstáculos que sea fiable, práctico y que no monopolice el acceso a la torreta motorizada de la que dispone el robot.

Para comprobar la fiabilidad se establece un criterio mínimo de quince minutos sin chocar. Para comprobar que sea práctico se comprobará que pueda convivir en un ordenador portátil con otros componentes software de control de un robot sin influir en la eficacia del mismo o del resto del software. Para que no monopolice el acceso a la torreta, ha de trabajar con las imágenes que le lleguen, sin solicitar cambios en la posición de la torreta continuamente.

Además, deberá funcionar correctamente en todo tipo de situaciones, no sólo las ideales. El entorno en el que se ha de mover será fundamentalmente plano, pero el sistema debe ser lo suficientemente permisivo como para considerar suelo los objetos finos que, a pesar de no pertenecer realmente al suelo, no impiden que el robot pueda pasar sobre ellos. Además, el sistema debe ser

suficientemente robusto como para adaptarse a cambios en la iluminación y clasificar correctamente las zonas del suelo que presentan reflejos.

Resumiendo, las características del sistema han de ser:

- Robusto y fiable.
- No debe capturar el control de la torreta estéreo.
- Debe funcionar en tiempo real, a la máxima frecuencia posible.
- No debe depender del entorno, es decir, ser lo más genérico posible.

Capítulo 2

Punto de partida: RoboComp y RobEx

El desarrollo de este proyecto de fin de carrera no ha partido ni mucho menos de cero. Cuando el proyecto comenzó ya había una gran base de trabajo hecho, tanto a nivel de hardware, que se ha dado totalmente resuelto, como de software, que me ha permitido dedicar la mayor parte del tiempo a asuntos directamente relacionados con el proyecto.

Tanto el robot RobEx como algunos componentes usados, ya habían sido construidos en el Laboratorio de Robóbica y Visión Artificial de la Universidad de Extremadura. El software desarrollado usa algunos de los componentes que se encuentran en la plataforma RoboComp[13][11]. Además del desarrollo de los componentes propios del proyecto, durante la realización del mismo también se han hecho ligeras mejoras a algunos componentes de RoboComp que ya existían.

A lo largo de este capítulo se describirá el entorno de desarrollo y herramientas usadas, se introducirá el paradigma de programación orientado a componentes, y se hará una breve descripción del repositorio de componentes RoboComp y el robot RobEx.

2.1. El robot RobEx

El robot RobEx[12, 14] es de tipo diferencial, esto es, el movimiento del robot depende únicamente de la velocidad de rotación de sus dos ruedas motrices. Además de las ruedas motrices, el robot dispone de una tercera rueda, de giro libre, que sirve de apoyo. La simplicidad de diseño y de los cálculos asociados al modelo diferencial son las principales razones que han definido el diseño, no sólo de RobEx, sino de otros muchos robots y gamas de ellos, como pueden ser: Segway, Kphera, o Roomba. Para más información ver el apéndice A.

RobEx es una base robótica libre desarrollada en el Laboratorio de Robótica y Visión Artificial de la UEx, Robolab. Sus planos se distribuyen bajo una licencia Creative Commons de tipo Share-Alike. Por tanto, su diseño está abierto a cualquiera que lo quiera consultar[12] o contribuir a él. Al ser de tipo diferencial, su construcción es relativamente sencilla, por lo que con algunos conocimientos de electrónica, cualquiera la puede construir. RobEx está diseñado para llevar uno o varios ordenadores portátiles a bordo para realizar procesos complejos que sean necesarios, generalmente relacionados con la visión y la inteligencia artificial.

Tanto los motores de la base, de corriente continua, como el resto de la electrónica, se alimentan de una batería del tipo que se suele usar para extender la autonomía de los ordenadores portátiles. La base está controlada por un microcontrolador dedicado al que se accede a través de una interfaz RS232 sobre USB. Para más información sobre RobEx, se puede consultar[4, 12, 14].

Su principal orientación es la investigación y la docencia. Estos robots llevan utilizándose más de tres años a diario en las asignaturas de Robótica y Teoría de Sistemas que se imparten en la carrera de Ingeniería en Informática de la UEx. El origen de su diseño y desarrollo hasta su estado actual nació de los diferentes proyectos de investigación realizados en Robolab desde su creación en el año 1999. Desde su primera versión[14], cada mejora y nueva funcionalidad que incorpora se prueba intensivamente tanto en el laboratorio como en las aulas, por lo que el resultante es bastante robusto.

Los objetivos de diseño de los robots RobEx son:

- Ser apropiado para entornos estructurados, pero que a la vez pueda ser modificado para otros tipos de terreno.
- Conseguir un robot de bajo coste y fácil de construir con capacidad de procesamiento intensivo a bordo.
- Que el precio no esté reñido con la calidad: fiabilidad y robustez.
- Poder ser ampliado con diversos accesorios de sensorización y manipulación, así como llevar varios portátiles.

Como se indicó antes, el robot RobEx es hardware libre. El robot y su diseño se distribuyen bajo la licencia "Creative Commons Attribution-Share Alike 3.0". El software del microcontrolador se distribuye bajo GPL. Con esto se pretende:

- Que cualquier persona tenga acceso al diseño y software del robot, y pueda fabricarlo por sí misma.
- Que la comunidad participe en la mejora y evolución del robot.
- Que cualquier empresa pueda usar o vender RobEx, pero que cualquier modificación hecha al robot o a su software se haga pública y mantenga la misma licencia.

En la figura 2.1 se muestra una representación del diseño de la parte mecánica del RobEx.

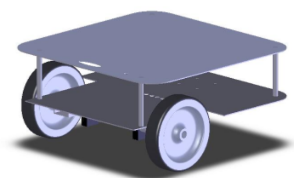

Figura 2.1: Representación del chasis de RobEx.

La estructura del robot está formada por dos planchas de aluminio que se separan y se afianzan entre sí mediante cuatro tubos de acero. Para colocar los motores en el chasis se usan soportes de acero, uno para cada motor. Los

soportes tienen dos orificios que se adaptan a los motores y a la vez se sujetan a la plancha de aluminio. Estos orificios son del mismo diámetro que el motor en dichos puntos de apoyo, de forma que éste queda firmemente sujeto cuando se aprietan los tornillos de fijación. La figura 2.2 ilustra el diseño de estos soportes.

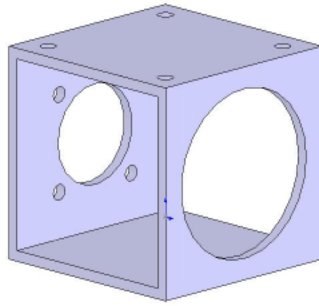

Figura 2.2: Representación de los soportes de los motores del chasis de RobEx.

Debido a que el diámetro del orificio de los ejes de las ruedas es de 15 mm, y el de los ejes de los motores es de 6 y 8 mm, se usan unos casquillos en aluminio con un pasador roscado para sujetar éste al motor. La sección mayor de los casquillos es suficientemente grande como para que puedan quedar unidos a las ruedas por presión. Para asegurar esta unión se utiliza también una gota de cianocrilato. La figura 2.3 muestra el diseño de los casquillos.

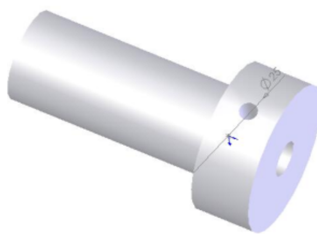

Figura 2.3: Representación de los casquillos que transmiten el movimiento de los motores a las ruedas de RobEx.

Para disponer de autonomía energética, el robot incorpora una batería recargable a bordo. Para ello se utiliza un batería de polímero de litio de 21,6V y 3500mAh. Estas baterías son de consumo doméstico común y se suelen utilizar

para ampliar la autonomía de ordenadores portátiles, por lo que son asequibles y están sobradamente probadas. Estas baterías proveen al robot de una autonomía de algo menos de dos horas.

La alimentación del robot tiene una tensión de salida de 12V, que corresponde con la tensión de funcionamiento de los motores. Además, tiene una segunda salida de fija de 5V que se puede usar para otros propósitos.

El sistema de control también se encarga de realizar las operaciones y cálculos necesarios para el control PID de los motores. El bucle de control PID de cada motor funciona a una frecuencia de 1 Khz. Por tanto, cada milisegundo se calcula la tensión de salida más apropiada para alcanzar el objetivo actual. En otros términos, el motor sólo está "sin control" o en lazo abierto periodos de 1 milisegundo.

La base dispone de un circuito integrado, LSI7266R1, que hace de contador de los pulsos procedentes de los dos codificadores ópticos. Éste se comunica con el microcontrolador mediante dos buses, uno de datos de 8 líneas de entrada y salida y otro de control de 5 líneas de entrada. Esto permite obtener los datos necesarios para llevar las cuentas de la odometría.

En la figura 2.4 se pueden ver dos RobEx básicos a los que se les ha añadido una cámara (ejemplar de la izquierda) y un escaner láser (ejemplar de la derecha).

Figura 2.4: Fotografía de dos RobEx básicos a los que se les ha añadido una cámara (izquierda) y un escaner láser (derecha).

2.2. Programación orientada a componentes

Dos de los principales problemas que se presentan cuando se crea software son la escalabilidad y la reusabilidad. Estos problemas son especialmente agudos cuando se refiere a software que se va a emplear en robótica. Esto es debido a que reaprovechar trozos de software es algo excepcionalmente común en este campo. A pesar de la importancia de la reusabilidad, generalmente se suele perder de vista este aspecto y se acaba creando software monolítico y poco reusable.

En el ámbito de la robótica es muy común que los investigadores implementen todos los algoritmos con un diseño rígido y orientado a una tarea y/o a un robot específico. De ser así, cuando finaliza la etapa de implementación el software desarrollado acaba siendo imposible de reusar. Suele estar tan ligado a una platafórma o tarea específica que resulta más práctico empezar de cero (debido a las dependencias y efectos colaterales derivados de su rigidez).

La programación orientada a componentes surge como solución a estos problemas. Es un enfoque que no tiene necesariamente que ver con concurrencia o computación distribuida, sino con cómo se organiza el software. La programación orientada a objetos representó un gran avance respecto a la programación estructurada, sin embargo, cuando el número de clases y sus interdependencias crece, resulta demasiado dificil entender el sistema globalmente. Es por tanto beneficioso disponer de un grado mayor de encapsulamiento, que aune diferentes clases relacionadas bajo una interfaz única, y permita comprender el sistema con menor grado de detalle. La programación orientada a componentes, que se propuso para solucionar este tipo de problemas, muchos la ven como el siguiente paso tras la programación orientada a objetos.

Un componente es un programa que provee una interfaz que otros programas (componentes) pueden usar. A su vez, estos programas hacen uso de programación orientada a objetos (así como en programación orientada a objetos se hace uso de programación estructurada). Esta dividisión en piezas de software de mayor tamaño que las clases, implementadas como subprogramas independientes ayuda a mitigar los problemas de los que se ha hablado antes y a aislar errores. Además, ayudan a distribuir la carga de cómputo entre núcleos, incluso, dependiendo de la tecnología usada, entre diferentes ordenadores en red.

Desde el punto de vista del diseño se pueden ver como una gran clase que ofrece métodos públicos. La única diferencia desde este punto de vista es que la complejidad introducida por las clases de las que depende el componente (o clase) que no son del dominio del problema desaparece porque la interfaz del componente las esconde. Un componente puede ser arbitrariamente complejo, pero un paso atrás, lo único que se ve es la interfaz que ofrece. Esto es lo que lo define como componente.

Ice es un framework de programación orientada a componentes muy interesante para su uso en robótica. A pesar de estar desarrollado por una empresa privada es open-source y está licenciado bajo GPL. Además de su caracter libre hay otras dos razones para elegir Ice. La primera razón es su facilidad de uso: al contrario que otras tecnologías como Corba, la filosofía de Ice es soportar sólo aquellas características que los usuarios vayan realmente a necesitar y evitar introducir otras características que raramente se usan y dificultan el aprendizaje. La segunda razón es la eficiencia: si bien Ice codifica la comunicación eficientemente tanto en términos de tiempo como de espacio, otras alternativas suelen codificar la comunicación en XML. El uso de XML tiene ventajas en otras aplicaciones donde es conveniente que los humanos puedan entender el tráfico y no sean factores críticos ni la latencia ni la eficiencia, pero dentro de un robot esto no ocurre.

Ice soporta dos tipos de comunicación, por llamada remota (tipo RPC) o por suscripción (mediante un servicio que hace de servidor de mensajes). Sin embargo, este último introduce un paso intermedio cuya latencia hace desaconsejable su uso en robótica.

Para realizar una conexión con un componente lo único que se ha de conocer es su "endpoint", es decir, la información necesaria para realizar la conexión: la dirección o nombre del ordenador al que se va conectar, el protocolo, el puerto y el nombre de la interfaz. Por ejemplo:

$$< nombre >:< tcp/udp > -p\ puerto\ -h\ host$$

donde 'nombre' es el nombre de la interfaz del componente al que se quiere hacer la conexión, 'puerto' es el puerto tcp o udp en el que el componente esté escuchando y 'host' el nombre del ordenador donde el componente se está

ejecutando.

Desde el punto de vista del programador, una vez la conexión está hecha, el uso de componentes Ice es extremadamente simple. Cuando se realiza una conexión se crea una instancia de un *proxy* al componente en forma de objeto. Al ejecutar un método del componente proxy el framework se encarga automáticamente de redirigir la llamada al componente remoto, por lo que la instancia del proxy se utiliza como si se tratase de una instancia de un objeto con la funcionalidad del componente.

Esta idea de usar un recurso remoto dentro de un programa no es nueva. Antes de la llegada de la programación orientada a objetos, ya había un protocolo en Unix llamado RPC para hacer llamadas remotas a procedimientos. Más tarde surgieron tecnologías como RMI, CORBA, DCOM, o Ice.

Para usar programación orientada a componentes se ha de dividir el diseño del software en piezas que ofrezcan una interfaz. A cambio obtendremos mayor reusabilidad, nos será más fácil aislar y encontrar fallos, y conseguiremos eliminar la necesidad de contemplar cientos de clases para comprender el software que desarrollemos. Para una descripción más detallada de Ice y de cómo se usa se puede consultar [8].

2.3. RoboComp

RoboComp[13], es un repositorio de componentes basados en Ice con aplicaciones en robótica y visión artificial. RoboComp se comenzó a desarrollar en Robolab en 2005. Actualmente el proyecto ha sido migrado a SourceForge, donde, además de tener la página del proyecto (http://sf.net/projects/robocomp), dispone de un wiki (http://robocomp.wiki.sf.net/), donde hay documentación y un repositorio al que se puede acceder incluso directamente con un navegador web (https://robocomp.svn.sf.net/svnroot/robocomp).

Dispone de componentes para captura y visualización de video, control del robot RobEx, detección y mantenimiento de regiones de interés (ROI), lectura y visualización de láser, lectura de joystick y Wiimote, entre otros muchos. Además de componentes, dispone de un generador automático de componentes y de

un programa gráfico de manipulación y control de componentes, managerComp.

La figura 2.5 muestra un grafo en el que se pueden ver los componentes de los que dispone RoboComp y las dependencias entre ellos.

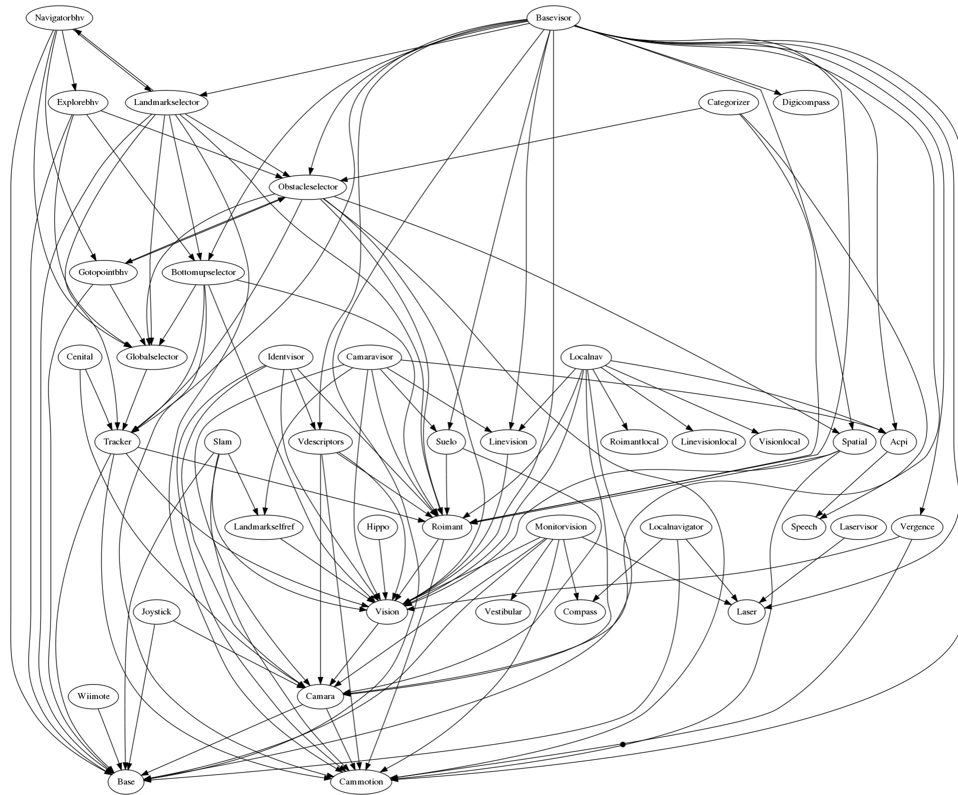

Figura 2.5: Grafo de componentes presentes en RoboComp.

2.4. Entorno de desarrollo

En esta sección se detalla el entorno de desarrollo usado, entendiendo como entorno de desarrollo el conjunto de herramientas y bibliotecas usadas.

IPP/FrameWave

IPP y FrameWave son dos bibliotecas para procesar señales de una y dos dimensiones. La elección de estas biblotecas frente a otras de similar objeto se tomó por su gran eficiencia. Ambas biblotecas son competencia directa, de

hecho, mientras IPP es desarrollada por Intel, FrameWave es desarrollada por AMD. La diferencia en el rendimiento respecto a sus competidores radica en que hacen uso de las instrucciones SIMD que aportan las extensiones de x86, 3DNow!, MMX y SSE y derivadas que ambos fabricantes incluyen en sus procesadores.

Ambas bibliotecas ofrecen APIs muy similares, pero se diferencian en la licencia bajo la que se distribuyen: mientras FrameWave se distribuye bajo la licencia libre *Apache 2.0*, IPP es software privativo. A pesar de que es gratuito para el uso personal bajo GNU/Linux, hay que pagar la licencia si se desea usar comercialmente, y su código fuente no es público.

La eficiencia del software desarrollado se debe en gran parte a la de estas bibliotecas. De ellas se han usado multitud de funciones para el filtrado, cambio de perspectiva, y el análisis estadístico de las imágenes recibidas de las cámaras.

Qt4

Qt4 es un framework de desarrollo cuyo objeto principal es el desarrollo de interfaces gráficas. A pesar de que sea este su principal uso, engloba una gran cantidad de funcionalidades distintas: interfaz multiplataforma con el sistema operativo (sistema de ficheros, procesos, hilos entre otras cosas), comunicación por red mediante sockets, interfaz con OpenGL, conexiones SQL, renderizado de HTML, y módulos de reproducción y streaming multimedia.

Atendiendo al lenguaje de programación, Qt está escrita en C++, pero existen multitud de bindings que hacen posible su uso desde otros lenguajes como Java, C#, Python, Perl y otros muchos.

Inicialmente fue desarrollada por Trolltech, una empresa noruega, bajo una licencia privativa. Después de varios cambios en la política de licencias pasó a distribuirse bajo una doble licencia GPL/QPL (esta última privativa). Finalmente, tras la compra de Trolltech por parte de Nokia, Qt fue licenciada bajo LGPL en 2009, eliminando así cualquier debate sobre la licencia de la biblioteca.

Ice

Ice es el middleware del que se habló en la sección anterior que permite crear componentes software. Es usado por RoboComp y, por tanto, por los componentes desarrollados en el proyecto de fin de carrera.

Es el principal producto de la empresa estadounidense ZeroC, y se distribuye bajo una doble licencia GPL+privativa para habilitar que las empresas desarrollen software privativo con Ice, a cambio del pago de una licencia. Las principales características de Ice frente a otras tecnologías similares es su rapidez, baja latencia y escalabilidad.

Uno de los problemas a resolver a la hora de crear componentes software es la creación de un lenguaje de definición de interfaces. Ice usa Slice, un lenguaje que se creo especialmente para este propósito.

CMake

CMake es una aplicación que permite generar automáticamente ficheros Makefile y ficheros de proyecto de varios IDE como KDevelop o Eclipse entre otros. CMake permite delegar la creación de ficheros Makefile, consiguiendo un resultado multiplataforma y robusto, sin llegar a perder el control del proceso de compilación.

CMake es una iniciativa libre que nació como respuesta a la ausencia de una alternativa suficientemente buena durante el desarrollo de una librería llamada ITK. Si bien dispone de soporte para Qt y algunas otras extensiones, es muy simple hacer nuevas extensiones. Por ejemplo, durante el desarrollo del proyecto he hecho una extensión para soportar la integración de los ficheros Slice de Ice.

Kate

Kate es un editor de texto, distribuido bajo la licencia GPL. A pesar de ser un editor avanzado no llega a ser un IDE, pero sí ofrece características de realce de código o ayuda a su escritura, incluso un sistema de plugins para extender las funcionalidades que trae por defecto.

managerComp

La aplicación managerComp permite visualizar, tanto gráficamente como en una lista, el estado de los componentes configurados en tiempo real. A pesar de estar integrado en RoboComp, se detalla independientemente por no ser un componente propiamente dicho. Además de la visualización del estado de los componentes, también permite arrancarlos y apagarlos. La figura 2.6 muestra la interfaz de managerComp. Fue creado por mí antes de comenzar el proyecto de fin de carrera. Al igual que el resto de herramientas de RoboComp, se distribuye bajo licencia GPL.

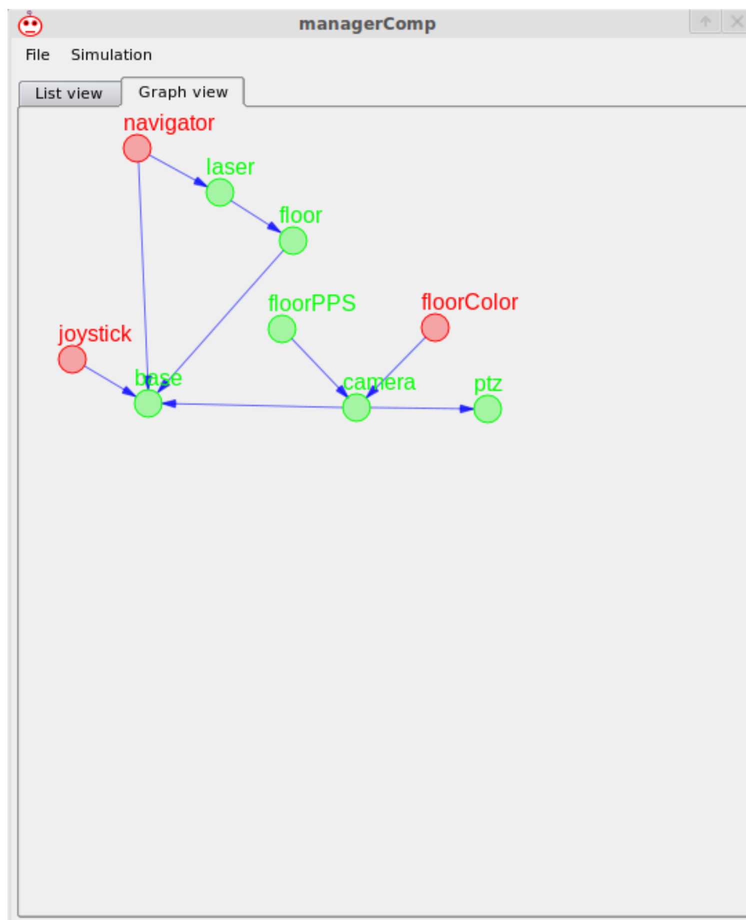

Figura 2.6: Captura de pantalla de managerComp. En la vista en modo de grafo de la aplicación se muestran los componentes monitorizados, en concreto, los relacionados con este proyecto de fin de carrera.

OpenSSH

OpenSSH es una impementación libre del protocolo SSH (Secure SHell) que ofrece tanto la parte del servidor como la del cliente. Esta herramienta es importante en la puesta en marcha de proyecto porque, junto a managerComp, nos permite ejecutar rápida y remotamente los componentes que se deseen. Además, gracias a que permite autenticación basada en llaves, se puede trabajar de forma segura sin necesidad de escribir la contraseña cada vez que se quiere cambiar el estado de algún componente.

OpenSSH es una iniciativa de los desarrolladores de OpenBSD. Se distribuye con distintas licencias (dependiendo de la pieza), pero a pesar de esto, todas ellas son libres, GPL o BSD.

GNU/Linux

Finalmente el sistema operativo usado durante el desarrollo y los experimentos, GNU/Linux. Gracias a él disponemos de un entorno de desarrollo gratuito y libre, herramientas de compilación, depuración, y una gran cantidad de bibliotecas complementarias. En el capítulo de puesta en marcha se supondrá instalada alguna distribución derivada de Debian.

Capítulo 3

Diseño del sistema

Desde del comienzo del desarrollo del sistema de detección de obstáculos se intuía que existían varias formas de tratar el problema de la detección de obstáculos, y que se podían integrar varios clasificadores simples en uno que obtuviese mejores resultados. Se planteó si este camino podría servir en robótica, no sólo para conseguir una clasificación superior a la de cualquier clasificador por separado, sino para cualquier otra tarea de detección o percepción.

Con el inicio del proyecto de fin de carrera se trató de indagar en la idea y se buscaron técnicas complementarias. Se analizaron los diferentes enfoques conocidos para disponer de ellos en el sistema:

- Técnicas basadas en flujo óptico denso.
- Técnicas basadas en flujo óptico no denso.
- Técnicas basadas en reproyección.
- Técnicas basadas en análisis de textura.
- Técnicas basadas en análisis del color.

Las técnicas basadas en flujo óptico denso son eficaces, pero son computacionalmente prohibitivas en un sistema que debe trabajar en tiempo real. Estas técnicas consisten en buscar la correspondencia entre todos los píxeles posibles de las dos imágenes que recibe el robot. Una vez se tienen las correspondencias

entre píxeles se comprueba si la posición de los píxels y su disparidad se ajustan al modelo que se tiene del suelo.

Las técnicas basadas en flujo óptico no denso tratan de conseguir un resultado lo más parecido a las densas, pero reduciendo drásticamente la cantidad de cómputo a realizar. Se buscan regiones de interés y se buscan correspondencias sólo para estas regiones. En caso de que se disponga de la configuración de las cámaras, la búsqueda de correspondencias queda reducida a una línea sobre la otra imagen. Lo propuesto en [2] se incluye en esta categoría.

Las técnicas basadas en reproyección reducen aun más el espacio de búsqueda. Si además de la configuración de las cámaras se conoce la posición del punto en el espacio puede averiguarse exactamente qué posición debería ocupar en cada una de las imágenes. Como se verá más adelante en profundiada, en esto se basa el componente floorPPSComp[10]. Lo propuesto en [3] y [19] también se incluye dentro de esta categoría.

Las técnicas basadas en análisis de textura analizan toda la imagen y realizan una clasificación suelo-obstáculos en función de las texturas presentes y cómo éstas varían en la imagen. Si bien podrían aportar información valiosa, no sería del todo determinante y la cantidad de cómputo necesario es, generalmente, considerablemente alta.

Las técnicas basadas en análisis de color, al igual que las basadas en textura, no aportan información totalmente fiable. Incluso en el caso de que el color del suelo y los obstáculos sea totalmente distinto y fácil de diferenciar (caso ideal para estos clasificadores [18, 3]), una hoja de papel de color provocaría un falso positivo. Aun así, se ha decidido usar por ser un buen complemento para el clasificador floorPPSComp, ya que éste no puede detectar obstáculos grandes que a la vez no sean texturados. Lo propuesto por [18] se incluye dentro de esta categoría.

3.1. Descripción global del sistema

En esta sección se describen los componentes que forman parte del sistema de detección de obstáculos, tanto los que han sido creados dentro del proyecto

como los que ya existían en RoboComp y han sido reutilizados. En concreto, los componentes creados han sido floorPPSComp, floorColorCueComp y floorComp. En otros componentes se han realizado cambios o mejoras, pero el desarrollo principal del resto de componentes involucrados ha sido realizado por el equipo de Robolab.

Gracias al uso de la programación orientada a componentes, se puede conseguir tener una visión global del sistema bastante fiel a la realidad sólo con ver el grafo de componentes. En la figura 3.2 se puede observar el grafo. En ella, los componentes desarrollados a lo largo del proyecto se encuentran sombreados en color azul. La dirección de los enlaces hace referencia a la dependencia existente para el funcionamiento, no al sentido en el que se establece la comunicación.

El nodo *Camera Motion* representa a camMotionComp. El objeto de este componente es mover la torreta del robot e informar de la posición de los motores que controla. El nodo *Camera* hace referencia a una instancia del componente cameraComp. Como se puede ver en el grafo, depende de camMotionComp, (lo llama con el fin de obtener la información de la posición de las cámaras). La tarea que realiza camaraComp es capturar video en tiempo real y ofrecer las imágenes junto con la información más reciente que obtenga de la torreta y la odometría. De este componente obtienen la información los clasificadores y el visor de imágenes, representado por el nodo *Visor*.

En la sección anterior se introdujeron los clasificadores. Éstos están representados en el grafo como *Color Classifier* y *PPS Classifier*. Cada uno manda su clasificación a floorComp (representado como *Floor*) y este se encarga de integrarla. En el capítulo 4 se detalla el funcionamiento de ambos componentes.

Finalmente el navegador (*Navigator*). Mediante el componente de conexión a la base, manda los comandos apropiados a los motores en función a la clasificación final construida por floorComp. Al igual que sucede con los dos últimos componentes mencionados, en el capítulo 4 se detalla el funcionamiento de floorComp.

Además de la parte algorítmica del sistema, es necesario resaltar el uso de filtros polarizadores. Estos filtros sólo dejan pasar la luz cuyo movimiento transversal se da en un ángulo determinado. Usándolos, se puede aprovechar el

fenómeno de polarización por reflexión que sufren las ondas electromagnéticas al rebotar sobre superficies brillantes. En función del ángulo con el que incida la luz en el plano de choque, y de los materiales de las fronteras (en este caso aire y suelo), existe un ángulo para el cual la polarización es total. Si la luz que provoca reflejos indeseados choca con dicho ángulo, el filtro polarizador eliminará totalmente el reflejo. A pesar de que esto se da raramente, el filtro siempre es capaz de eliminar, al menos parcialmente, los reflejos. En la figura 3.1 se pueden ver dos fotografías de filtros polarizadores.

Figura 3.1: A la izquierda se ve un filtro polarizador de montura C. Este tipo de filtro es el que se suele usar en aplicaciones científicas e industriales. A la derecha, un filtro polarizador del tipo que se suele usar comúnmnete en fotografía. En los experimentos se usaron dos filtros de este tipo.

En el resto de la sección se tratará cada uno de los componentes restantes, más detalladamente pero desde un punto de vista conceptual. Para cada uno de ellos se hará una descripción de su fin y del origen de los datos con los que trabaja. Además, gracias al uso de programación orientada a componentes, se usarán los ficheros Slice que definen sus interfaces, ya que son una buena manera de entender el rol que tiene cada uno dentro del sistema.

3.2. baseComp

El componente baseComp, que fue uno de los primeros componentes en entrar en el repositorio de RoboComp, es el que se encarga de hacer de interfaz

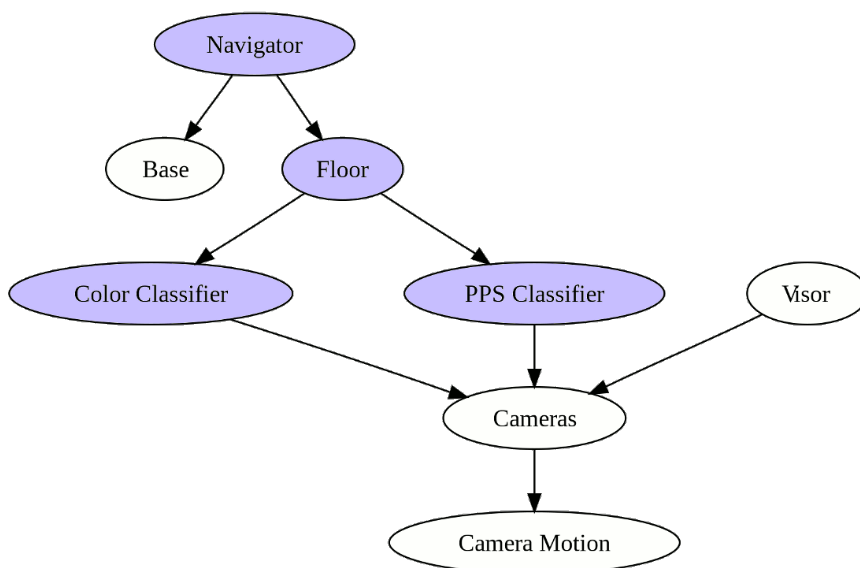

Figura 3.2: Grafo de componentes en el que aparecen los componentes desarrollados en el proyecto (en azul) junto con los que necesarios que ya existían en RoboComp (en blanco).

con la base del robot físicamente. La conexión al robot RobEx se realiza mediante una conexión RS-232 sobre USB. El comportamiento del componente queda definido por las invocaciones remotas que reciba mediante su interfaz Ice y la información del robot que lea del puerto serie. La interfaz permite leer la posición y velocidad de las ruedas de la base, así como solicitar la modificación de dichos valores de diferentes formas mediante la invocación de sus métodos remotos.

Dado que el componente fue creado previamente y no forma parte del proyecto, no se ha considerado apropiado explicar su funcionamiento interno. Lo importante es conocer su rol dentro del sistema. La interfaz Ice y los datos con los que trabaja el componente baseComp se definen en su correspondiente fichero Slice:

```
1  module RobolabModBase
2  {
3    exception HardwareFailedException{ string what; };
4    exception OutOfRangeException{ string what; };
5
```

```
6     struct TBaseState
7     {
8       float x;    // Sideways displacement in mm
9       float z;    // From displacement in mm
10      float alfa; // Angle rotated since last reset in rads
11      float advV; // Current advance speed
12      float rotV; // Current rotation speed
13      float adv;  // Incremental measures
14      float rot;
15      bool isMoving;
16    };
17
18    interface Base
19    {
20      // Get Base local state
21      void getBaseState( out TBaseState state );
22      // Set base advance and rotation speed
23      idempotent bool setSpeedBase( float adv , float rot);
24      // Set base position to (x,y) local coordinates at v speed en mm/sg and rads/sg resp.
25      idempotent bool setPosBase( int x , int y , float alfa, float adv , float rot );
26      // Stop the base
27      bool stopBase();
28      // Reset Odometer
29      idempotent bool resetOdometer();
30      // Set Odometer to given value
31      idempotent void setOdometer( TBaseState state );
32      // Set wheels speed
33      idempotent bool setSpeedWheels( float left , float right );
34      // Get wheels speed in rads/sg
35      idempotent bool getSpeedWheels( out float left , out float right );
36      // Set wheels at (left,right) position in rads going at vi,vr speed in rads/sg
37      idempotent bool setPosWheels( int left , float vi, int right , float vr);
38      // Get wheels pos
39      idempotent bool getPosWheels( out int left , out int right );
40      // Set the base motors break state to normal operation so the robot can move.
41      idempotent bool setMotorsStateNormal();
42      // Set the base motors break state to free ( free wheel movement )
43      idempotent bool setMotorsStateFree();
44      // Set the base motors break state to break ( wheels can't move )
45      idempotent bool setMotorsStateBreak();
46    };
47  };
48
```

El componente es llamado por camaraComp para poder adjuntar la información de la odometría a las imágenes y por localNavigatorComp. Como veremos, este último es el componente que controla el movimiento del robot.

3.3. camMotionComp

Este componente, al igual que baseComp, también se reutiliza y no ha sido desarrollado dentro del proyecto. Ofrece una interfaz de control para una torreta estéreo, en concreto una de tilt común y pan independiente en cada cámara. Su objetivo es mover los servos de la torreta tal y como se le ordene, así como responder a las consultas de la configuración de la torreta (las posiciones de los servos que controla).

Ha sido programado para ser compatible con distintos tipos de servos digitales: Diolan, Dynamixel y Megarobotics. Además, la gama de servos con la que trabaja se puede extender fácilmente con la implementación de una clase virtual que se creó para ello.

```
1  #ifndef CAMMOTION_ICE
2  #define CAMMOTION_ICE
3
4  module RobolabModCamMotion
5  {
6    exception HardwareFailedException{ string what; };
7    exception OutOfRangeException{ string what; };
8
9    struct TMotorRanges // In Radians
10   {
11     float min;
12     float max;
13     byte number;
14   } ;
15   struct TMotorState   // In Radians
16   {
17     float pos;
18     float vel;
19     float power;
20     int p;  // in steps
21     int v;
22     bool isMoving;
23   } ;
24   struct THeadRanges   // In Radians
25   {
26     TMotorRanges left;
27     TMotorRanges right;
28     TMotorRanges tilt;
29     TMotorRanges leftTilt;
30     int baseline;
31   };
```

```
32    struct THeadState   // In Radians
33    {
34      TMotorState left;
35      TMotorState right;
36      TMotorState tilt;
37      TMotorState leftTilt;
38      bool isMoving;
39    };
40    struct TParams    //Configuration Params
41    {
42      int TILTMOTOR;
43      int LEFTMOTOR;
44      int RIGHTMOTOR;
45      int LEFTCAMERA;
46      int RIGHTCAMERA;
47      int BOTHCAMERAS;
48      int RIGHTZEROPOS;
49      int LEFTZEROPOS;
50      int TILTZEROPOS;
51      string device;   //Communications port
52      string handler;  //Drive for servomotor hardware
53      int baseline;
54      bool tiltInvert;
55      bool leftInvert;
56      bool rightInvert;
57    };
58
59    interface CamMotion
60    {
61      void resetHead();                   // Send cameras to ZEROPOS and set zero speed.
62      void stopHead();                    // Stop head where it is now
63      void setPanLeft(float pan);         // Set PanI servo to pan rads
64      void setPanRight(float pan);        // Set PanD servo to pan rads
65      void setTiltLeft(float tilt);       // Set Tilt servo to tilt rads
66      void setTiltRight(float tilt);      // Set Left Tilt servo to tilt rads
67      void setTiltBoth(float tilt);       // Set both cameras to tilt rads
68      void getMotorRanges(byte motor, out TMotorRanges info);
69      void getMotorState(byte motor, out  TMotorState state);
70      void getHeadState(out THeadState state);
71      void getHeadRanges(out THeadRanges ranges);
72
73      void setRadSaccadic(float pan, float tilt, int cam);
74
75      bool isMovingMotor(byte motor);
76      bool isMovingHead();
77    };
78  };
79
80  #endif
```

Dentro del sistema, camMotionComp se encarga únicamente de realizar el control de los servos y atender a las peticiones de información de camaraComp. Dentro del sistema no hay ningún componente que mueva las cámaras, pero sí es posible que algún componente ajeno al sistema de detección lo haga. El objetivo es que los componentes que lo necesiten conozcan la configuración de la torreta, independiendemente de cual sea.

3.4. camaraComp

El componente camaraComp es otro de los primeros componentes de RoboComp. Su principal función es capturar vídeo y atender a las peticiones de imágenes que otros componentes realicen. Además, ha de adjuntar a las imágenes la configuración de la posición que tenía la torreta y el estado de la odometría cuando la imagen fue tomada. Tiene capacidad tanto para mandar imágenes de una cámara, como de dos a la vez.

Para evitar problemas de sincronización debidos a la latencia de la comunicación, cuando trabaja con dos cámaras, puede devolver ambas imágenes simultaneamente en una única llamada. Esta característica es básica para trabajar con visión estereoscópica. Otra de las características interesantes de camaraComp es que permite trabajar con distintos tipos de cámaras: v4l2 (Video For Linux v2), IEEE 1394 (FireWire), o mediante una tubería Unix hacia MPlayer (lo que además de extender aun más el abanico de cámaras soportadas, permite la captura de imágenes desde ficheros de vídeo).

```
1   #ifndef CAMARA_ICE
2   #define CAMARA_ICE
3
4   #include <CamMotion.ice>
5   #include <Base.ice>
6
7   module RobolabModCamara {
8     exception HardwareFailedException { string what; };
9     exception MovingImageException { string what; };
10
11    sequence<byte> imgType;
12    sequence<int> intVector;
13
```

```
14  struct TCamParams {
15    string name;
16    string driver;
17    string device;
18    string mode;
19    int focal;
20    int width;
21    int height;
22    int size;
23    int numCams;
24    int FPS;
25    int timerPeriod;
26    int leftCamera;
27    int rightCamera;
28    int bothCameras;
29    int inverted;
30    int leftInverted;
31    int rightInverted;
32
33    int saturation;
34    int lineFreq;
35
36    bool talkToBase;
37    bool talkToCamMotion;
38  };
39
40  interface Camara {
41    // YUV420 format - 2 planes unpacked
42    idempotent void getYUVImage(int cam, out imgType roi,
43     out RobolabModCamMotion::THeadState hState,
44     out RobolabModBase::TBaseState bState
45     ) throws HardwareFailedException;
46
47    // Luminance - 1 Plane
48    idempotent void getYImage(int cam, out imgType roi,
49     out RobolabModCamMotion::THeadState hState,
50     out RobolabModBase::TBaseState bState
51     ) throws MovingImageException;
52
53    // Luminance in LogPolar
54    idempotent void getYLogPolarImage(int cam, out imgType roi,
55     out RobolabModCamMotion::THeadState hState,
56     out RobolabModBase::TBaseState bState
57     ) throws MovingImageException;
58
59    // Luminance - 1 Plane. Compressed and resized.
60    idempotent void getYImageCR(int cam, int div, out imgType roi,
61     out RobolabModCamMotion::THeadState hState,
62     out RobolabModBase::TBaseState bState
```

```
63        ) throws MovingImageException;
64
65       // RGB packed for visualization - 3 planes
66       idempotent void getRGBPackedImage(int cam, out imgType roi,
67        out RobolabModCamMotion::THeadState hState,
68        out RobolabModBase::TBaseState bState
69        ) throws MovingImageException;
70
71       // Lum + RGB - 4 planes unpacked
72       idempotent void getYRGBImage(int cam, out imgType roi,
73        out RobolabModCamMotion::THeadState hState,
74        out RobolabModBase::TBaseState bState
75        ) throws MovingImageException;
76
77       // Return relevan comp params
78       TCamParams getCamParams();
79
80       // Inner feeding of images
81       idempotent void setInnerImage(imgType roi);
82     };
83   };
84
85   #endif
```

Todos los componentes del sistema trabajan con imágenes en color, y no se usan otras capacidades del componente que no sean las de captura de imágenes.

El único método de camaraComp invocado por otros componentes del sistema es getRGBPackedImage(). Como se puede ver, el método devuelve la/las imagen/imágenes con la información de la configuración de la torreta y la odometría en una sola llamada. Esto hace que la mayoría de los componentes no necesiten conectarse, ni a baseComp, ni a camMotionComp.

3.5. floorColorCueComp y floorPPSComp

Respecto a lo necesario para entender el funcionamiento global del sistema, lo importante de floorPPSComp y floorColorCueComp es que cada uno de los componentes obtiene una clasificación diferente en función a las imágenes que obtienen de camaraComp. Estos dos componentes representan gran parte del trabajo realizado en el proyecto de fin de carrera.

El primero de ellos, floorColorCueComp, utiliza únicamente la información proveniente de una de las cámaras. Debido a que la clasificación sólo se realiza

en el espacio binocular (el espacio capturado por las dos imágenes) y que el clasificador no hace uso de información estéreo, una única imagen es suficiente (la intersección de dos conjuntos se encuentra necesariamente en cualquiera de los dos conjuntos). Para cada píxel de la imagen capturada se comprueba si su color pertenece al modelo de lo que se considera suelo. Si y sólo si es así, se clasifica como suelo por este clasificador. El modelo de color de suelo es un histograma de los colores encontrados en el suelo durante una fase de entrenamiento. En el siguiente capítulo se verá a fondo el proceso de clasificación de floorColorCueComp.

El segundo, de ellos usa la información de ambas cámaras, así como la que camaraComp obtiene de camMotionComp. Dado que se trabaja con información estereoscópica, este componente sí solicita a camaraComp las dos imágenes de la torreta. Con esta información, el componente hace una predicción de cómo se vería desde el punto de vista de la cámara derecha lo que se ve en la camara izquierda (suponiendo que todo lo que ve es suelo). Que la predicción falle para alguna zona de la imagen (que no sea certera), significa que lo que se esté viendo en dicha zona no forma parte del suelo. El proceso de clasificación de floorPPSComp se detalla en profundidad en el siguiente capítulo.

Los componentes floorColorCueComp y floorPPSComp no ofrecen ningún método remoto, por tanto, no se muestran sus ficheros de interfaz. En cambio, estos componentes, toman un papel activo y envian su información a floorComp. Esto se ha decidido así porque el cómputo que realizan supone la mayor parte de la carga de trabajo del sistema de detección de obstáculos. De esta forma, cada vez que obtienen nuevos resultados se hacen efectivos sin necesidad de hacer “polling”.

3.6. floorComp

El componente floorComp es el tercer componente creado en el proyecto. Usa las dos clasificaciones de floorColorCueComp y floorPPSComp para construir una definitiva. En el uso de dos clasificaciones de distinta naturaleza para una misma tarea es donde estriba la bondad del sistema de detección. Cada com-

ponente analiza información de distínta índole. El componente floorPPSComp se encarga de información geométrica, que es la más fiable, pero que en muchas ocasiones no está disponible. En cambio, floorColorCueComp se engarca de analizar la información cromática, que a pesar de que no es fiable, siempre está disponible y puede ayudar en caso de duda.

```
1  #ifndef FLOOR_ICE
2  #define FLOOR_ICE
3
4  module RobolabModFloor
5  {
6    struct FloorParams
7    {
8      int steering;
9      int width;
10     int height;
11   };
12
13   sequence<byte> floorMatrixType;
14
15   interface Floor
16   {
17     void getFloorMatrix(out floorMatrixType floorImage);
18     void setFloorChannel(int channel, floorMatrixType floorImage);
19   };
20 };
21
22 #endif
```

3.7. localNavigatorComp

El navegador local, localNavigatorComp, es el componente superior en la jerarquía. En función a la clasificación final realizada por floorComp actualiza un pequeño modelo de su alrededor (dicho modelo se vé en el capítulo 4). Con el mencionado modelo y la posición del punto destino, el componente selecciónа una dirección y una velocidad que manda a la base del robot hasta llegar a dicho punto. El destino debe ser alcanzable localmente, es decir, la trayectoria a describir hasta llegar al punto ha de ser relativamente simple. Esto es así porque el objeto del sistema desarrollado es la navegación local, para alcanzar puntos lejanos se han de usar otras técnicas basadas en *Visual SLAM* (ver [5]).

3.8. Diseño genérico de un componente

El diseño de un nuevo componente suele ser siempre muy parecido, de hecho, cuando se incluye un nuevo componente en RoboComp se usa generalmente una herramienta que lo genera automaticamente. El generador de código también incluye el código necesario para crear una conexión a los componentes que se le especifiquen. A los componentes que se generen automáticamente se le pueden añadir después todas las clases que sean necesarias. Además, muchas veces se desea añadir en el fichero de configuración algún parámetro y modificar el fichero de interfaz por defecto (que no ofrece ningún método). De no usar el generador automático de código, la creación de componentes pequeños sería bastante tediosa ya que la relación entre las líneas de código Ice y las propias de la tarea del componente sería considerablemente grande.

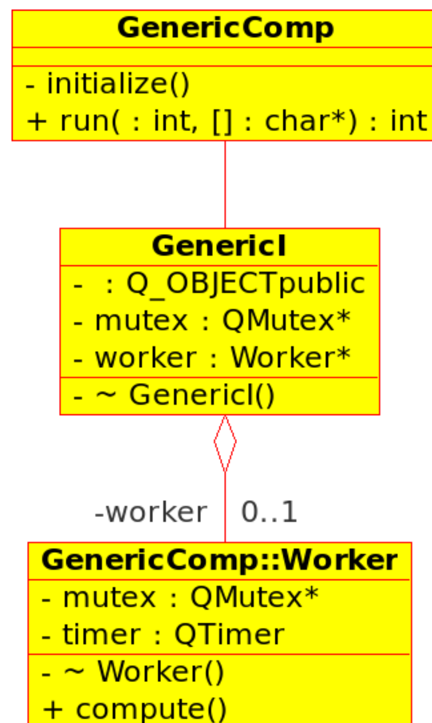

Figura 3.3: Diagrama de clases del componente genérico genericComp.

Como se puede observar en la figura 3.3, a parte del código de programa principal, el componente plantilla consta de tres clases: GenericComp, GenericI y Worker. Cuando se usa el generador de código éste modifica los nombres de las

clases GenericComp y GenericI en función al nombre del componente cambiando *Generic* por lo que corresponda en función al nombre del componente.

Cada componente consta de, al menos, dos hilos de ejecución, siendo estos el hilo principal y un hilo separado para responder a las llamadas Ice. El último de estos dos hilos se crea cuando se instancia una interfaz Ice (a la clase creada se le tiene que pasar como parámetro la clase que se hace cargo de las llamadas). El hilo principal de ejecución se encarga de la parte activa del componente, hacer las llamadas Ice necesarias si las hay, y los cálculos asociados al comportamiento del componente. Para facilitar la comprensión del código la clase GenericComp delega en Worker todo el trabajo y se ve GenericI como una clase que, de forma transparente, responde a las llamadas remotas.

Las clases Worker y GenericI, que como ya hemos visto se encargan de la parte *interna* y *externa* respectivamente, se ejecutan en exclusión mutua gracias a un mutex que comparten. El bloqueo del mutex se realiza cada vez que GenericI recibe una llamada o que Worker va a cambiar alguna parte de su estado que pueda modificar las respuestas de GenericI a sus invocaciones remotas.

Como se puede ver en el diagrama, GenericI tiene acceso a la clase Worker. Dicha condición, que sólo se da en ese sentido, es necesaria porque generalmente toda la información relativa al estado del componente se guarda dentro de Worker.

Capítulo 4

Algoritmos e implementación

En los capítulos previos se ha introducido el problema, se han planteado las formas conocidas de resolverlo, los objetivos del proyecto, y el punto del que se partía. Finalmente, en el capitulo anterior, se ha descrito globalmente un enfoque que, como muestran los resultados experimentales, mejora los enfoques previos similares. En este capítulo se describen los algoritmos que rigen el comportamiento de los componentes que forman el sistema descrito.

El resto del capítulo se divide en secciones, una para cada componente. En cada una de ellas se hará una descripción lo suficientemente profunda como para que, sin disponer del código del componente, se pueda implementar. Para ello, en alguos casos, se describen mediante pseudocódigo los algoritmos usados.

4.1. floorPPSComp

Como se adelantó, el detector de obstáculos floorPPSComp hace la clasificación en base a la información geométrica que, gracias a sus dos cámaras, puede deducir. La clasificación se basa en la idea de que el suelo es aproximadamente plano. Dando esto por supuesto, cosa que también se da por sentado en el resto de los clasificadores basados en la geometría del par de cámaras y el suelo conocidos, el suelo induce una relación de homografía entre las imágenes de las dos cámaras del robot. Una homografía es una transformación geométrica que establece una correspondencia entre las posiciones de los píxeles en las imágenes suponiendo que lo que se ve en las imágenes es un plano conocido.

Dada la posición de un píxel, imagen de un punto de ese plano común dentro de una cámara, las coordenadas del mismo punto del plano en la imágen de la otra cámara se pueden calcular premultiplicando las coordenadas del píxel en la imagen inicial por la *matríz de homografía*:

$$x' = Hx,$$

De esta forma, para cada píxel de la imagen de una cámara, se puede calcular en qué posición se debería encontrar en la otra cámara si el punto perteneciese al plano. Dicho de otra forma, la homografía nos permite estimar cómo se vería un plano desde otro punto de vista cualquiera.

Para poder hallar la matriz de homografía (asociada a un cambio de perspectiva concreto de un plano en concreto) es necesario conocer los parámetros extrínsecos de las cámaras y la posición del plano respecto a la cámara original. Además, también es necesario conocer los parámetros intrínsecos en el caso de que las cámaras sean diferentes. Conociendo estos parámetros, la matriz de homografía se calcula de la siguiente manera [7]:

$$H = K'(R - tn^T/d)K^{-1},$$

donde:

- K y K' son las marices de parámetros intrínsecos de las cámaras de partida y destino, respectivamente.
- R y t son la matriz de rotación y el vector de traslación que llevan desde el centro de proyección de la cámara de partida hasta el de la de destino.
- n es el vector perpendicular al plano que induce la homografía, normalizado.
- d es la mínima distancia que hay desde el punto de vista de partida hasta el plano.

Dado que en el proyecto se toma como perspectiva inicial la cámara izquierda y como final la cámara derecha, esto es, las imágenes se transfieren desde la perspectiva de la cámara izquierda a la de la derecha, se entenderá *imagen o*

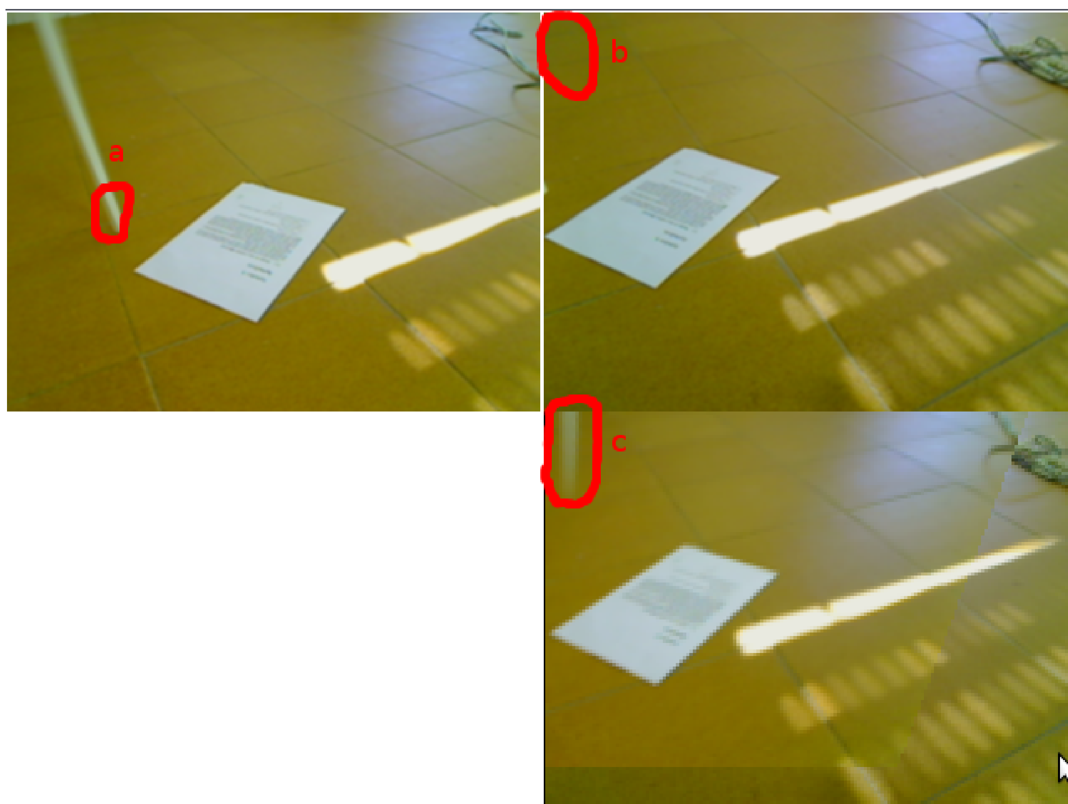

Figura 4.1: Imágenes del par estéreo acompañadas del warping de la cámara izquierda. Como se puede ver, sólo aquellas zonas de la imagen pertenecientes al suelo transfieren bien a la imagen esperada. Este ejemplo muestra en la imagen izquierda cómo la parte inferior de la varilla (a) no transfiere a la imagen prevista como debería. En la imagen de la cámara derecha no aparece (b) y, sin embargo, en el resultado del warping sí (c).

cámara inicial, *imágen o cámara izquierda*, *imágen o cámara origen*, o *imágen o cámara primera*, como el mismo concepto, y viceversa. En la figura 4.1 se puede ver un par de imágenes de la torreta estéreo junto con el resultado del cambio de perspectiva.

Si las cámaras del robot son estáticas, o si su único grado de libertad es un movimiento de rotación sobre el eje perpendicular al suelo, ningún movimiento que haga el robot o su torreta cambiará la posición relativa entre cámaras, ni entre el suelo y las cámaras, ni por tanto, la matriz de homografía. Esto nos lleva a dos conclusiones, la primera de ellas es que no es necesario calcular la

homografía en tiempo real, sino sólo una vez. La segunda es a su vez derivada de la primera conclusión, al ser necesario calcularla sólo una vez se puede calibrar mediante técnicas que no necesiten conocer los parámetros del par de cámaras. Para ello hay diversas técnicas, [20, 15, 1, 6] entre otras muchas. Durante el desarrollo del proyecto se implementó y probó un algoritmo inspirado en [15] que en algunas ocasiones consigue una calibración casi perfecta en menos de 20 segundos con sólo 5 pares de imágenes. Esto se ve también más adelante en el capítulo dedicado a los experimentos.

La clasificación suelo-obstáculo es llevada a cabo mediante la simulación del cambio de perspectiva (*warping* en inglés) de una de las cámaras hacia la perspectiva de la segunda, y comparando el resultado de este paso con la imagen que se está realmente capturando. En condiciones ideales, suponiendo que el modelo de cámaras *pin-hole* es perfecto, cada píxel de la segunda imagen debería ser exactamente igual que el píxel que está en la misma posición en el resultado del cambio de perspectiva. Sin embargo, bajo condiciones reales tenemos que solucionar los siguientes problemas:

- Reflejos de la luz en el suelo.
- Mala o inexistente sincronización de las cámaras.
- Diferente respuesta de las cámaras a un mismo color o intensidad.
- Rugosidades o deformaciones en el plano del suelo.
- Holgura o incertidumbre en la configuración real de la torreta.
- Estimaciones pobres de la matriz de homografía.

En [3], que representa el enfoque más cercano al del proyecto, los píxeles se clasifican como obstáculo si la diferencia entre los valores de luminancia son mayores que un umbral, comparando los píxeles que están en la misma posición de las imágenes que deberían ser iguales en caso de ser todo suelo: la de la cámara izquierda cambiada de perspectiva y la de la derecha.

El resultado de este método depende demasiado de las condiciones en las que se desenvuelva el robot. Para que funcione bien se necesita que la estimación de

la matriz de homografía sea muy precisa, que no haya casi reflejos en el suelo y que las condiciones de luz en las dos cámaras y sus respuestas sean semejantes. Si alguna de estas premisas falla, el algoritmo tendrá muchos falsos positivos y, posiblemente, algunos falsos negativos. Realmente, es muy difícil tener una calibración perfecta de la matriz de homografía (más aun si se estima en tiempo real), por lo que es muy común que este algoritmo de falsos positivos en los bordes de los objetos. Debido a los errores que obtiene el enfoque propuesto en [3] y que se pueden ver en 4.2, se ha propuesto una segunda fase en el proceso de clasificación.

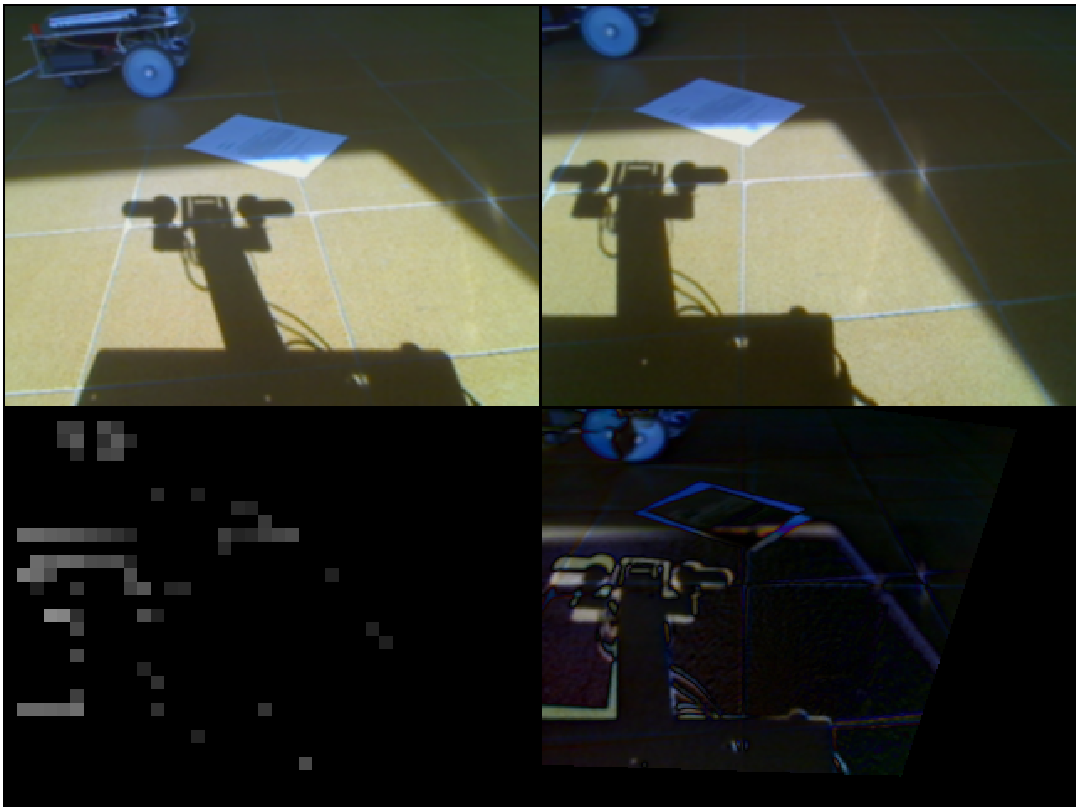

Figura 4.2: Imágenes derecha e izquierda transformada, acompañadas del valor absoluto de su resta. Se puede observar que esta última imagen raramente será útil directamente si la homografía no está muy bien calibrada. En este caso la propuestas vista en [3] fallaría.

La primera fase es similar a lo propuesto en [3], pero no tiene como objetivo

realizar la clasificación final, sino descartar las zonas de imágen que claramente no corresponden a un obstáculo. Para conseguir este efecto, el umbral de diferencia, a partir del cual un píxel se considera obstáculo es algo menor. Con esto se consigue reducir al máximo posible los falsos negativos a costa de tener muchos falsos positivos, que quedan pendientes para su descarte en la segunda fase del algoritmo. Aparte de esto, el algoritmo se diferencia también en dos aspectos más. El primero es que no se compara píxel a píxel, sino por ventanas (particiones imaginarias de la imagen). El segundo es que no se compara la luminancia, sino cada uno de los tres colores (rojo, verde, azul) por separado. Si alguno de los colores supera el umbral en la ventana, la ventana se considera candidato a obstáculo. Los valores de color de la ventana se obtienen mediante interpolación lineal.

En la segunda fase, a cada una de las ventanas candidatas se le pasa un segundo test. Este segundo test consiste en comparar la salida máxima de la correlación cruzada normalizada para cada uno de los tres canales de color respecto a una ventana más grande de la imagen complementaria. Si el máximo de esta operación para alguno de los canales es inferior a un umbral, la ventana no pasa el test y se clasifica finalmente como obstáculo por floorPPSComp. El uso de la información de color en ambas fases nos permite diferenciar los distintos colores a los que les corresponde la misma luminancia. En la segunda fase también se usa un umbral relativamente bajo.

La idea interesante es el uso de dos umbrales distintos que miden información de distinta índole y que por separado son ligeramente permisivos. Esto obliga a que las ventanas clasificadas como obstáculo tengan que dar positivo en las dos pruebas, con ello se consigue disminuir los errores. Los falsos positivos bajan porque hay un doble umbral, que aunque es poco restrictivo por separado, hace necesario que los falsos positivos lo sean en ambas pruebas (cosa que muy pocas veces se da). Los falsos negativos bajan porque si corresponden realmente a un obstáculo es poco probable que, siendo obstáculo, queden por debajo del umbral de alguna de las fases.

Dado que sólo se aplica el segundo test a las ventanas candidatas a la clasificación como obstáculo, las mejoras introducidas por la segunda fase no suelen

acarrear un aumento considerable en el cómputo necesario. Al respecto, hay que tener en cuenta que, mientras no haya ventanas candidatas, el incremento del coste computacional será cero ya que no se aplicará a ninguna ventana el segundo test. Por contra, si todas las ventanas son candidatas (por ejemplo, cuando el suelo está lleno de obstáculos o hay una linterna enfocando a una cámara), el test se aplicará a todas las ventanas. En cualquier caso, la complejidad será O(n) en ambos casos. Esta mejora no sólo disminuye la aparición de falsos positivos en los bordes de los objetos o en suelos texturados, que son la principal desventaja de los métodos anteriores, sino que además nos permite descartar aquellos objetos finos sobre los que el robot puede pasar y no representan un obstáculo real.

En cualquier caso, independientemente de lo finas que sean las medidas, habrá dos homografías distintas: la homografía real que el suelo induce entre las dos cámaras, y la homografía que hemos calculado. La segunda fase nos permite trabajar con imprecisiones *relativamente* notables sin desbaratar la clasificación.

Los reflejos de la luz dependen del punto de vista de cada cámara, por lo que la misma fuente de luz provocará un reflejo en distintos puntos de la imagen (ver figura 4.1). Estos reflejos son prácticamente una constante en suelos pulidos o brillantes, que son los más comunes en entornos estructurados. A pesar de que los reflejos no pueden ser detectados como tal, el segundo test minimiza su impacto si las cámaras no están muy alejadas entre sí (lo que implica que la posición del reflejo en las cámaras aparece en posiciones aproximadamente cercanas). Además, para mitigar más aun este efecto, se han instalado dos filtros polarizadores de luz en las cámaras, con lo que la aparición de reflejos disminuye.

Resumiendo, la principal mejora del algoritmo respecto a métodos previos es que es parcialmente inmune a homografías imprecisas debido a que, localmente, pequeñas imprecisiones en la homografía producen poca deformación afín. La búsqueda de la mejor correspondencia de las ventanas candidatas en su entorno soluciona los efectos de estas imprecisiones. Si una ventana no es un obstáculo realmente, en algún punto de su entorno obtendrá una buena correlación, por lo que se descartará. Como previamente se adelantó, estos errores en el cálculo de la homografía son muy comunes cuando se usan torretas motorizadas. Los

escenarios en los que esto suele pasar son:

- **Estimación de la configuración de la torreta errónea o con holgura:** Las pequeñas rotaciones o traslaciones de una torreta motorizada debida a holgura o pequeños impactos que descalibren la estimación pueden hacer que el ángulo estimado y el ángulo real sean distintos, aumentando la diferencia entre la homografía real y la estimada. En estos casos, otros métodos similares darán una gran cantidad de errores en suelos texturados y bordes de objetos.

- **Torretas motorizadas:** Cuando la homografía se calcula en tiempo real, se pueden obtener medidas erróneas debido tanto a problemas de hardware como de software (por falta de sincronía o cualquier otro motivo). Este problema también ocasionará estimaciones poco precisas de la homografía. Este problema, que dependiendo del hardware y el software usado podría ser muy común, haría que el clasificador fuese inútil.

- **Desincronización entre cámaras:** El mismo problema que ocasiona que los motores de las torretas no estén bien sincronizados lo ocasiona la falta de sincronía entre las dos cámaras o entre las cámaras y la lectura de posición de los motores de la torreta.

Una vez que el segundo test ha terminado, se genera una imagen binaria donde cada píxel representa cada una de las ventanas en las que se ha dividido la imagen. Evidentemente, cuanto menor sea el tamaño de las ventanas mayor será la resolución final, sin embargo no tiene mucho sentido hacer las ventanas de un píxel de ancho porque la calidad final también depende del tamaño de la ventana de correlación, siendo dicha ventana el área donde se busca la correspondencia de las ventanas candidatas. Una ventana de correlación demasiado pequeña, por otra parte, haría perder las ventajas que la segunda fase aporta. Aun así, dentro de un margen razonable para estos tamaños, el algoritmo funciona bien. En la implementación se han puesto como tamaños 8x8 a las ventanas de división y 24x24 a la ventana de correlación trabajando con imágenes de 320x240 (trabajando a 640x480 se podrían definir los tamaños como 16x16

y 48x48 o, por ejemplo, 12x12 y 48x48 para obtener más resolución en la salida del algoritmo).

Suponiendo que el punto de vista destino es la cámara derecha, el proceso se puede resumir en el siguiente pseudocódigo:

1. **Copiar** la imagen derecha, I^R, a un buffer temporal I^{T1}.

2. Usar H para **cambiar la perspectiva** de la imagen izquierda, I^L, a I^{T1}. Esto deja intactos los píxeles que no tienen correspondencia en la imágen derecha, de ahí el paso anterior.

3. Calcular el **valor absoluto de la resta** I^{T1} - I^R y guardar el resultado en I^{T2}.

4. Para cada ventana que supere el umbral:

    a) **Si** el máximo valor de la correlación cruzada normalizada en la ventana de correlación supera un umbral para todos los canales de color, la ventana se descarta y se clasifica como suelo.

    b) **Si no**, la ventana se clasifica como obstáculo.

El primer paso se toma para que la zona de la imagen que no pertenece al espacio binocular no se clasifique como obstáculo, ya que se restarían píxeles sin correspondencia con píxeles de imagen (cuyo resultado no necesariamente es cero). Al copiar la imagen con la que se va a restar posteriormente en el buffer donde se guarda el resultado del "warping", la resta en aquellos píxeles sin correspondencia será cero porque serán los mismos necesariamente. Si los puntos sin correspondencia se ignoran de alguna otra forma, el primer paso se puede saltar ya que precisamente sirve únicamente para este propósito.

Opcionalmente, si la resolución de tamaño de ventana no fuese suficiente, se puede ampliar y ejecutar una operación de tipo "flood-fill" en el centro de las ventanas que son clasificadas como obstáculos por el segundo test.

Hasta ahora se ha hablado de cómo generar la clasificación y sobre las ventajas respecto a los enfoques previos, sin embargo, evidentemente, la clasificación producida por el componente tiene sus limitaciones. A pesar de trabajar mejor

que los demás clasificadores similares es imposible, por la naturaleza del análisis hecho, detectar un obstáculo donde no hay textura o algún tipo de cambio. En tal caso sucede lo mismo que en una hipotética habitación uniformemente iluminada y pintada, sin sombras ni detalles: no se distinguiría donde empieza o acaba el suelo (no, la habitación no tiene por qué ser blanca). Dado que las cámaras del robot no tienen tanta precisión, este problema se encuentra mucho más fácilmente en mayor o menor medida (imágenes 4.6,4.3).

Figura 4.3: Situación problemática para clasificar por geometría. Hay un tabla grande sobre la pared. Dado que la tabla es más grande que el robot, el sistema detecta que no puede pasar por los extremos de la tabla. Sin embargo, en el resto de la zona de imagen ocupada por la tabla, donde no hay textura, no se puede detectar ningún obstáculo por esta vía. Esto lleva a una clasificación errónea si se usa sólo un clasificador ya que el robot no puede pasar por encima de la tabla.

4.2. floorColorCueComp

El componente floorColorCueComp es otro detector de obstáculos. En este caso, se usa información cromática de las imágenes para la clasificación. Está inspirado en el trabajo de Ulrich en [18], pero ha sido modificado para mejorar la clasificación. El entrenamiento consiste en la selección de varias regiones de imagen del suelo y el cómputo de un histograma tridimensional en base a esas

regiones de imagen. La selección de imágenes puede ser llevada a cabo manualmente por un operador humano o, bajo la condición de que todos los obstáculos del entorno sean texturados (paredes con cierta textura), automáticamente usando la información del clasificador por homografía que se ha descrito en la sección anterior.

En vez de usar dos histogramas (uno de tono y otro de saturación) unidimensionales por separado como se propone en [18], en base a su mayor poder discriminativo, se decidió usar un único histograma tridimensional. Con un histograma de tres dimensiones, suponiendo que el umbral es el óptimo, se puede distinguir cualquier región arbitraria en ese espacio, con tanta precisión como el histograma tenga. Sin embargo, con tres histogramas unidimensionales por separado (en [18] se usan dos para hacerlo totalmente invariante a la luminancia) no sucede lo mismo, únicamente se podrían describir cubos. La figura 4.4 lo expone gráficamente con un ejemplo en dos dimensiones. Además, el uso de histogramas por separado lleva a conclusiones erróneas: supongamos que se quiere clasificar como obstáculo los rojos poco saturados y los verdes muy saturados. En tal caso, con histogramas separados, cualquier combinación es válida, por lo que también se clasificaría así los verdes poco saturados y los rojos muy saturados.

En el histograma generado, dos ejes representan el rojo normalizado y el verde normalizado, y el tercero un valor aproximado de luminancia (R+G+B). Dependiendo del número de divisiones del histograma en cada eje, se obtendrá más o menos invarianza respecto a la propiedad que representa (a menos divisiones más invarianza). En particular, es deseable tener cierta invarianza a cambios en la luminancia hasta cierto punto, pero no total. Una invarianza total respecto a la luminancia nos haría perder capacidad de diferenciación.

Suponiendo que los píxeles se representan por tuplas de tres bytes RGB cuyos valores van de 0 a 255, generamos el histograma con 128 divisiones en los ejes X e Y y 16 en el Z. Por tanto, los ejes X, Y y Z corresponden a:

$$X : 128 * R * 3/(R + G + B),$$

$$Y : 128 * G * 3/(R + G + B),$$

$$Z : (R + G + B)/6$$

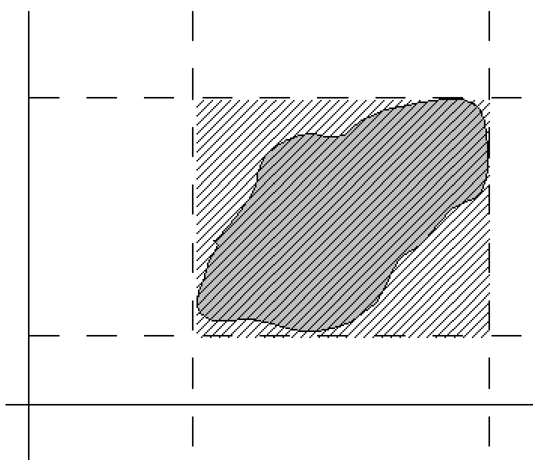

Figura 4.4: Comparación entre el poder discriminativo de un histograma de dos dimensiones y el de dos histogramas de una dimensión. Las conclusiones se pueden extender a tres dimensiones. El área rayada de la segunda figura define el área de decisión de dos histogramas unidimensionales, mientras que el área sombreada de gris, visible en ambas figuras, delimita el área definida por un histrograma bidimensional.

Una vez se ha generado el histograma se termina el entrenamiento pasándole un filtro paso bajo con una máscara 5x5x5.

La clasificación se divide también en dos fases. En la primera fase cada píxel se clasifica como obstáculo o suelo en función a la presencia de su color en el histograma. Si la presencia es menor que un umbral el píxel se clasifica como obstáculo. El umbral se puede definir como un porcentaje de las muestras del histograma, de forma que no depende del tamaño del entrenamiento.

A pesar de que la mayoría de los detectores de obstáculos basados en color usan el espacio de color HSV, se decicidió no usarlo porque los valores de color son muy inestables cuando la saturación o la luminancia son bajas[1]. Mientras que otros métodos como [18, 3] no tienen en cuenta los píxeles con baja saturación, los experimentos llevados a cabo demostraron que es una mala decisión, los píxeles cercanos al gris no deben ignorarse. El enfoque tridimensional demostró

[1] La codificación en HSV de (1,0,0) en RGB es, (0,100,0). La codificación de (0,1,0) es (120,100,0). A pesar de ser colores similares (prácticamente negro) y de tener la máxima saturación, el tono es totalmente distinto. Por tanto, el tono no es sólo inestable cuando hay poca saturación, sino también cuando la luminancia es baja.

mejores resultados en todos los experimentos llevados a cabo.

La segunda fase de la clasificación ventanea la salida de la primera fase usando un método de interpolación que divide por cuatro el tamaño de la imagen y toma como valor de la ventana el menor valor de la misma (una ventana se clasifica como obstáculo sí y sólo si todos los píxeles de la ventana han sido clasificados como tal). Este paso elimina casi todo falso positivo producido por el ruido.

La principal desventaja de esta clasificación es que es heurística. Nada impide que algún obstáculo tenga un color similar al del suelo, ni que el suelo no se vaya a manchar jamás. De guiar el robot únicamente por esta clasificación el robot se podría parar ante manchas o papeles en el suelo. Como consecuencia, en suelos de colores muy variados es imposible detectar obstáculos. No porque el componente esté mal hecho, sino porque no se puede clasificar en base al color si el color de los obstáculos y el suelo es prácticamente el mismo. Como ejemplo de estos problemas se pueden ver las figuras 4.5 y 4.6.

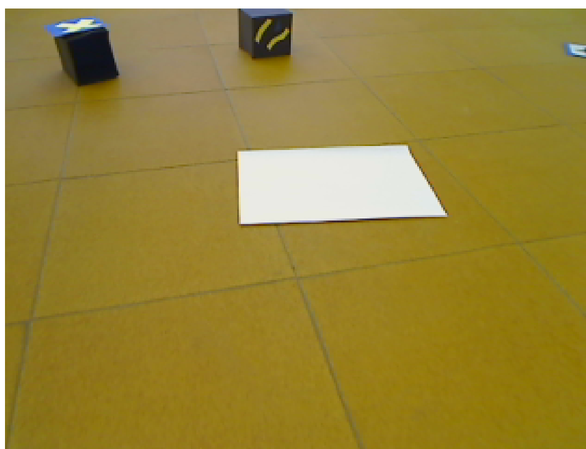

Figura 4.5: Situación problemática para clasificar por apariencia. Hay una hoja en el suelo y el blanco no forma parte de la gama de colores del suelo. Esto lleva a una clasificación errónea de usar sólo un clasificador ya que la hoja no es un obstáculo para el robot.

En la figura 4.7 se puede ver una captura de pantalla de los resultados de floorColorCueComp que se tomó durante uno de los experimentos.

Figura 4.6: Situación problemática para clasificar por apariencia. Hay un cartón en la pared, de la gama de colores del suelo. Desafortunadamente, esto lleva a una clasificación errónea de no haber visto el obstáculo desde otra perspectiva antes, que es el caso más común.

4.3. floorComp

Un sistema de detección de obstáculos basado únicamente en uno de los métodos descritos anteriormente funciona bien bajo determinadas circunstancias. Éstas serían un suelo plano y un entorno altamente texturado en el caso del método geométrico. Para el método de clasificación por color las condiciones serían que los conjuntos de color de los obstáculos y del suelo fuesen casi totalmente disjuntos. Evidentemente, estas condiciones se cumplen muy raramente y, por tanto, el uso de los dos clasificadores está justificado gracias a la mejora en la clasificación que se consigue por la ventaja de tener en cuenta las diferentes propiedades de los diferentes indicios. Una clasificación basada únicamente en el color conllevaría grandes errores muy frecuentemente como la clasificación de trozos de papel en el suelo como obstáculos (esto no debería ser así porque un trozo de papel en el suelo representa difícilmente un obstáculo). Por otra parte, usar únicamente un clasificador geométrico haría chocar al robot con las paredes que no tengan texturas.

Se propone el uso de ambos componentes para obtener una única clasificación de mejor calidad que cualquiera de las otras dos. En [3] se sugiere la fusión de los

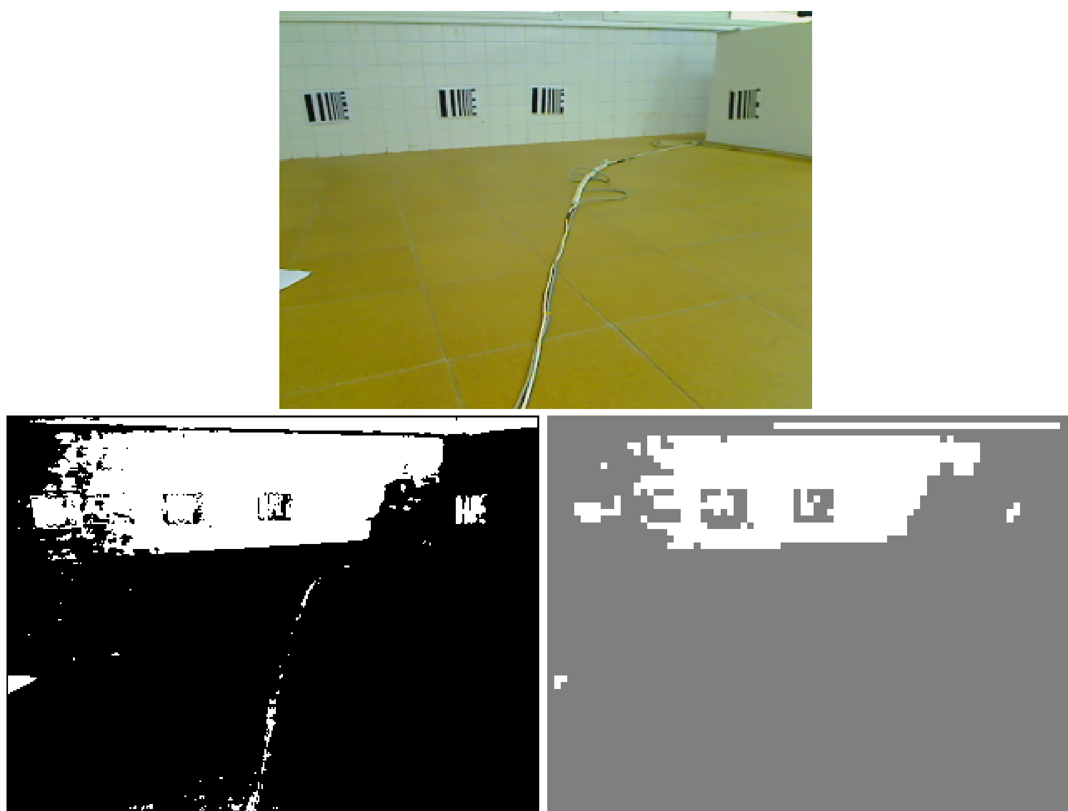

Figura 4.7: Captura de pantalla de los resultados de floorColorCueComp.

resultados de dos clasificaciones distintas mediante los operadores *OR* o *AND*. Si se supone el uso del operador *OR* aquellos objetos o manchas que no sean del color del suelo, pero sobre las que el robot pueda caminar, serán clasificadas erróneamente como obstáculos. Si se supone el uso de *AND* para la fusión, el robot se podría chocar con objetos texturados de color similar al suelo o con objetos no texturados de color distinto al del suelo. A continuación se describe la metodología propuesta en el proyecto para realizar la fusión.

La principal base de la propuesta es la suposición de que una pequeña región clasificada como obstáculo por parte del clasificador basado en color y no por la parte geométrica es, casi con toda seguridad, un objeto lo suficientemente plano para no representar un obstáculo para el robot, o una mancha. No queremos que este tipo de regiones de imagen se clasifiquen como obstáculo. De acuerdo con esto, las zonas de imagen clasificadas como obstáculo por el clasificador de color solamente, y que además están aisladas, son ignoradas y se clasifican como

suelo.

En los entornos estructurdos las paredes carentes de textura son muy comunes y, al carecer de textura, estas zonas que realmente son obstáculos no se clasifican bien por el detector de obstáculos por homografía. A pesar de que las pequeñas regiones aisladas detectadas por floorColorCueComp se ignoren, aquellas que sean grandes y lleguen a la parte superior de la imagen son suficientemente sospechosas de ser una pared como para asumir que son una pared sin textura y, por tanto, un obstáculo. Así pues, estas regiones se clasifican como obstáculos a pesar de que no se detecten por floorPPSComp.

Ambas pistas, la basada en color y la basada en geometría, son fusionadas para obtener una única imagen binaria final en la que dada píxel se corresponde con una ventana de imagen. Las ventanas clasificadas como obstáculos provocan píxeles blancos en la imagen final y viceversa. Como se verá en el capítulo 5, la fusión de pistas de diferente naturaleza permite navegar a través de entornos tanto estructurados como no estructurados. La única restricción es que el suelo sea *localmente* plano.

Lógicamente, floorComp se diseñó con ánimo de que fuese útil para la navegación local. Una forma muy fácil de usarlo es delimitar una zona *peligrosa* de la imagen en función de hacia donde se esté moviendo el robot, de tal forma que cuando dentro de esa zona de imagen haya ventanas categorizadas como obstáculo el robot pare, gire y siga hacia otra dirección. Así fue como se hizo durante las fases más tempranas del desarrollo de floorComp, antes de que se usase un navegador aparte.

A pesar de que una imagen binaria como salida es suficiente, es deseable tener también otra salida que sea semejante a la que produciría un escaner láser. La razón de esto es que directamente de la imagen no se obtiene ninguna información de la distancia a los obstáculos, ya que esta depende de las características de la cámara y su posición dentro del robot. Esto hace que, a pesar de que se pueda usar la imagen binaria para navegar sin ningún tipo de problema, sea más fácil aún de usar. Otra de las ventajas que se obtiene de la transformación de coordenadas (se pasa de coordenadas de imagen a coordenadas dentro del suelo) es que se puede usar cualquiera de los algoritmos previamente dise-

ñados para navegar. De hecho, localNavigatorComp, el navegador usado en el proyecto, trabaja con esta información.

La transformación se realiza escaneando la imagen binaria en búsqueda de ventanas clasificadas como obstáculos. Cada una de ellas se cambia de sistema de referencia, pasando de ser coordenadas de imagen a coordenadas relativas al robot. Una vez así, se calcula el ángulo y la distancia respecto al robot y, si para ese ángulo, la distancia de la ventana es menor de la que se conocía (las distancias a los obstáculos se guardan en un vector que se inicializa al máximo rango permitido por el láser) se modifica el valor para ese ángulo, dejando el valor de la distancia calculada (ver figura 4.8).

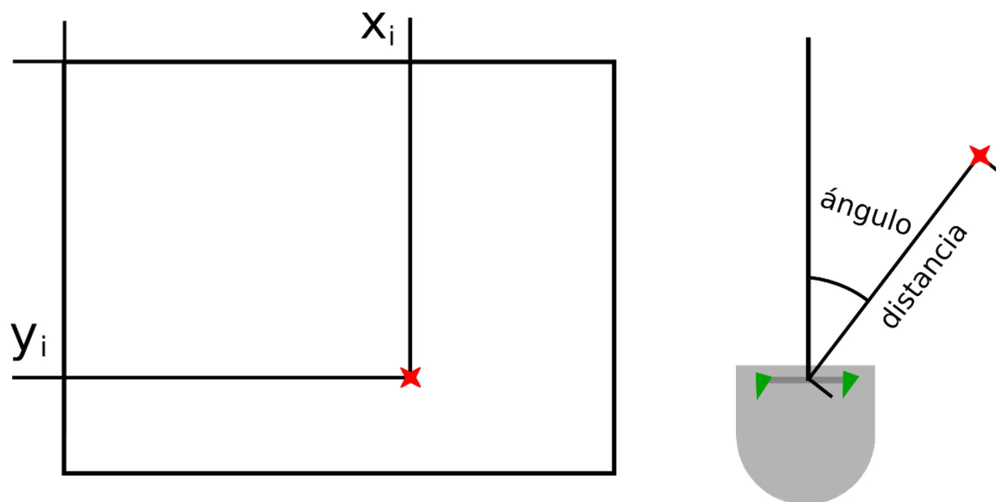

Figura 4.8: Cambio de coordenadas de imagen (parte izquierda) a coordenadas polares tomando como sistema de referencia el robot (parte derecha).

Para hacer que floorComp sea más fácil de usar, ademas de ofrecer la interfaz *Floor*, ofrece la interfaz *Laser* (no hay ningún impedimento para que un componente instancie dos clases de interfaz distintas), por lo que los componentes pueden usar esta última interfaz sin saber si las medidas vienen realmente de un dispositivo láser o son estimaciones hechas por el componente floorComp.

A pesar de las ventajas de la fusión de las clasificaciones, si ningún clasificador es capaz de detectar como tal algún obstáculo real, el robot puede llegar a chocarse. Un ejemplo de esta situación se puede ver en la figura 4.6. Para evitar estos problamas habría que idear algún otro clasificador o usar alguna técncia

de interpolación de los resultados. Afortunadamente estos casos, al igual que el ejemplo de la habitación, no son muy comunes.

El componente trabaja con la información que le esté disponible, bien sea la clasificación por apariencia, por geometría, o ambas. En la figura 4.9 se puede ver una captura de pantalla del componente trabajando con ambas clasificaciones.

Figura 4.9: Captura de pantalla de floorComp. En la parte superior se ve gráficamente la clasificación (canal rojo para el floorColorCueComp, verde para floorPPSComp). En la parte inferior se dibuja el modelo del suelo que se puede obtener del componente.

4.4. localNavigatorComp

El propósito del componente es la navegación local en base a una buena clasificación de obstáculos. Para ello, localNavigatorComp aprovecha la salida de suelo libre transformada en coordenadas polares mediante la interfaz *Laser* que ofrece el componente floorComp. Esto hace que localnavigatorComp pueda trabajar también con lásers reales.

El algoritmo utilizado para navegar no ha sido desarrollado como parte del proyecto, sino que se ha usado uno ya existente que da buenos resultados, VFH* [17]. Lo que sí se ha hecho es extrapolar temporalmente la medida del láser para ampliar el ángulo cubierto de manera aproximada, ya sea por un láser simulado o por uno real.

Una peculiaridad que sucede con la medida láser simulada por floorComp es que abarca demasiado poco ángulo (sólo el que cubra el espacio binocular de la torreta en cada momento), lo que hace que no se sepa si hay obstáculos a los lados a no ser que se haga un sacádico o se gire el robot. Esto provoca que la cantidad de sacádicos o giros hagan demasiado lenta la navegación si no se aplican medidas para evitarlo.

Para solventar este problema es para lo que se incluyó en localNavigatorComp la extrapolación de la medida láser. Consiste en crear una memoria de trabajo más amplia que la medida que da el láser directamente, en este caso de 360º. Esta aproximación modela el entorno como una polilínea cerrada y la actualiza con la última información que llegue de la interfaz láser a la que se conecta y con la odometría. Esta polilínea cerrada, dentro de la cual se sitúa el robot es lo que denominamos "ampliación o extensión de láser".

Dado que esta memoria es válida sólo a corto plazo y que sabemos que no es totalmente fiel a la realidad, se debe ir actualizando con el tiempo haciendo giros o sacádicos, pero aun así, reduce el ritmo necesario al que se tienen que producir. Por tanto, hace más rápido al robot y le permite hacerse una idea de lo que tiene detrás, por lo que se podría hacer que fuese marcha atrás sin necesidad de mirar.

Partiendo de la memoria a corto plazo, sólo hay que aplicar el algoritmo VFH*[17] (cuya implementación en localNavigatorComp no ha formado parte

del proyecto) y mandar las órdenes correspondientes al componente *Base* para navegar. El siguiente capítulo cubre los experimentos realizados con el sistema completo y cada componente.

El algoritmo de actualización de la memoria a corto plazo, que no se encuentra dentro del componente localNavigatorComp, sino dentro de la clase de RoboComp *extendedRangeSensor*, es el siguiente:

1. **Actualizar** cada lectura en t_{k-1} a t_k con la información de la odometría.
2. **Sobrescribir** sobre lo actualizado las medidas *actuales* aportadas por la interfaz láser.
3. **Interpolar** los ángulos no cubiertos.

El último paso es obligatorio porque al hacer la traslación de las medidas durante el tiempo no es necesario que la transformación sea una relación uno a uno. Esto hace prácticamente imposible que todos los ángulos queden cubiertos.

Capítulo 5

Experimentos

En este capítulo se presentan diferentes experimentos que evalúan distintos aspectos de los componentes del sistema, así como su eficacia y robustez global. Para cada uno de ellos se describe la prueba, lo que pretende demostrar, y el resultado obtenido.

5.1. Calibración evolutiva de la homografía

Este experimento consistió en calibrar automáticamente la configuración del par estéreo en base a algunos pares de fotogafías, mediante una aproximación del algoritmo descrito en [15]. La idea es optimizar evolutivamente un vector en el que se guardan los parámetros que determinan la homografía. Con esto se pretende demostrar que floorPPSComp puede funcionar con una torreta estática o de pan común, sin necesidad de conocer ni los parámetros intrínsecos ni los extrínsecos.

El proceso consiste en tomar algunos pares de fotografías con la cámara estéreo en las que sólo aparezca el suelo, aplicar la misma reproyección que en floorPPSComp, y minimizar la luminancia presente en el valor absoluto de la resta entre la imagen izquierda reproyectada y la imagen derecha. Dentro del vector a optimizar se incluyen todos los parámetros intrínsecos y extrínsecos exceptuando la focal, que por suponerse las dos cámaras iguales no influye. Durante los experimentos, se incluyó el calibrador dentro de floorPPSComp para más comodidad.

En la figura 5.1 se puede ver una captura de pantalla de la clase de calibración. A pesar de que los valores obtenidos producen una homografía con un error muy bajo, los valores de los parámetros no son necesariamente los reales, sólo lo serán los valores de la matriz de homografía resultante (no se puede usar para calibrar el par estéreo). En la parte superior de la figura 5.1 se puede ver un par de imágenes de la torreta. En la parte central se puede ver: a la izquierda la imagen izquierda cambiada de perspectiva, y a la derecha el valor absoluto de su resta con la derecha. En la parte inferior se ve la umbralización de la resta y la gráfica del error a través del tiempo. Además, en la gráfica aparece el tiempo empleado.

En menos de un minuto, y con un ordenador de gama baja, se obtienen resultados casi perfectos en casi todas las pruebas hechas. Resultados como los de la figura 5.1 demuestran que el método es eficaz, eficiente y práctico. El único requisito para que el algoritmo pueda aprender rápidamente es que haya una textura poco fina en el suelo.

Para hacer más rápido el aprendizaje se decidió decimar la imagen en función al error. Dado que cuando el ajuste es muy pobre no es necesario mucho detalle, el decimado no estorba mucho, sin embargo se consiguen realizar muchas más generaciones por unidad de tiempo. Esta funcionalidad está incluida dentro de floorPPSComp, por lo que es muy fácil de probar. Para que el entrenamiento funcione debe mostrarse un suelo texturado, de otra manera es evidente que no se puede encontrar ningún error (y por tanto no se podría aprender nada).

5.2. Navegación

Ya que no hay ningún método matemático para comprobar la bondad del sistema planteado, la única forma de evaluarlo es probarlo en condiciones reales. Dado que la navegación autónoma guiada únicamente por visión, a pesar de los avances que ya existen [4, 5], es un campo totalmente abierto, se estableció al comienzo del proyecto de fin de carrera, que navegar durante quince minutos sin chocar era ya un éxito suficiente.

En el resto de esta sección, para tratar de demostrar la utilidad de cada uno

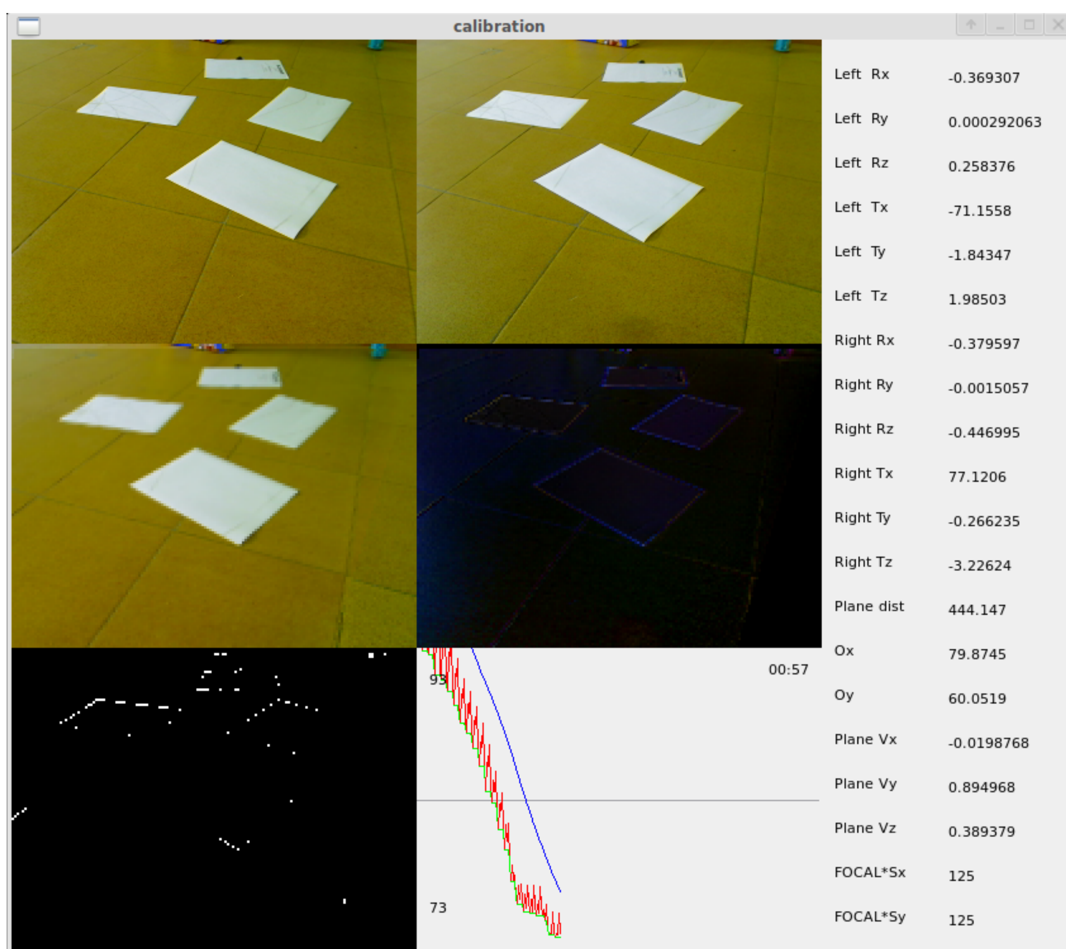

Figura 5.1: Captura de pantalla del calibrador de homografias. En la parte superior de la figura, dos imágenes de la torreta. En la parte inferior, la izquierda cambiada de perspectiva y el valor absoluto de su resta con la derecha. En la parte de abajo la umbralización de la resta y la gráfica del error.

de los susbistemas, se exponen los resultados obtenidos usando el sistema por partes y globalmente, comentando los errores de clasificación que se producen, tanto positivos como negativos.

5.2.1. Guiado por geometría

En este experimento se usó una configuración incompleta, en la que no se contempla la información cromática y el resto de subsistemas siguen funcionan-

do. El resultado del experimento es lo que se predijo y explicó en el capítulo 4.

En las distintas pruebas, el comportamiento es más que satisfactorio mientras se reunan las condiciones necesarias para que el componente floorPPSComp trabaje bien: la presencia de textura en los obstáculos (el suelo no importa que no tenga textura). Sin embargo, en cuanto se le presenta un obstáculo carente de textura, bordes o cualquier tipo de detalle que la cámara no pueda capturar, floorPPSComp es incapaz de obtener ningún tipo de información geométrica del obstáculo, por lo que no puede categorizarlo como tal.

El resultado del experimento fue satisfactorio porque se sabía de antemano que en algunas situaciónes delicadas el componente fallaría. En el resto de situaciones funciona bien y permite navegar al robot sin problema. Sin embargo, es evidente que no es suficiente.

5.2.2. Guiado por apariencia

En este experimento se usó una configuración incompleta, en la que no se contempla la información estéreo, el resto de subsistemas siguen funcionando. Lo obtenido, al igual que en el caso del uso único de información geométrica, es lo que se predijo y explicó en el capítulo 4.

En todas las pruebas realizadas, excepto en el caso del cartón del mismo color en la pared de la figura 4.6, floorColorCueComp no se ha chocado. El problema que tiene floorColorCueComp no son los falsos negativos, sino los falsos positivos. Siempre que el clasificador se encuentra una hoja de papel, cualquier objeto fino, o mancha en el suelo, se categoriza como obstáculo (siempre y cuando no sean del mismo color que el suelo). A pesar de que pueda parecer un mal menor, esto depende del entorno, en algunas ocasiones las hojas tiradas en el suelo o las manchas pueden ser muy comunes. Además, estos resultados excesivamente “miedosos” se han obtenido porque los obstáculos (pared incluida) tienen un color bien diferenciable del suelo (exceptuando el citado cartón), total o parcialmente.

5.2.3. Fusión

En base a los resultados obtenidos en los dos anteriores experimentos, resulta claro que ninguno de los enfoques conocidos basados en apariencia ni en geometría son suficientes por sí mismos. Los dos clasificadores del proyecto se comportan mejor que todos los similares conocidos y aun así, por separado, no son suficientes. En este experimento se demuestra que usando los dos clasificadores descritos en el proyecto y fusionando las salidas se obtienen mejores resultados. Se usan en el experimento los siguientes subsistemas:

- Detección por color
- Detección por estéreo
- Fusión de los dos detectores
- Navegación basada en la fusión anterior

En este experimento se batio de sobra el objetivo de quince minutos sin chocar establecido al comienzo del proycto. Con un navegador que mueve hacia delante el robot hasta encontrar un obstáculo y repite el mismo proceso cambiando de dirección se superó una hora sin chocar.

En las imágenes presentes en la figuras 5.2.3 se pueden ver algunas capturas de pantalla de floorComp durante el experimento.

5.3. Comportamiento en suelos rugosos

La homografía es matemáticamente perfecta únicamente para suelos totalmente planos (ver apéndice B). Sin embargo, el segundo test aporta cierta flexibilidad en cuanto a esto y a la calidad de la homografía que se usa. Para comprobar si esa flexibilidad es suficientemente útil lo más apropiado es hacer pruebas. Las pruebas se hicieron en los exteriores de la Escuela Politénica de Cáceres. El suelo presenta grandes rugosidades que los humanos no notan porque los pies son mayores que éstas, sin embargo RobEx sí lo hace debido a sus

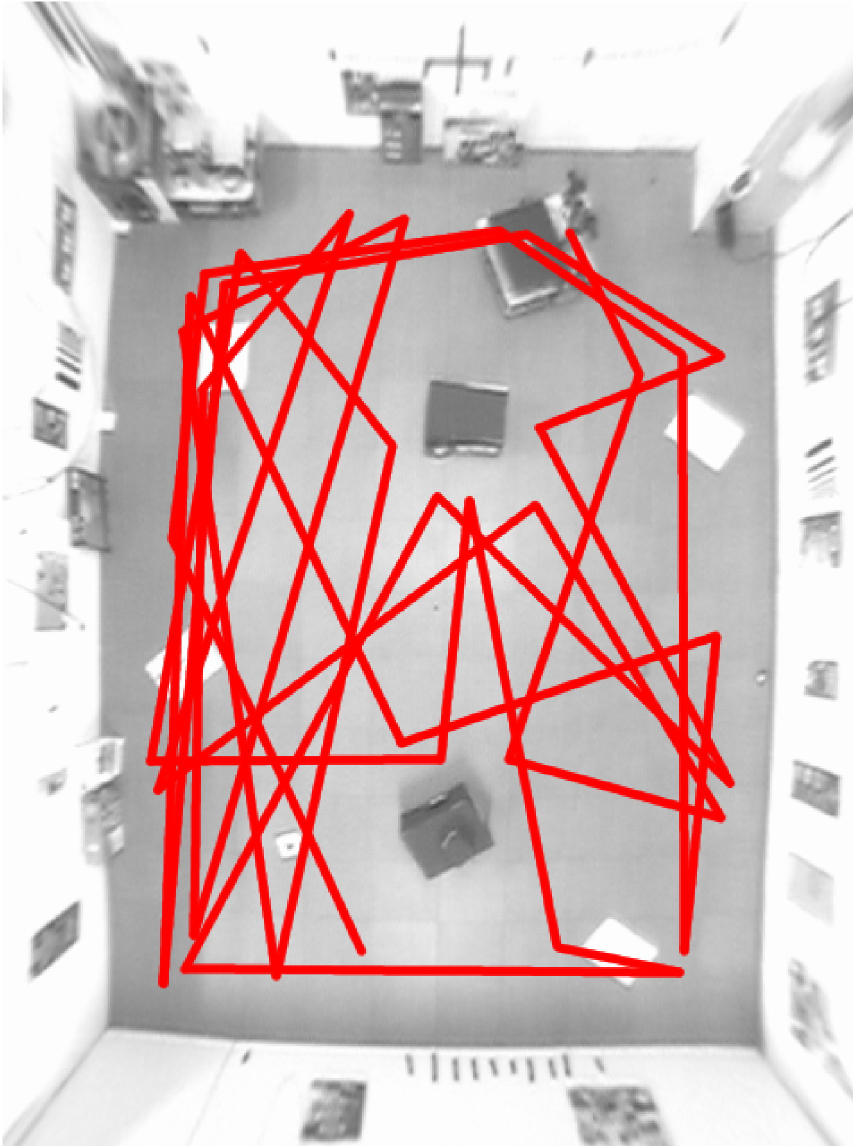

Figura 5.2: Parte de la trayectoria descrita durante la prueba de los 15 minutos usando la fusión descrita de los dos detectores.

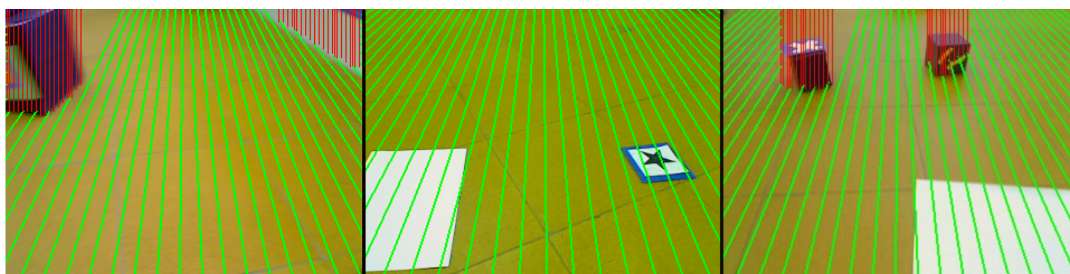

Figura 5.3: Capturas de floorComp durante los experimentos.

finas ruedas. Estos pequeños escalones mueven el plano modelado por la homografía respecto al real, además de que hay pequeñas diferencias en la altura del suelo.

En las distintas pruebas realizadas se vieron algunos errores de clasificación parpadeantes, igual que sucede cuando hay brillos. Sin embargo la frecuencia de los errores es suficientemente baja como para que sea absorbido por cuaquier filtro, el del VFH por ejemplo. En la figura 5.4 se puede ver una captura de pantalla de floorPPSComp realizada el día en el que se hizo la primera prueba.

5.4. Acceso compartido a la torreta

Por construcción, el sistema no bloquea el movimiento de la torreta. Por tanto si se calcula la homografía en tiempo real, a pesar de que la torreta se mueva, el sistema sigue funcionando.

El único problema surge mientras se produce el movimiento de la torreta. Para que el algoritmo trabaje correctamente la torreta debe estar quieta un pequeño periodo de tiempo, el suficiente para que la información de la torreta se actualice y las cámaras capturen una imagen nítida. Exceptuando esto no hay ningún otro problema. Este problema, que no tiene que ver directamente con la solución propuesta y se puede mitigar subiendo el voltaje de la torreta a 19V, lo tienen también los humanos en cierta medida, por lo que no parece que sea un gran problema.

Las pruebas se hicieron mediante un componente ajeno que modificaba pe-

riodicamente la configuración de la torreta, de tal forma que el robot giraba la vista de izquierda a derecha. Esto permitió al robot detectar obstáculos a ambos lados sin moverse.

5.5. Calibración algebraica de la homografía

En este experimento se comprobó el correcto funcionamiento de la calibración algebraica de la torreta para comprobar si, con el hardware actual de RobEx, es posible recalcular la homografía en tiempo real y sin que floorPPSComp cometa errores. Para ello se configuró el componente para que, en vez de usar alguna homografía por defecto o usar la calibración evolutiva (que no es suficientemente rápida como para aprender en tiempo real), se calculase en función a las fórmulas presentes en B.

Gracias al segundo test introducido en floorPPSComp, los resultados fueron totalmente satisfactorios. A pesar de estar trabajando con una torreta calibrada manualmente, en la que la homografía tiene grandes errores (se puede ver la figura B.1 del apéndice B), no se notó gran diferencia con respecto al funcionamiento con la torreta estática.

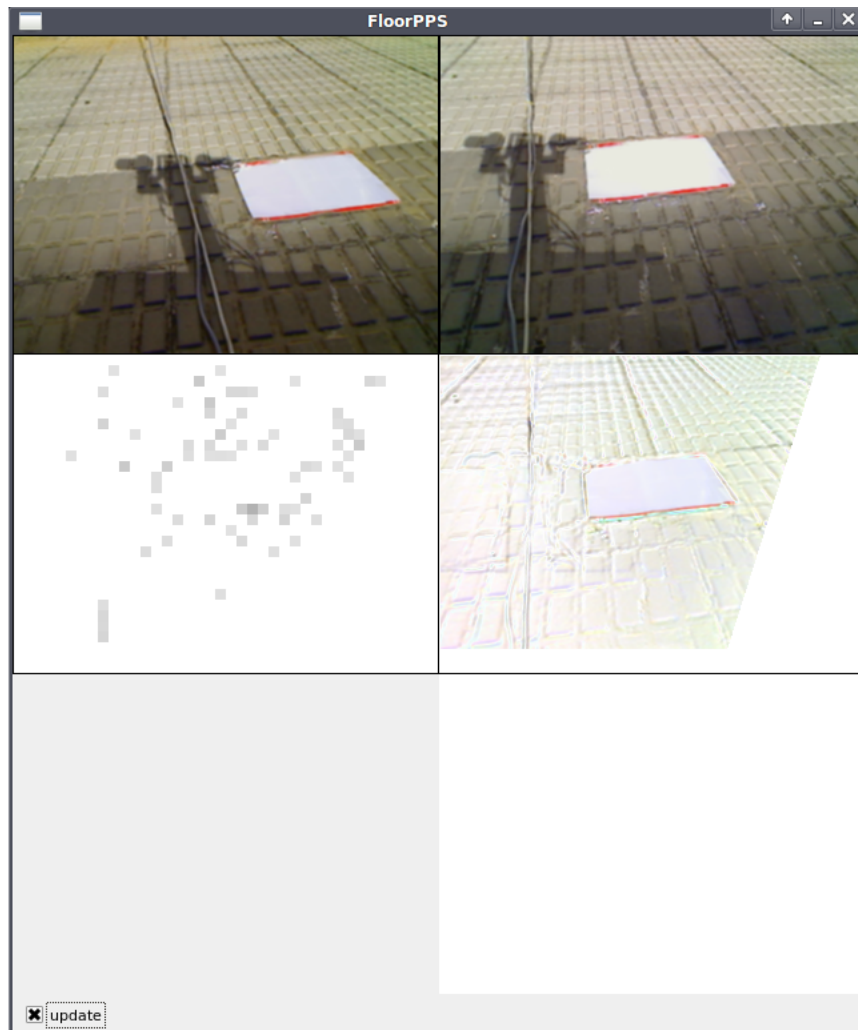

Figura 5.4: Captura de pantalla de floorPPSComp sobre suelo rugoso. Es importante resaltar que, exceptuando las dos imágenes de las cámaras, se han invertido los colores del resto de las imágenes para ahorrar tinta. Como se aprecia en las imágenes centrales hay ventanas que pasan el primer test. Sin embargo, como se puede ver en la más inferior, se detecta todo como suelo (no hay ninguna ventana marcada).

Capítulo 6

Instalación y puesta en marcha

Este capítulo trata la instalación y puesta en marcha del sistema de detección de obstáculos, desde la descarga del software hasta las opciones de configuración. El proceso de puesta en marcha se ha dividido en las siguientes secciones:

1. **Instalación de requisitos:** Como todo software, el sistema de detección de obstáculos tiene dependencias software que hay que solventar. Esto se trata en la sección 6.1.

2. **Descarga:** En la sección 6.2 se muestran las diferentes formas de obtener el código fuente.

3. **Compilación:** El proceso de compilación, junto a los posibles prblemas que se puedan encontrar, se tratan en la sección 6.3.

4. **Configuración:** Los distintos pasos para configurar el sistema una vez está compilado se describen en la sección 6.4.

5. **Ejecución:** Finalmente, en la sección 6.5 se describe el proceso de ejecución.

6.1. Instalación de requisitos

Ya se habló de esto en el capítulo 2, sin embargo es necesario no sólo enumerar los requisitos, sino dar una manera fácil de cumplirlos. Para esto se supondrá

que se ha instalado un sistema operativo derivado de Debian. Esto permite instalar sencillamente los paquetes necesarios.

IPP

IPP es una biblioteca privativa y, hasta el momento, no se puede obtener de ninguno de los repositorios de paquetes oficiales de ninguna distribución conocida. Esto nos obliga a hacer una instalación totalmente manual. Para descargar la biblioteca hay que ir al siguiente enlace https://registrationcenter.intel.com/RegCenter/AutoGen.aspx?ProductID=1287&rm=NCOM. Para poder hacer la descarga se nos pedirá el correo electrónico, a donde nos mandarán la licencia de uso de IPP. Es importante elegir correctamente la versión de la biblioteca, de no hacerlo es probable que no se le saque todo el partido a IPP.

Una vez se ha descargado el software y se ha descomprimido, la instalación es guiada y muy sencilla, raramente se encuentran problemas en este paso. En el transcurso de ésta, se nos pedirá la licencia, para ello nos dará la opción de introducir manualmente el código de licencia o dar la ruta donde se encuentra el fichero que se nos envió al correo desde Intel. Para integrar IPP con RoboComp es conveniente añadir a siguiente línea al fichero *.bashrc* dentro del directorio home (o al fichero que corresponda si no se usa bash).

```
export IPPROOT=/opt/intel/ipp/5.3.3.075/ia32/
```

Además, después de añadir la siguiente línea al fichero */etc/ld.so.conf*, se ejecutará como superusuario el comando *ldconfig*. En la línea, los directorios *5.3.3.075* y *ia32* se cambiarán en función a la versión de IPP que se haya instalado y la arquitectura de procesador.

```
/opt/intel/ipp/5.3.3.075/ia32/sharedlib/
```

Para futuros trabajos sería interesante migrar a FrameWave debido a que se obtiene la misma eficiencia con un software totalmente libre e integrado en la mayoría de distribuciones GNU/Linux modernas. En caso de querer probar FrameWave, los paquetes a instalar serían:

```
libfwbase1
libfwbase1-dev
```

Qt4

El framework Qt suele venir instalado por defecto en casi todas las distribuciones GNU/Linux modernas, sin embargo, unicamente se encuentran disponibles los binarios. Para instalar las cabeceras, y las herramientas de desarrollo asociadas a Qt, se instalarán los siguientes paquetes:

```
qt4-dev-tools
libqt4-dev
```

Ice

Como ya se ha visto, Ice es el middleware en el que se basa RoboComp. En este caso, no suele estar instalado ni las cabeceras ni los binarios. Los paquetes a instalar son:

```
libzeroc-ice33-dev
zeroc-ice33
```

CMake

CMake se necesitará para compilar los componentes de RoboComp. Gracias a que es totalmente libre, se encuentra en casi todas las distribuciones GNU/-Linux modernas. Actualmente no viene por defecto instalado, pero dado que su uso está creciendo rápidamente es probable que pronto lo esté. En caso de no estar instalado (se puede comprobar ejecutando el comando *cmake*) se instalará con el siguiente paquete:

```
cmake
```

Kate

Kate es el editor de texto que se ha usado para escribir el código del proyecto. En este caso se recomienda la instalación por su comodidad y la capacidad de resaltar la sintaxis de los ficheros.

```
kate
```

De todas formas, esto es sólo una sugerencia, es evidente que cada uno puede usar el editor de texto con el que más cómodo se sienta.

managerComp

La herramienta managerComp es la que nos permite visualizar, tanto gráficamente como en una lista, el estado de los componentes configurados en tiempo real, así como modifcar dicho estado. Deberemos instalar los paquetes siguientes:

```
pyqt4-dev-tools
python-qt4-dev
```

Esto no instala managerComp, simplemente nos permitirá instalarlo sin problemas cuando llegue el momento.

OpenSSH

Como se explicó en el capítulo 2, OpenSSH nos permite la ejecución remota de comandos de una forma segura. El cliente suele venir instalado por defecto en la mayoría de distribuciones modernas de GNU/Linux, sin embargo, no suele pasar lo mismo con el servidor. El servidor se instala con el siguiente paquete:

```
openssh-server
```

Su uso junto a managerComp nos permite controlar los componentes muy fácilmente, es por ello que es interesante dedicar unos minutos a su configuración. A pesar de no ser excesivamente tedioso, se ha decidido dedicarle un apéndice para que se pueda consultar más rápidamente. La información se puede encontrar en el apéndice C.

Otros

Además, para que camaraComp compile será necesario instalar los siguientes paquetes también:

```
libavc1394-dev
libdc1394-22-dev
libraw1394-dev
```

6.2. Desgarga

El software se encuentra en el disco que acompaña a la memoria del proyecto, por lo que para obtener el código fuente basta con copiarlo de ahí. Sin embargo es probable que se prefiera conseguir la última versión del mismo. En tal caso se necesitará un cliente de subversion para acceder al repositorio de RoboComp en SourceForge. El paquete a instalar es el siguiente:

```
subversion
```

Con subversion instalado, se descarga RoboComp con el siguiente comando:

```
svn co https://robocomp.svn.sourceforge.net/svnroot/robocomp robocomp
```

Una vez termina la ejecución del comando, debería haberse creado un nuevo directorio llamado robocomp en el directorio desde el cual se invocó. Dentro de ese directrio se encuentra todo el código de RoboComp.

6.3. Compilación

En esta sección se especifica el proceso de compilación del software necesario para que el sistema esté funcionando.

Componentes

Los componentes se compilan todos igual y, si no hay ningún problema, es muy sistemático. El proceso, que se repite para cada componente con el que se quiera contar, consiste en situarse en el directorio del componente a instalar y ejecutar dos comandos. Se puede hacer en una unica línea:

```
cmake . && make
```

Esto dejará el binario del componente en el correspondiente diectorio del componente. Para ejecutar cuaquier componente Ice suele ser útil la opción –*Ice.Config=X*, con la que se especifica donde ha de buscar el programa su fichero de configuración.

managerComp

La aplicación managerComp es muy sencilla de compilar. Con sólo dos comandos se compila e instala. Como superusuario:

```
cd <path_a_robocomp>/Components/stable/managerComp
make install
```

6.4. Configuración

Al igual que sucede con la compilación, con la configuración se puede hacer también una distinción entre la de los componentes y la de managerComp. La configuración de los componentes consiste en la modificación de los ficheros de configuración que llevan por defecto. La configuración de managerComp es algo más tediosa.

Para cada componente a usar, se copia el fichero que está en su subdirectorio *etc/*, *etc/config*, a un directorio donde se puedan poner los ficheros de configuración de los componentes que se van a usar, sin modificar los que existen en el repositorio. Se han de copiar los ficheros de configuración de todos los componentes que se van a usar, es decir, baseComp, cammotionComp, camaraComp, floorppsComp, floorcolorcueComp, floorComp y localnavigatorComp. Evidentemente, a cada uno se le pondrá un nombre distinto y que vaya en consonancia con el componente al que están asociados.

En todos los ficheros de configuración se tendrá que modificar el host al que se dirigen las conexiones. En este caso, se supondrá que todos los componentes menos localnavigatorComp (que no es usado por ningún otro componente) se ejecutan en el robot. Por tanto, en cada fichero de configuración guardado se cambiarán las líneas tipo *XXXProxy* de forma que de estar presente la opción *-h* (que especifica el host en el que reside el componente al que se pretende hacer una conexión) se cambia el host que esté puesto por el nombre de host o la IP del robot. De no existir la opción *-h* el componente intentará hacer una conexión local, por lo que habrá que añadir la opción *-h* seguida del nombre de host o la IP de robot en caso de que no se desee este comportamiento.

Además, dependiendo del componente habrá otros parámetros de configuración que se puedan querer cambiar. A continuación se muestran los ficheros

de configuración de cada componente. Son autoexplicativos, por lo que no es necesario extenderse más:

baseComp

```
BaseComp.Endpoints=tcp -p 10004

Device=/dev/ttyUSB0 # Dispositivo serie de conexión al robot.
Gear=FAUL           # Tipo de encoder del motor.
```

cammotionComp

```
CamMotionComp.Endpoints=tcp -p 10005

Device=/dev/ttyUSB1    # Dispositivo serie de conexión a la torreta.
Handler = Dynamixel    # Tipo de servo montado (depende del hardware).

TILT_MOTOR = 2         # ID del servo del movimiento de tilt.
LEFT_MOTOR = 1         # ID del servo del movimiento de pan izquierdo.
RIGHT_MOTOR = 0        # ID del servo del movimiento de pan derecho.
LEFT_CAMERA = 0        # ID de la cámara izquierda.
RIGHT_CAMERA = 1       # ID de la cámara derecha.
BOTH_CAMERAS = 5       # ID para referirse a ambas cámaras.
RIGHT_MOTOR_ZEROPOS = 512  # Offset del servo del pan derecho.
LEFT_MOTOR_ZEROPOS = 520   # Offset del servo del pan izquierdo.
TILT_MOTOR_ZEROPOS = 410   # Offset del servo del tilt.
BASELINE = 155             # Distancia entre los ejes de los servos.
```

camaraComp

```
CamaraComp.Endpoints=tcp -p 10001

CamMotionProxy=cammotion:tcp -p 10005 -h robot
BaseProxy=base:tcp -p 10004 -h robot

Camara.Focal = 880           # Parámetros de la cámara.
Camara.FPS = 30
Camara.Width = 320
Camara.Height = 240

Camara.TalkToBaseComp = false           # Inf. de odometría
Camara.TalkToCamMotionComp = false      # Inf. de la torreta

Camara.Device = /dev/video0,/dev/video1 # Dispositivo(s) de cámara.
Camara.Driver = V4L2               # Tipo de cámara.
Camara.Width = 320                 # Ancho de imagen solicitado.
Camara.Height = 240                # Alto de imagen solicitado.
Camara.FPS = 15                    # Frecuencia de actualizacińn (Hz).
Camara.NumCamaras = 2              # Número de cámaras a controlar.

Ice.MessageSizeMax=10240           # Máximo tamao de mensaje grande es
                                   # importante para poder *mandar*
                                   # video en clor a buena resolución.
```

floorppsComp

```
FloorPPSComp.Endpoints=tcp -p 10047

CameraProxy = camara:tcp -p 10001 -h robot
FloorProxy = floor:tcp -p 10046 -h robot

Ice.MessageSizeMax=10240          # Igual que en camaraComp, esto
                                  # es importante para *recibir*
                                  # mensajes grandes.
```

floorcolorcueComp

```
FloorColorCueComp.Endpoints=tcp -p 10048

CameraProxy = camara:tcp -p 10001 -h robot
FloorProxy = floor:tcp -p 10046 -h robot

HistogramPath = /home/luisj/mat.rcd # Ruta a fichero que contiene
                                    # el histograma obtenido en el
                                    # entrenamiento

Ice.MessageSizeMax=10240            # Igual que en camaraComp, esto
                                    # es importante para *recibir*
                                    # mensajes grandes.
```

floorComp

```
FloorComp.Endpoints=tcp -p 10046 # Lo interesante de este componente
LaserComp.Endpoints=tcp -p 11003 # es que, como se puede ver, ofrece
                                 # dos interfaces distintas.

BaseProxy = base:tcp -p 10004 -h robot           # Endpoint
CamMotionProxy = cammotion:tcp -p 10005 -h robot # cammotionComp

Floor.Width=160                  # Ancho de la imagen de salida
Floor.Height=120                 # Alto de la imagen de salida
```

localnavigatorComp

```
LocalNavigatorComp.Endpoints=tcp -p 18007

LaserProxy = laser:tcp -p 11003 -h robot # Endpoint de la interfaz
                                         # laser, en este caso,
                                         # floorComp.
BaseProxy = base:tcp -p 10004 -h robot   # Endpoint de baseComp
```

managerComp

En la figura 6.1 se muestra el diálogo de configuración de un nuevo componente dentro de managerComp. Se encuentran los siguientes campos:

1. **Alias**
2. **Path**
3. **Up Command**
4. **Down Command**
5. **Dependences**
6. **Endpoint**
7. **Configuration file**

El campo *alias* hace referencia al alias que se usará dentro de la aplicación para el componente (es útil para expresar las dependencias de una forma más compacta que con el endpoint). El campo *path* indica en qué directorio local se deben ejecutar las órdenes del componente, tanto cuando se quiera encender como apagar. Los campos *up command* y *down command* son los comandos a ejecutar para encender o apagar el componente, respectivamente. *Dependences* hace referencia a las dependencias de un componente, expresadas como una lista de alias separados por comas. *Endpoint* es el la forma de localizar al componente de la que se ha hablado en capitulos anteriores. *Configuration fie* indica el path del fichero a modificar cuando se solicite cambiar la configuración del componente mediante la interfaz gráfica de managerComp.

Una vez se ejecuta managerComp se le pueden añadir nuevos componentes a contorlar mediante la acción *File*, *Edit*. En ese momento se mostrará la interfaz de la figura 6.1 que permite editar la lista de componentes monitorizados.

6.5. Ejecución

Una vez está configurado managerComp se puede consultar el estado de los componentes con cualquiera de sus dos interfaces. En ambos modos los

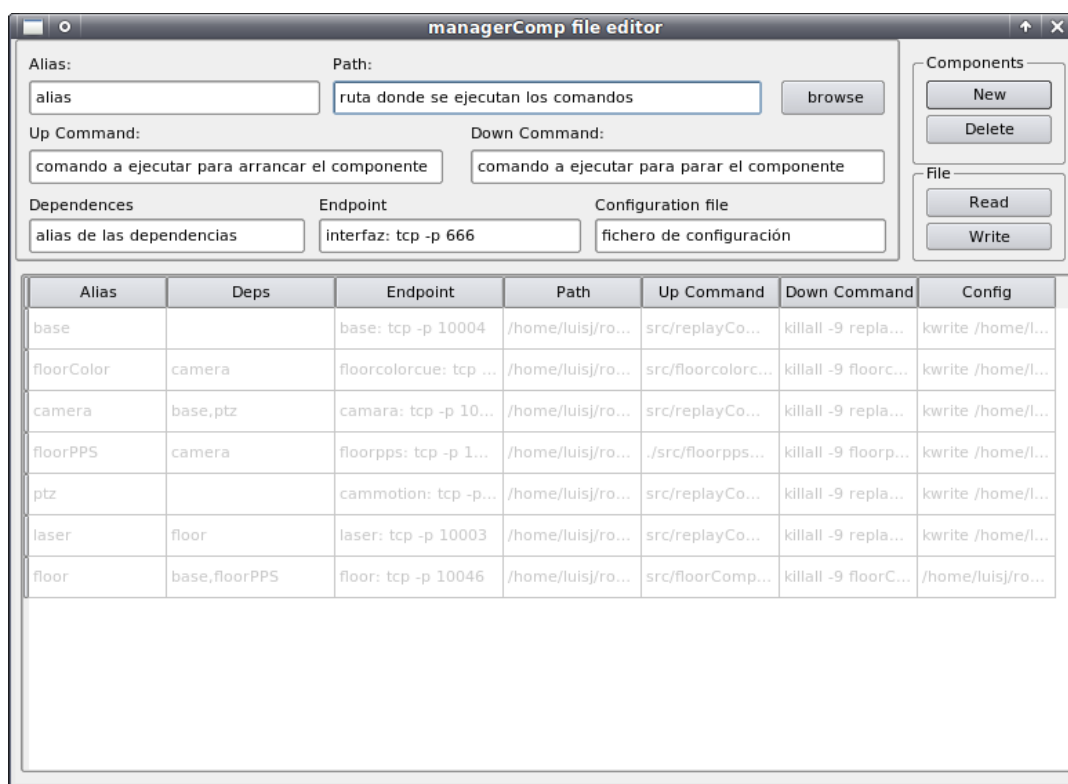

Figura 6.1: Diálogo de configuración de managerComp. Se muestra una guía de cómo rellenar la información de manipulación de un componente.

componentes accesibles se muestran en verde y los apagados en rojo. En el modo grafo (figura 6.2) se ha de hacer clic sobre el nodo en cuestón para poder encenderlo o apagarlo.

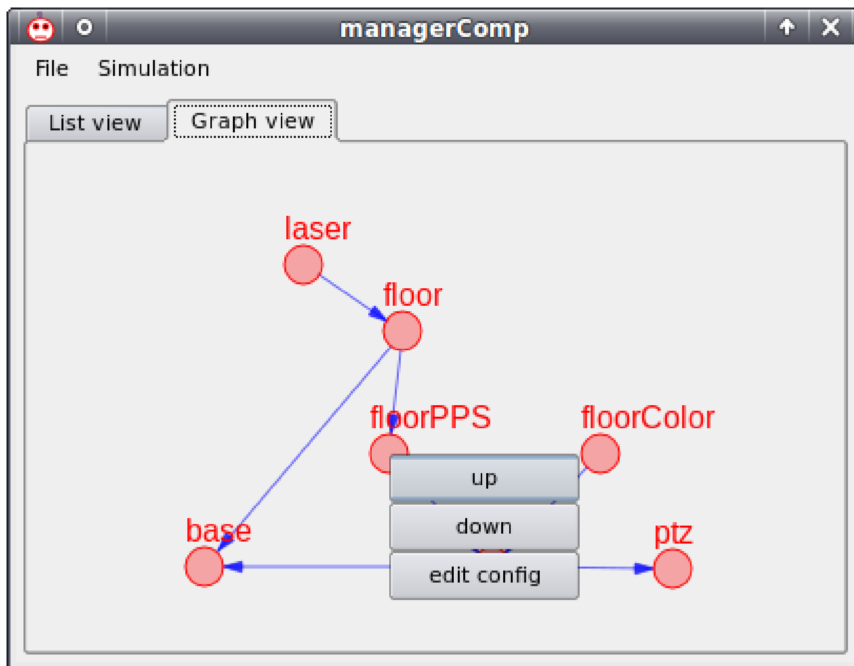

Figura 6.2: Interfaz de managerComp en modo grafo.

En el modo lista (figura 6.3), que ofrece más información y es más apropiado para cuando hay algún probema con alguno de los componentes, los componentes se cambian de estado haciendo doble clic sobre ellos.

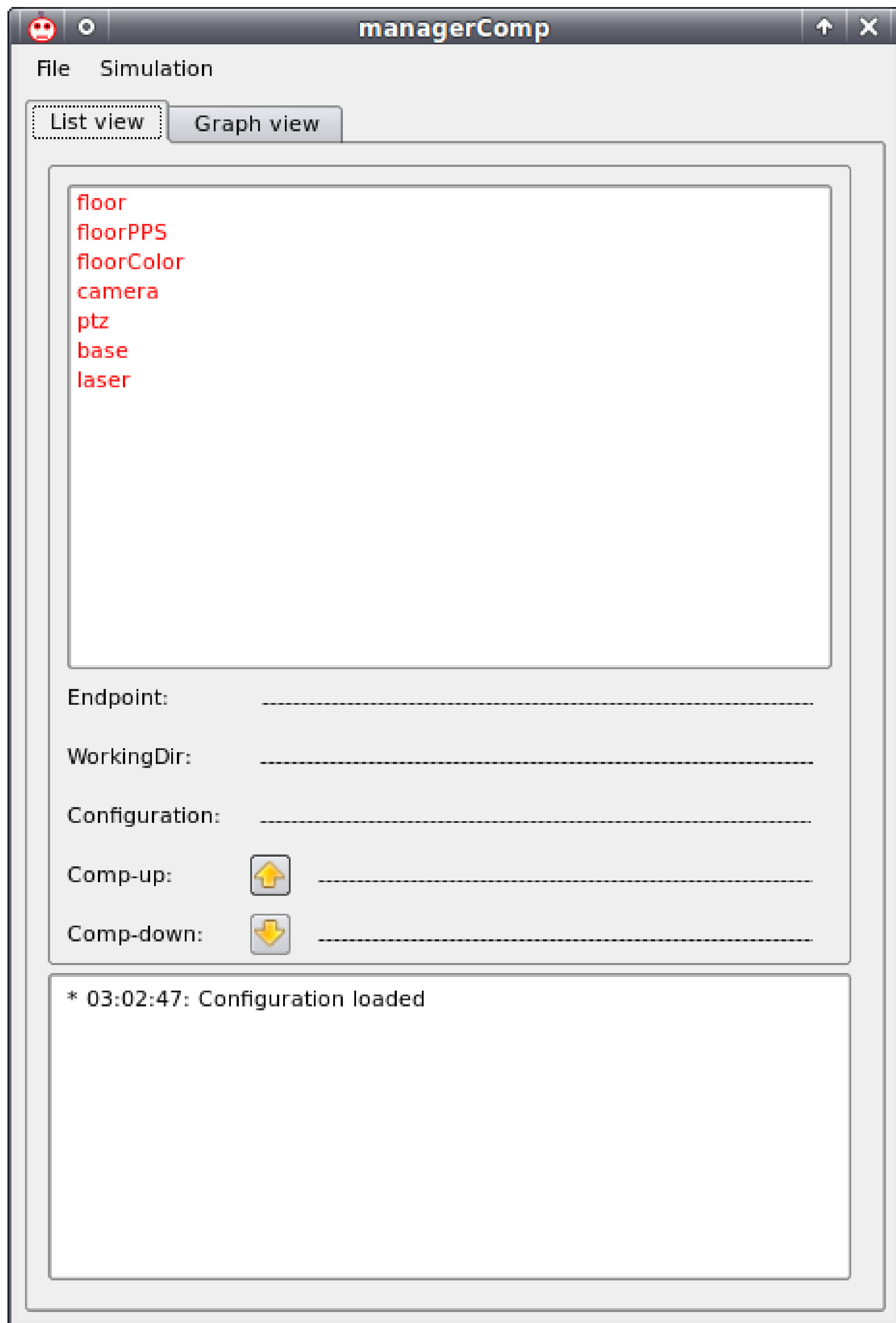

Figura 6.3: Interfaz de managerComp en modo lista.

Capítulo 7

Conclusiones

Después de varios meses de trabajo se consiguió llegar a una solución satisfactoria que permite al robot navegar. Las pruebas realizadas demuestran que las técnicas desarrolladas hacen que el robot se pueda desenvolver en entornos aproximadamente planos. Además, debido a su diseño basado en componentes, sería fácil ampliarlo o acoplarlo a otros componentes existentes para que funcionen ambos en conjunto, extendiendo así las capacidades del robot.

Se han comparado varias veces las propiedades del sistema con las de sistemas parecidos usados en la actualidad. Asimismo, se ha demostrado que funciona mejor en varias circunstancias, no teniendo más puntos débiles respecto a éstos que un ligero incremento en el coste computacional.

Apéndice A

Modelo de base robótica diferencial

Las bases robóticas de tipo diferencial son aquellas que tienen únicamente dos ruedas motrices y, opcionalmente, una o varias ruedas de apoyo. Dichas ruedas de apoyo no influyen en el desplazamiento de la base, su única función es garantizar la estabilidad. El ejemplo más común de vehículo de este tipo es la silla de ruedas.

La simplicidad de diseño y de los cálculos de la odometría con el modelo diferencial son las principales razones que suelen motivar su uso, no sólo de RobEx, sino de otros muchos robots y gamas de ellos, como pueden ser: Segway, Kphera, o Roomba.

Veamos cómo resolver el problema de la odometría. Todo se basa en que, para una velocidad de ruedas concreta (cada una de las dos ruedas a una velocidad constante), éstas describen dos circunferencias concéntricas. La odometría se resuelve asumiendo que entre cada actualización que se realice, las velocidades de las ruedas no cambian (cuanto menor sea el periodo de actualización, más se acercará esto a la realidad). Siendo así, sólo hay que calcular los desplazamientos realizados entre muestreo y muestreo e integrarlos. Suponiendo las siguientes definiciones que se pueden ver en la figura A.1 y que hacen referencia a lo que ocurre en cada paso:

- $\boldsymbol{R}$ radio de las ruedas.
- $\boldsymbol{r_d}$, $\boldsymbol{r_i}$ radio de giro de las ruedas derecha e izquierda, respectivamente.
- $\boldsymbol{S}$ distancia entre ruedas.

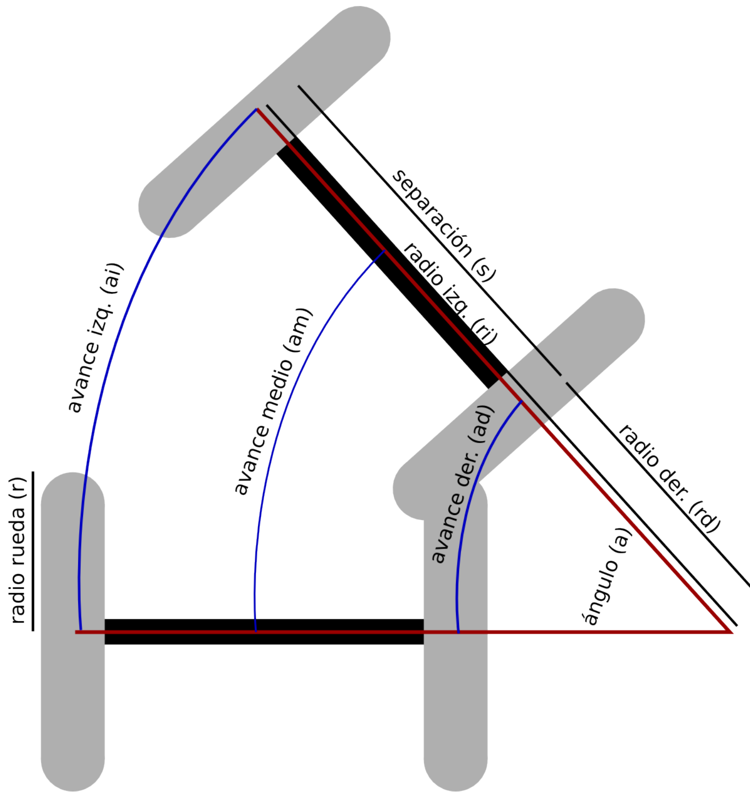

Figura A.1: Modelo de base robótica diferencial.

- $\boldsymbol{a}$ ángulo de giro de la base.
- $\boldsymbol{A_d}$, $\boldsymbol{A_i}$ distancia recorrida por las ruedas derecha e izquierda, respectivamente.
- $\boldsymbol{a_m}$ distancia recorrida por el centro de la base.

$$A_d = r_d * a \tag{A.1}$$

$$A_i = r_i * a \tag{A.2}$$

$$r_i = r_d + S \tag{A.3}$$

entonces, restando A.2 de A.1 se obtiene:

$$A_i - A_d = r_i * a - r_d * a \tag{A.4}$$

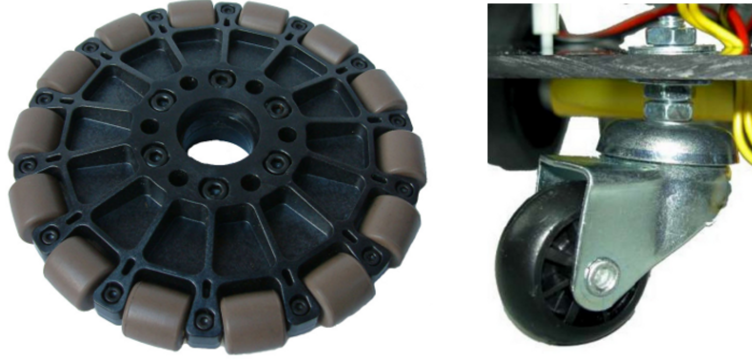

Figura A.2: Ejemplos de ruedas de apoyo para robots de tipo diferencial.

que sustituyendo r_i tal y como en A.3:

$$A_i - A_d = (r_d + S) * a - r_d * a = S * a \tag{A.5}$$

Por lo que el ángulo de giro de la base del robot será:

$$a = (A_i - A_d)/S \tag{A.6}$$

El avance del punto intermedio de las ruedas será la media del avance de las dos ruedas:

$$a_m = (A_i + A_d)/2 \tag{A.7}$$

Dado que a_i y a_d se calculan fácilmente conociendo el radio de las ruedas y el ángulo que se han movido durante el periodo de muestreo, sólo tenemos que saber integrar los desplazamientos y giros de la base. Suponiendo que se guarda la posición del robot en un vector $p_k = [x_k, y_k, ang_k]$ y aproximando el arco descrito por la base por su cuerda, válido cuando el periodo de muestreo tiende a cero, se puede integrar la posición mediante:

$$p_k = p_{k-1} + \begin{bmatrix} a_m * cos((\pi/2) + p_{k-1}(3)) \\ a_m * sin((\pi/2) + p_{k-1}(3) \\ p_{k-1}(3) + a \end{bmatrix} \tag{A.8}$$

Apéndice B

La relación de homografía

Una homografía es una relación existente en geometría proyectiva que vincula la proyecciones de puntos de un plano π con sus correspondientes proyecciones en otro plano. De acuerdo con [7]: Una homografía $h(x)$ es una relación definida y aplicada sobre P^2, de forma que tres puntos x_1, x_2, x_3 cualesquiera peternecen a la misma línea si y sólo si igualmente lo hacen $h(x_1)$, $h(x_2)$, $h(x_3)$. Dicho de otra forma: una aplicación h definida y aplicada sobre P^2 es una homografía si y sólo si existe una matriz H no singular cuadrada de tamaño 3x3, de forma que para cada punto de P^2 representado por un vector x, $h(x) = Hx$.

Debido al uso de coordenadas homogéneas, de los nueve elementos de la matriz H, únicamente hay ocho grados de libertad y, por tanto, ocho parámetros. Si los sistemas de coordenadas son euclideos, la relación por homografía está más restringida y pasa a contar con seis grados de libertad (ver [7]).

En la figura B.1 se pueden ver dos imágenes tomadas simultáneamente con la torreta del robot y el resultado de aplicar la homografía a la imagen de la cámara izquierda. En la figura, el par estéreo fue calibrado manualmente (lo que conlleva errores considerables) y aún así se puede observar que los errores no son muy grandes.

Para poder hallar la matriz de homografía (asociada a un cambio de perspectiva concreto de un plano en concreto) es necesario conocer los parámetros extrínsecos de las cámaras y la posición del plano respecto a la cámara original. Además, también es necesario conocer los parámetros intrínsecos en el caso de que las cámaras sean diferentes. Conociendo estos parámetros, la matriz de

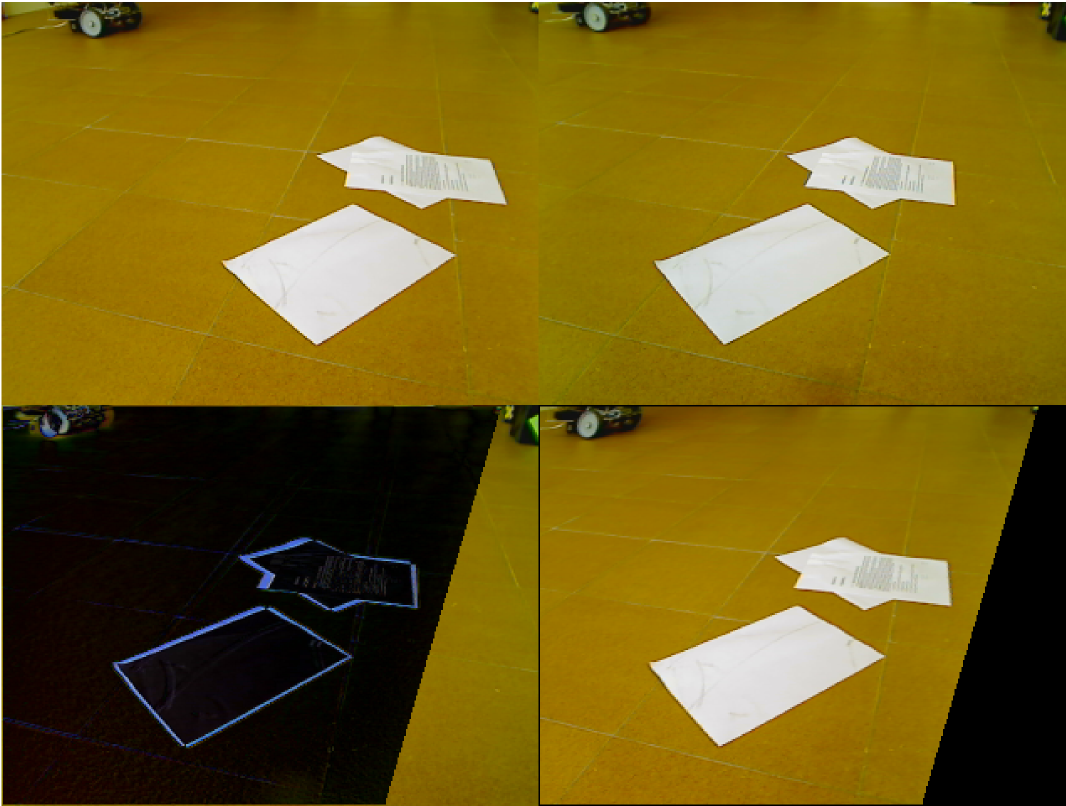

Figura B.1: Arriba las imágenes de las cámaras del robot. Abajo a la derecha, el resultado de la reproyección de la imagen izquierda. Abajo a la izquierda el valor absoluto de la resta de la reproyección con el contenido real de la imagen derecha.

homografía se calcula de la siguiente manera [7]:

$$H = K'(R - tn^T/d)K^{-1},$$

donde:

- K y K' son las marices de parámetros intrínsecos de las cámaras de partida y destino, respectivamente.
- R y t son la matriz de rotación y el vector de traslación que llevan desde el centro de proyección de la cámara de partida hasta el de la de destino.
- n es el vector perpendicular al plano que induce la homografía, normalizado.

- d es la mínima distancia que hay desde el punto de vista de partida hasta el plano.

Si se usa RoboComp, la homografía se puede calcular fácilmente usando la clase RMat::Homo que se encuentra definida e implementada en los ficheros Clases/qmatriz/qhomo.h y Clases/qmatriz/qhomo.cpp respectivamente. El código sería el siguiente:

```
1   Ri.update(hState.tilt.pos, hState.left.pos, 0);
2   Rd.update(hState.tilt.pos, hState.right.pos, 0);
3
4   nPlane = QVec::vec3(0,
5                       cos(-hState.tilt.pos),
6                       sin(-hState.tilt.pos)).toColumnMatrix();
7
8   Ti = QVec::vec3( DESP_CAMARA, 0, 0).toColumnMatrix();
9   Td = QVec::vec3(-DESP_CAMARA, 0, 0).toColumnMatrix();
10
11  homography.update(K,
12                    Rd*(Ri.transpose()),
13                    Rd.transpose()*(Ti-Td),
14                    nPlane,
15                    dPlane);
16
17  coeffs[0][0]=homography(0,0);
18  coeffs[0][1]=homography(0,1);
19  coeffs[0][2]=homography(0,2);
20  coeffs[1][0]=homography(1,0);
21  coeffs[1][1]=homography(1,1);
22  coeffs[1][2]=homography(1,2);
23  coeffs[2][0]=homography(2,0);
24  coeffs[2][1]=homography(2,1);
25  coeffs[2][2]=homography(2,2);
```

El código calcula la matriz de homografía tomando como sistema de referencia el punto medio entre las dos cámaras. Por ello, en las dos primeras líneas se construyen dos matrices de rotación, una para cada cámara (si el sistema de referencia fuese una de las cámaras sólo haría falta una matriz de rotación). El vector normal al plano se construye en la línea 4. Dado que debe ser un vector unitario, se usan las funciones seno y coseno. En las líneas 8 y 9 se construyen los vectores desplazamiento de las cámaras. El código restante hace uso de la clase RMat::Homo, que calcula H según la ecuación vista previamente.

Apéndice C

Configuración de OpenSSH para RoboComp

OpenSSH es una herramienta que nos permite establecer conexiones shell de forma segura, permite cifrar el tráfico y autenticar los extremos que mantienen la comunicación. En este apéndice se verá cómo configurarlo para conseguir que el uso de managerComp sea más cómodo.

Cuando se usan componentes Ice es muy común querer arrancar con managerComp un componente que no está en la máquina en la que se está sentado. En estos casos se suele configurar managerComp para que ejecute un comando remoto usando ssh a la hora de apagar o encender un componente. Por ejemplo, en el campo *up command*:

```
ssh robot /usr/bin/camaraComp --Ice.Config=/home/usuario/cam.conf
```

En el apéndice se tratan dos problemas comunes: el primero es la autenticación automática sin necesidad de escribir el password para cada comando, el segundo es el retardo de la conexión.

El primer problema tiene fácil solución. Normalmente el servidor de ssh viene configurado para tratar de obtener los nombres de los hosts que se conectan a el. Desactivando esta opción se conseguirá reducir la latencia de la ejecución de los comandos. Para solventar el problema sólo se ha de buscar y cambiar a "no" cualquiera las siguientes lineas que se encuentren en el fichero de configuración */etc/ssh/sshd_config*:

```
UseDNS yes
VerifyReverseMapping yes
```

Para resolver el problema de la autenticación se ha de crear una “llave de autenticación” y copiarla en cada una de las máquinas en las que se quiera ejecutar comandos remotos sin escribir el password. Para crear la llave se usa el ssh-keygen. Se ejecuta sin argumentos, se acepta el path de la llave por defecto y se deja el password vacío (de no dejar el password vacío lo que se conseguiría es poder usar un único password para diferentes máquinas):

```
persona@pclocal\$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/persona/.ssh/id_rsa):
Created directory '/home/persona/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/persona/.ssh/id_rsa.
Your public key has been saved in /home/persona/.ssh/id_rsa.pub.
The key fingerprint is:
7d:76:54:d5:98:69:0d:3b:2d:ff:cb:2b:f9:8c:d7:99 persona@pclocal
```

Con esto se tendría un par de llaves (privada/pública) en /home/persona/.ssh. El segundo paso es copiar la llave pública a cada una de las máquinas en las que se quiera ejecutar comandos remotos (si es que hay más de una). Para ello se utiliza ssh-copy-id:

```
persona@local\$ ssh-copy-id usuario@robot
The authenticity of host 'robot (111.111.111.111)' can't be establishe
RSA key fingerprint is 4a:10:9d:14:94:18:f9:9b:ac:48:d3:f4:0c:67:dc:dd
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'robot,111.111.111.111' (RSA) to the list o
usuario@robot's password:
Now try logging into the machine, with "ssh 'usuario@robot'", and chec

  .ssh/authorized_keys

to make sure we haven't added extra keys that you weren't expecting.
```

Es importante tener en cuenta que, con esto, la seguridad del acceso a la máquina remota también depende en el secreto de la clave privada. Por tanto, es conveniente vigilar los permisos de los ficheros de las llaves que se encuentran en /home/persona/.ssh/.

Bibliografía

[1] Y.I. Abdel-Aziz and H.M. Karara. Direct linear transformation from comparator coordinates into object space coordinates in close-range photogrammetry. proceedings of the symposium on close-range photogrammetry. *American Society of Photogrammetry.*, 1971.

[2] W. Ritter B. Heisele. Obstacle detection based on color blob flow. *Intelligent Vehicles Symposium*, 1995.

[3] Parag Batavia and Sanjiv Singh. Obstacle detection using adaptive color segmentation and color stereo homography. *International Conference on Robotics and Automation*, 2001.

[4] Pilar Bachiller Burgos. Percepción dinámica del entorno en un robot móvil. *Tesis doctoral*, 2008.

[5] A J Davison and D W Murray. Simultaneous localisation and map-building using active vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2002.

[6] J. Gaspar, J. Santos-Victor, and J. Sentieiro. Ground plane obstacle detection with a stereo vision system. 1994.

[7] Richard Hartley and Andrew Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2004.

[8] Michi Henning and Mark Spruiell. *Distributed Programming with Ice.*, revision 3.3.0 edition, 2008.

[9] H.F Leonard, J.J. Durrant-whyte. Simultaneous map building and localization for an autonomous mobile robot. *International Conference on Intelligent Robots and Systems*, 1991.

[10] Pilar Bachiller Luis J. Manso, Pablo Bustos and José Moreno. Obstacle detection on heterogeneous surfaces using color and geometric cues. In *Actas del X Workshop de Agentes Físicos. Cáceres*, Septiembre 2009.

[11] Luis J. Manso Pilar Bachiller, Pablo Bustos. *Advances in Robotics, Automation and Control*, chapter Attentional Selection for Action in Mobile Robots, pages 111–136. I-Tech, 2008.

[12] Robolab. Robex arena - http://robexarena.com. 2009.

[13] Robolab. Robocomp project - http://robocomp.wiki.sourceforge.net. 2009.

[14] Agustín Sánchez Domínguez. Diseño y construcción de un controlador de motores dc basado en microcontroladores. 2007.

[15] Carsten Bruckhoff Thomas Bergener and Christian Igel. Evolutionary parameter optimization for visual obstacle detection. *The International Institute for Advanced Studies in Systems Research and Cybernetics*, 2000.

[16] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. Probabilistic robotics (intelligent robotics and autonomous agents). 2005.

[17] Iwan Ulrich and Johann Borenstein. Vfh*: Local obstacle avoidance with look-ahead verification, 2000.

[18] Iwan Ulrich and Illah Nourbakhsh. Appearance-based obstacle detection with monocular color vision. *Proceedings of the AAAI National Conference on Artificial Intelligence*, 2000.

[19] Zhu Changan Yong Chao. Obstacle detection using adaptive color segmentation and planar projection stereopsis for mobile robots. *International Conference on RoboticsJntelligentSystems and Signal Processing*, 2003.

[20] Jin Zhou and Baoxin Li. Homography-based ground detection for a mobile robot platform using a single camera. *International Conference on Robotics and Automation, IEEE*, 2006.

Índice de figuras

1.1. Robot RobEx. 12

2.1. Chasis de RobEx 21
2.2. Soportes de los motores de RobEx 22
2.3. Casquillo de RobEx 22
2.4. RobEx básicos 23
2.5. Componentes de RoboComp. 27
2.6. Captura de pantalla de managerComp. 30

3.1. Filtros polarizadores. 36
3.2. Componentes de RoboComp necesarios para el funcionamiento del proyecto. 37
3.3. Diagrama de clases del componente genérico genericComp. . . . 46

4.1. Imágenes del par estéreo acompañadas del warping de la cámara izquierda. 51
4.2. Imágenes derecha e izquierda transformada, acompañadas del valor absoluto de su resta. 53
4.3. Situación problemática para clasificar por geometría. Obstáculo grande sin textura. 58
4.4. Comparación entre el poder discriminativo de un histograma de dos dimensiones y el de dos histogramas de una dimensión. . . . 60
4.5. Situación problemática para clasificar por apariencia. Hoja en el suelo. 61
4.6. Situación problemática para clasificar por apariencia. Cartón en la pared. 62

4.7. Captura de pantalla de los resultados de floorColorCueComp. . 63
4.8. Cambio de coordenadas de imagen a coordenadas polares tomando como sistema de referencia el robot. 65
4.9. Captura de pantalla de floorComp. 66

5.1. Captura de pantalla del calibrador automático de homografias. . 71
5.2. Parte de la trayectoria realizada por el robot durante la prueba de los 15 minutos. 74
5.3. Capturas de floorComp durante los experimentos 75
5.4. Captura de pantalla de floorPPSComp sobre suelo rugoso. . . . 77

6.1. Diálogo de configuración de managerComp. 89
6.2. Interfaz de managerComp en modo grafo. 90
6.3. [Interfaz de managerComp en modo lista. 91

A.1. Modelo de base robótica diferencial. 96
A.2. Ejemplos de ruedas de apoyo. 97

B.1. Relación de homografía. 100

Índice de cuadros

1.1. Consumos de distintos dispositivos láser y cámaras. 14