Departamento de
Informática e Ingeniería
de Sistemas
Universidad Zaragoza
1542

ROBOTICS
UNIVERSIDAD DE ZARAGOZA

Trabajo Fin de Grado

Construcción de un mapa incluyendo objetos reconocidos desde un robot basado en Raspberry Pi

Map building including recognised objects from a Raspberry Pi based robot

Autor

Víctor José Moya Raso

Directores

Ana Cristina Murillo Arnal

Danilo Tardioli

Grado en Ingeniería Electrónica y Automática
Departamento de Informática e Ingeniería de sistemas
Escuela de Ingeniería y Arquitectura, Universidad de Zaragoza
2018

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

D./Dª. Víctor José Moya Raso,

con nº de DNI 77135697E en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) Grado, (Título del Trabajo) Construcción de un mapa incluyendo objetos reconocidos desde un robot basado en Raspberry Pi,

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 30 de Mayo de 2018

Fdo: Víctor José Moya Raso

Agradecimientos

Aprovecho este espacio para dar las gracias a mis directores del proyecto. A Ana por estar siempre dispuesta a ayudarme, buscando cualquier momento para atender mis dudas, a pesar de las trabas que nos ponía Gmail en los días más críticos. A Danilo por prestarse siempre a enseñarme algo nuevo y guiarme en el mundillo de la informática. También agradecer a mi familia y amigos por todo el apoyo mostrado a lo largo de estos años.

Resumen

El presente trabajo se centra en el estudio de las posibilidades de llevar a cabo tareas realizadas por robots que integran hardware de alta gama, en una plataforma robótica de bajo coste. De esta forma se podría implementar el estudio de dichas tareas en asignaturas impartidas en la Universidad de Zaragoza. El proyecto consta de varias fases, tal como se describe en la presente memoria.

En primer lugar, se aborda la selección del hardware de acuerdo al objetivo de elaborar una plataforma robótica de bajo coste y con el conocimiento de las necesidades requeridas para llevar a cabo las tareas pretendidas. De acuerdo con el hardware seleccionado se escoge el software apropiado que permitirá la realización de las tareas. Para llevar a cabo el proyecto se hace uso de ROS, un entorno de trabajo ampliamente utilizado en proyectos robóticos, debido a que permite el desarrollo de aplicaciones de forma modular a la vez que facilita la reutilización de software y la integración del hardware. La segunda parte del proyecto consiste en el estudio de los métodos necesarios para llevar a cabo las tareas del proyecto. Las distintas tareas se plantean individualmente pero juntas conforman la aplicación final elaborada en el trabajo. Dichas tareas abordan el cálculo de la odometría, la construcción de un mapa del entorno con localización del robot respecto del mapa (SLAM) y la localización de vehículos en el mapa elaborado.

En la parte final de la memoria se describe la evaluación y validación detallada de todos los módulos del proyecto: evaluar los errores de odometría obtenidos, el rendimiento de la plataforma robótica durante la realización de la aplicación, la precisión en la elaboración del mapa y la tasa de acierto en la detección de vehículos. Mediante estos análisis se ha comprobado que la odometría solo puede ser utilizada de forma complementaria a otros sensores debido al error procedente de la misma. Una conclusión importante de los experimentos es que se ha verificado el correcto funcionamiento de Raspberry Pi 3 al ejecutar varias tareas en ROS de manera simultánea, con la excepción de los riesgos procedentes de una excesiva temperatura, la cual debe ser controlada. Por otra parte, se ha concluido que *GMapping* presenta una mayor precisión en la construcción del mapa respecto a *Cartographer*. Sin embargo, *Cartographer* muestra menores exigencias de cómputo del procesador, por lo que se requiere menos tiempo para la ejecución de la tarea de SLAM. Por último, se ha entrenado y evaluado el clasificador *AdaBoost* basado en *Haar features*, destinado a la detección de vehículos en el entorno de experimentación, y se ha integrado con el resto del sistema para anotar el mapa con las posiciones donde se han ido detectando los coches.

Índice general

1. Introducción **1**
1.1. Motivación 1
1.2. Objetivos y planificación 3
1.3. Entorno de trabajo 5
1.4. Trabajos relacionados 5
1.5. Organización del proyecto 7

2. Plataforma robótica **9**
2.1. Hardware 9
2.2. Software 14

3. Arquitectura del sistema **17**
3.1. Módulo de desplazamiento 18
3.2. Módulo de mapeado 20
3.3. Módulo de detección 22
3.4. Integración 24

4. Evaluación y Experimentación **25**
4.1. Análisis de la odometría 26
4.2. GMapping vs Cartographer 28
4.3. Análisis de la detección de vehículos 33
4.4. Aplicación final 35

5. Conclusiones **38**

Anexos **41**

A. Manual de uso **44**
A.1. Instalación del software 44
A.2. Comunicación del PC con el Robot 46
A.3. Lanzamiento de la aplicación 47

B. Cámara RGB-D **49**

Lista de Figuras

1.1. Plataformas robóticas comerciales de bajo coste 1
1.2. Estudio de aplicabilidad del entorno ROS en Raspberry Pi 2
1.3. Escenario real y vehículos a detectar 3
1.4. Diagrama de Gantt del desarrollo de este proyecto 5
1.5. Ejemplos de tareas de reconocimiento visual 6
1.6. Mapa anotado con información semántica de habitaciones reconocidas . 7

2.1. Computadores de placa reducida Raspberry Pi 10
2.2. Motores y placas de adaptación 11
2.3. Sensores considerados 11
2.4. Esquema de conexión del hardware de la plataforma robótica 12
2.5. Plataforma robótica elaborada en el proyecto 13
2.6. Esquema de alto nivel del software utilizado en el proyecto 15

3.1. Esquema conceptual de los módulos del sistema 17
3.2. Modelo cinemático de un robot de tracción diferencial 19
3.3. Diagrama de movimientos posibles por teleoperación 20
3.4. Máscaras utilizadas para las Haar features 22
3.5. Ejemplos de imágenes utilizadas en el reconocimiento de vehículos . . . 23
3.6. Gráfico del funcionamiento global del sistema 24

4.1. Mapa modelo recreado en el laboratorio 25
4.2. Representación de las trayectorias de prueba 27
4.3. Evolución temporal de la temperatura en la CPU 29
4.4. Evolución temporal de la memoria RAM en uso 30
4.5. Tiempo de la CPU dedicado a la gestión de código de usuario 31
4.6. Entorno de experimentación con anotación de esquinas 32
4.7. Mapas obtenidos a la resolución de 0,01m/píxel 32
4.8. Construcción del mapa con anotación semántica de vehículos detectados 36
4.9. Construcción del mapa incluyendo la trayectoria realizada 36
4.10. Proceso completo de construcción del mapa anotado 37

B.1. Asus Xtion PRO Live 49
B.2. Imágenes captadas por cámara RGB-D 50
B.3. Imagen de profundidad convertida a escaneo láser 50

Lista de Tablas

2.1. Coste del hardware seleccionado para el proyecto 13

4.1. Error procedente de la odometría 27
4.2. Porcentajes desglosados de la memoria RAM en uso 30
4.3. Error obtenido en la construcción del mapa 33
4.4. Evaluación de los clasificadores obtenidos 34

Capítulo 1

Introducción

1.1. Motivación

En los últimos años hemos sido testigos del gran avance que han experimentado los computadores, siendo de especial interés en este trabajo, las placas de prototipado rápido con muy bajo coste, como Raspberry Pi o Arduino. Gracias a su mejora en la capacidad de procesamiento y a la aparición de sensores económicos con prestaciones superiores, se ha hecho posible la realización de sistemas de bajo coste que lleven a cabo aplicaciones robóticas muy diversas. De hecho, hoy en día ya se comercializan plataformas robóticas basadas en computadores de este tipo con un bajo coste, las cuales tienen, generalmente, un enfoque educativo. En la Figura 1.1 se muestran algunos ejemplos de estas plataformas robóticas de bajo coste.

Figura 1.1: Plataformas robóticas comerciales de bajo coste. De Izquierda a derecha: Robot Arduino, Robot LEGO Mindstorms y Robot Makeblock[1].

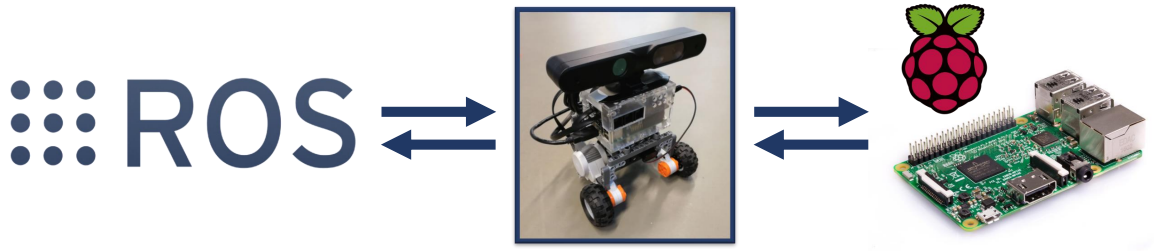


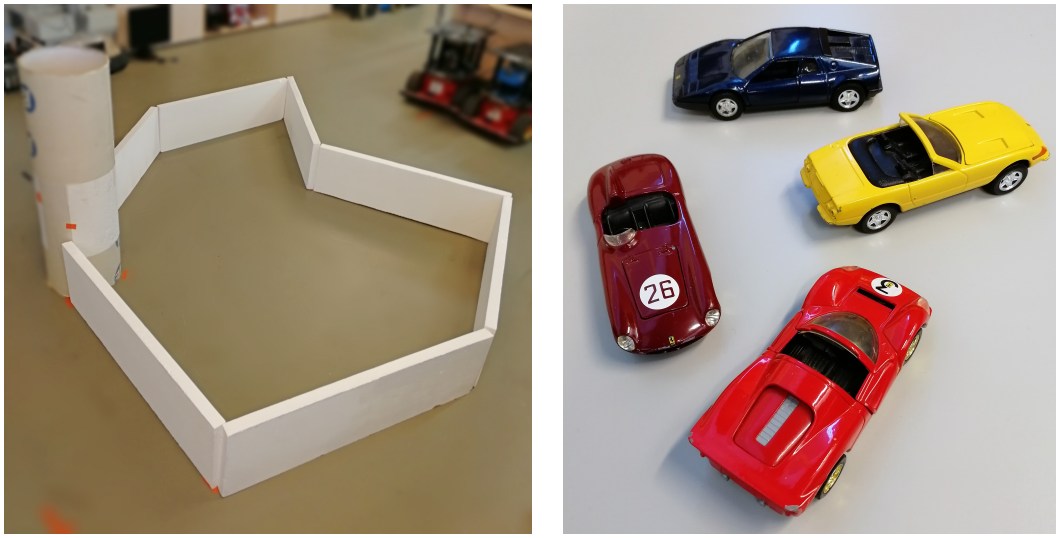
Figura 1.2: Estudio de las posibilidades de aplicabilidad del entorno ROS en un sistema robótico de bajo coste basado en Raspberry Pi.

Con el conocimiento de que actualmente las tareas robóticas son ejecutadas por robots que integran hardware de un elevado coste, en este proyecto se pretende estudiar las capacidades de una plataforma robótica (como las mostradas en la Figura 1.1) de llevar a cabo la realización de tareas robóticas comunes en entornos reales. De acuerdo con lo mostrado en la Figura 1.2, se quiere hacer uso de ROS (*Robot Operating System*), entorno de trabajo muy utilizado en sistemas robóticos que incluye software que permite desarrollar aplicaciones de forma modular, además de proveer una serie de *packages* para la realización de tareas tales como localización y mapeo simultáneo, planificación, etc.

El uso de ROS en una computadora de placa reducida como Raspberry Pi, junto con los sensores y actuadores necesarios, permitirá desarrollar una plataforma robótica de bajo coste en la cuál integrar diversas tareas. Estas tareas abarcan tanto el campo de la robótica como el campo de la visión por computador, que se benefician mutuamente, dotando al proyecto desarrollado de una mayor funcionalidad.

Por una parte, dicha funcionalidad consiste en la integración de tareas puramente robóticas como el cálculo de la odometría, es decir, la estimación de la posición del robot durante la navegación y por ello se pretende el desarrollo de una plataforma dotada de movilidad. Además, se plantea la consecución de tareas más avanzadas, como es la construcción de un mapa del entorno y la localización del robot en el mismo a partir de la información procedente de los sensores. En este proyecto se trabajará sobre un escenario modelo, representado en la Figura 1.3 (a), que servirá para facilitar la repetibilidad de los experimentos. Por otra parte, se quiere complementar con la adaptación de técnicas de visión por computador en la plataforma robótica para añadir información semántica al mapa del entorno. En particular, el trabajo se centra en la detección de vehículos como los mostrados en la Figura 1.3 (b).

[1]Robot Arduino, recuperado de `https://www.arduino.cc/en/Main.Robot/`.
Robot LEGO, recuperado de `https://shop.lego.com/en-US/LEGO-MINDSTORMS-NXT-2-0-8547`.
Robot Makeblock, recuperado de `https://www.makeblock.es/productos/mbot_ranger/`.



(a) (b)

Figura 1.3: (a) Escenario modelo de trabajo que facilita la repetibilidad de los ensayos, (b) Vehículos a detectar que constituyen la información semántica del mapa elaborado.

Una de las claves que han motivado el desarrollo del proyecto es el objetivo de estudiar la aplicabilidad del prototipo construido en asignaturas impartidas en la Universidad de Zaragoza. El uso de ROS en estas plataformas de bajo coste permitiría reproducir de manera práctica tareas ejecutadas por robots de gama alta.

1.2. Objetivos y planificación

El objetivo principal del proyecto es estudiar las capacidades de un robot móvil de bajo coste de realizar tareas llevadas a cabo por robots de gama alta. Estas tareas engloban tanto la aplicación de técnicas de SLAM (*Simultaneous Localization And Mapping*), que consisten en la construcción de un mapa del entorno y la localización en el mismo, como la aplicación de técnicas de visión por computador para el reconocimiento de objetos. Las tareas realizadas para la consecución de este objetivo son las siguientes:

- Instalación y familiarización con ROS, empleado para el desarrollo de aplicaciones robóticas tanto a nivel académico como industrial. Para la familiarización con el entorno de ROS se han desarrollado los tutoriales de su *wiki*[2], gracias a

[2] `http://wiki.ros.org/ROS/Tutorials/`

los cuales se han adquirido los conceptos básicos de funcionamiento, explicados en profundidad en la sección 2.2.

- Estudio de las técnicas de SLAM disponibles en ROS para la construcción de un mapa 2D del entorno: *GMapping*[3] [1], *Hector SLAM*[4] [2] o *Cartographer*[5] de Google [3]. Estudio del funcionamiento básico de estas técnicas y la aplicación de las mismas haciendo uso de las integraciones existentes en ROS.

- Instalación y familiarización con OpenCV, biblioteca ampliamente utilizada en aplicaciones de visión por computador. En concreto este estudio se centra en su instalación en Raspberry Pi y en la asimilación de las funcionalidades que proporciona *OpenCV*[6] para la detección de objetos, permitiendo adaptar tanto el sistema de entrenamiento como de clasificación a la tarea de detección de este proyecto.

- Adaptación del clasificador estudiado a la detección de vehículos por medio de la recopilación de los datos necesarios para su entrenamiento. Evaluación de la efectividad del clasificador obtenido e integración de dicho clasificador para la localización del vehículo en el mapa.

- Integración de la placa de conexión de productos de LEGO Mindstorms con la Raspberry Pi, permitiendo el control de los actuadores para dotar al robot de la movilidad necesaria.

- Diseño de la plataforma robótica y su consiguiente desarrollo con la integración del hardware necesario para la consecución de los objetivos fijados.

- Realización de pruebas experimentales sobre la plataforma robótica elaborada, obteniendo resultados sobre la viabilidad de los métodos estudiados y demostración de la aplicación final desarrollada que engloba las distintas tareas a ejecutar.

- Documentación de la memoria del proyecto, realizando una recopilación de los ensayos efectuados, y realización del manual de usuario como punto de partida para el desarrollo de futuros estudios o aplicaciones relacionadas con este proyecto. El manual de usuario se encuentra en el anexo A.

La organización temporal de las tareas descritas se encuentra reflejada en el diagrama de Gantt, representado en la Figura 1.4.

[3] `http://wiki.ros.org/gmapping/`
[4] `http://wiki.ros.org/hector_slam/`
[5] `http://wiki.ros.org/cartographer/`
[6] `https://docs.opencv.org/3.4.1/d2/d64/tutorial_table_of_content_objdetect.html`

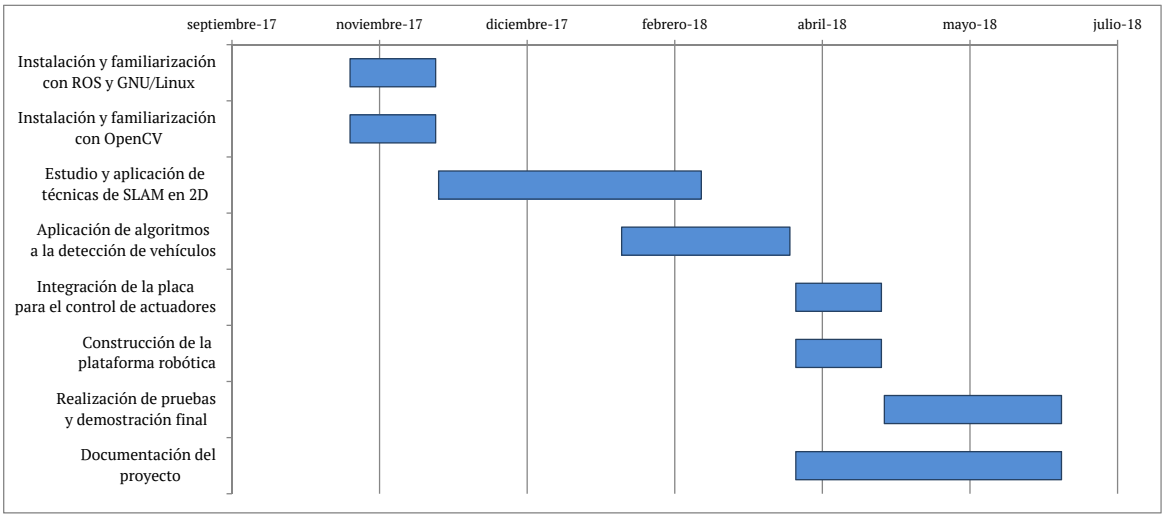


Figura 1.4: Diagrama de Gantt del desarrollo de este proyecto.

1.3. Entorno de trabajo

Este proyecto se ha llevado a cabo en el entorno de trabajo de ROS, sobre el sistema operativo Linux, en particular la distribución Ubuntu. Tanto el software como el hardware empleados se encuentran ampliamente explicados en el capítulo 2.

Este proyecto ha sido realizado en el laboratorio del grupo de Robótica, Percepción y Tiempo Real (*RoPeRT*)[7] del Instituto Universitario de Investigación en Ingeniería de Aragón.

1.4. Trabajos relacionados

A partir de la reciente evolución de computadores como Raspberry Pi y Arduino, es cada vez más frecuente la realización de proyectos que pretenden desarrollar diversas aplicaciones en plataformas robóticas y de esta forma explorar las capacidades de un robot de bajo coste. A continuación, se recogen algunos de los trabajos de investigación especialmente interesantes para este proyecto, que se basan en el estudio de diversos algoritmos de robótica y visión por computador en plataformas robóticas de bajo coste.

[7] `http://robots.unizar.es`

[8] Imagen de `http://blog.electricbricks.com/2010/05/reconocer-objetos-con-opencv/`

Localización y Mapeo Simultáneos en 2D. Uno de los objetivos de este trabajo consiste en estudiar la ejecución en Raspberry Pi de técnicas SLAM [4] para la construcción de un mapa y la localización simultánea del robot en el mismo. Numerosos trabajos se centran en el mismo objetivo, buscando a su vez el mejor rendimiento posible. Un trabajo cercano a este proyecto se centra en el estudio del rendimiento de Raspberry Pi al ejecutar el algoritmo de GMapping [1] sobre la plataforma robótica de bajo coste TurtleBot [5].

Otro estudio se centra en evaluar algoritmos de SLAM [6]; de especial interés para este proyecto la evaluación de GMapping y Hector SLAM [2]. Este último permite realizar la tarea de SLAM sin necesidad de odometría, lo que supone la ventaja de la disminución de carga computacional. Sin embargo, el hecho de no disponer de odometría unido a la carencia de implementación de cierre de bucle repercute en una disminución de precisión respecto otros algoritmos de SLAM.

Muchos de estos trabajos, entre ellos el [5], construyen el mapa a partir de sensores económicos, basados en visión, como cámaras RGB-D; éstas permiten llevar a cabo la construcción del mapa 2D convirtiendo los datos 3D del sensor de profundidad de la cámara en una medida similar a la que se obtiene por medio del uso de un LIDAR (*Light Detection and Ranging*) [7]. Además de la ventaja en el coste, un sensor RGB-D permite la detección de obstáculos a diferentes alturas para evitar la colisión con estos, mientras que un LIDAR proporciona la detección en un único plano horizontal.

(a) (b)

Figura 1.5: Ejemplos de tareas de reconocimiento visual. (a) Reconocimiento de rostros, (b) Plataforma robótica de bajo coste con reconocimiento de objetos[8].

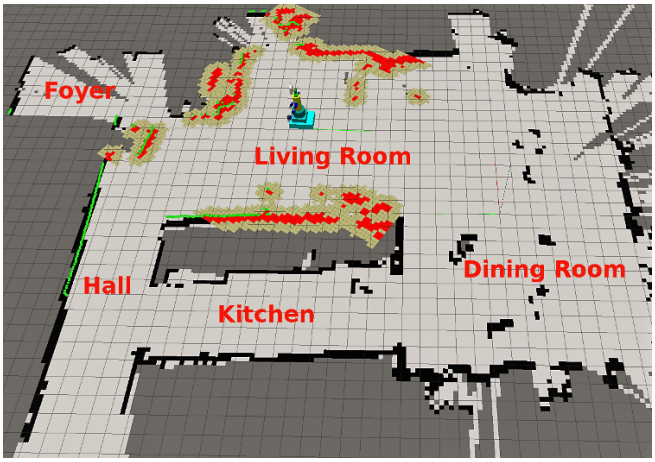


Figura 1.6: Mapa construido por un robot móvil con información semántica de las habitaciones reconocidas[9].

Reconocimiento Visual. El campo de visión por computador está presente en diversos proyectos, que se centran en dar mayor funcionalidad a las plataformas robóticas a través de aplicaciones de reconocimiento visual automático. Por ejemplo, algunos proyectos pretenden la interacción con las personas mediante la detección de rostros a través de algoritmos implementables en plataformas de bajo coste [8], tal como se muestra en la Figura 1.5 (a). La Figura 1.5 (b) muestra otro ejemplo de plataforma de bajo coste diseñada para la detección de objetos. Encontramos ejemplos de esta tarea tanto en entornos industriales [9] como en entornos educativos [10].

Otras aplicaciones basadas en visión por computador de las que encontramos ejemplos en plataformas de bajo coste incluyen el reconocimiento de escenas en lugar de simples objetos (lo que permite llevar a cabo la construcción de mapas anotados según el tipo de estancia, como el realizado por la plataforma Pi Robot de la Figura 1.6), o aplicaciones de seguimiento visual, como el ejemplo de recuento de personas [11].

1.5. Organización del proyecto

En el presente capítulo se han introducido tanto las motivaciones del proyecto como los objetivos del mismo, estableciendo los pasos a seguir para su consecución. En el capítulo 2 se analiza el diseño de la plataforma robótica, comenzando por la justificación del hardware utilizado y terminando por el software implementado. En el

[9]Imagen de `http://www.pirobot.org/blog/0015/`

capítulo 3 se explica la integración del proyecto en la plataforma robótica, así como su funcionamiento, comentando los principales módulos del sistema. En el capítulo 4 se recogen los resultados obtenidos en los distintos ensayos y el desarrollo de la aplicación final. Por último, en el capítulo 5 se exponen las conclusiones extraídas en el trabajo, además de posibles mejoras a implementar en trabajos futuros.

Capítulo 2

Plataforma robótica

Este capítulo describe la plataforma robótica desarrollada tanto a nivel hardware como a nivel de software. En primer lugar, se realiza una comparativa de las opciones estudiadas para el hardware del proyecto. En segundo lugar, se resumen las decisiones en la selección del software teniendo en cuenta el hardware disponible.

2.1. Hardware

Esta sección describe el diseño y la decisión de los principales componentes de la plataforma robótica desarrollada en este proyecto, entre ellos, el computador utilizado, los sensores empleados y el sistema que permite el movimiento del robot. Destacar que la elección de los componentes ha estado determinada por la premisa de diseñar un robot de bajo coste.

Computador. Desde el inicio del desarrollo del proyecto se tuvo en cuenta que el computador utilizado debía ser de tamaño y coste reducidos: un computador que pudiese incorporarse en plataformas robóticas pequeñas, permitiendo el estudio de diversas técnicas llevadas a cabo por robots de alta gama. Una de las placas de prototipado rápido más utilizadas en el desarrollo de proyectos robóticos es Raspberry Pi. La razón de su utilización son las ventajas que presenta respecto a otros computadores de placa reducida. Por ejemplo, Raspberry Pi presenta una mayor potencia de cálculo que Arduino y una mayor facilidad en la conexión del hardware que BeagleBone.

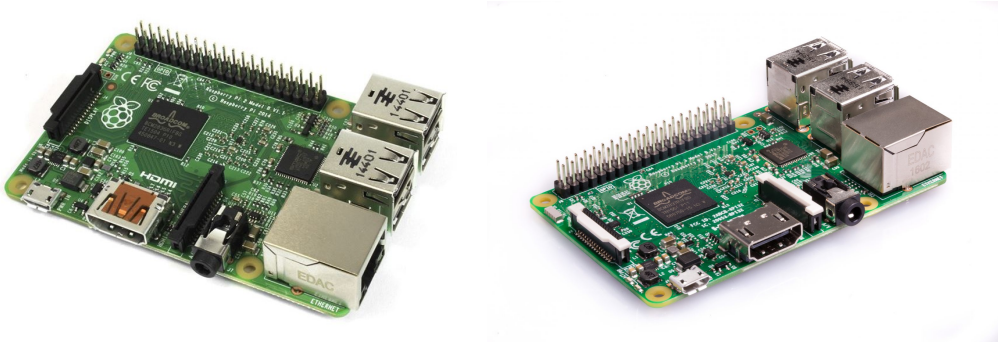


Figura 2.1: Computadores de placa reducida utilizadas en este proyecto. De izquierda a derecha: Raspberry Pi 2 Modelo B (2015) y Raspberry Pi 3 Modelo B (2016)[1].

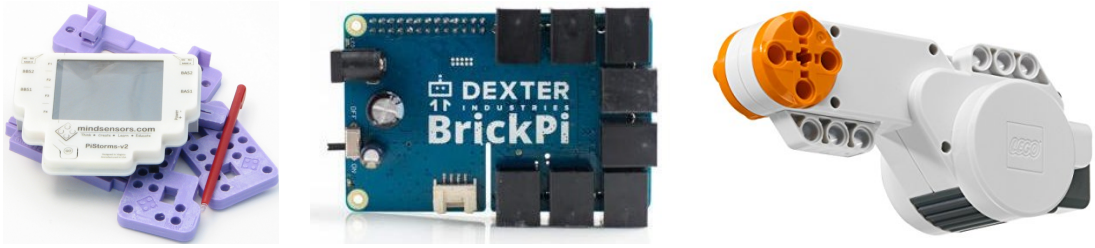
Debido a esto, desde las etapas iniciales de este proyecto se trabajó sobre Raspberry Pi 2 Modelo B, que presenta un procesador *Quad-Core* de *900MHz*. Sin embargo, la necesidad de mayor rendimiento determinó el uso de Raspberry Pi 3 Modelo B, ya que presenta un evolución de dicho procesador, manteniendo una estructura *Quad-Core*, pero incrementando la velocidad de *900MHz* a *1,20GHz*.

Otra razón del cambio es la incorporación de conectividad que obtuvo Raspberry Pi 3 Modelo B en su lanzamiento, incluyendo *Wi-Fi* y *Bluetooth*, esto evita el uso de adaptadores externos con el consiguiente ahorro en el coste de desarrollo del proyecto. Ambas versiones de Raspberry se muestran en la Figura 2.1.

Motores y Placas de Adaptación. A la hora de dotar de movilidad se optó por emplear los motores de LEGO Mindstorms, como el mostrado en la Figura 2.2 (c) y para conformar la estructura del robot se utilizaron piezas de LEGO. La decisión de utilizar estos productos estuvo condicionada por su reducido precio, además de permitir libertad en la ejecución del diseño estructural del robot. Cabe mencionar que los actuadores de LEGO Mindstorms empleados incluyen encoders, utilizados en el proyecto para realizar el cálculo de la odometría, explicado en la sección 3.1.

Para establecer la comunicación entre el computador del robot y los productos de LEGO Mindstorms, se estudió el uso de dos módulos. Estos módulos hacen el papel del Mindstorms Brick, cerebro del sistema Mindstorms que permite el desarrollo de aplicaciones en las plataformas robóticas de LEGO, como la mostrada en la Figura 1.1. Comentar que el uso de Mindstorms Brick se descartó debido a su coste aproximado

[1]Raspberry Pi 2B (`https://www.raspberrypi.org/products/raspberry-pi-2-model-b/`). Raspberry Pi 3B (`https://www.raspberrypi.org/products/raspberry-pi-3-model-b/`).



(a) (b) (c)

Figura 2.2: Placas de adaptación a Raspberry Pi de componentes LEGO Mindstorms y motor de LEGO Mindstorms empleado en el proyecto. (a) Controlador PiStorms-v2, (b) Módulo BrickPi3 y (c) Motor LEGO Mindstorms[2].

de 200€, superior al coste combinado de Raspberry Pi 3 y uno de los módulos estudiados, que se encuentra en torno a los 120€. Los módulos estudiados son el controlador PiStorms y el módulo BrickPi, mostrados en las Figuras 2.2 (a) y 2.2 (b) respectivamente. Ambos son totalmente aptos, ya que tienen un funcionamiento correcto y son compatibles con Raspberry Pi 3.

La diferencia más notable entre ambos módulos reside en la robustez que presenta BrickPi frente a PiStorms, ya que BrickPi permite el desarrollo de una estructura más resistente. Por ello, aunque PiStorms posee una pantalla con la que interactuar, la cual provee mayor funcionalidad al usuario, se optó por emplear el módulo BrickPi3.

Sensores. En el desarrollo del proyecto se estudió la utilización de distintos sensores, los cuales se fueron descartando por diversos motivos como restricciones de peso, coste o requisitos funcionales del sistema. Los sensores que se estudiaron para su inclusión en la plataforma robótica son LIDAR y cámaras.

(a) (b) (c)

Figura 2.3: Sensores considerados para la plataforma robótica. (a) LIDAR Hokuyo y RaspiCAM, (b) Microsoft Kinect, (c) Asus Xtion PRO Live[3].

[2]PiStorms-v2 (`http://www.mindsensors.com/content/78-pistorms-lego-interface`). BrickPi3 (`https://www.dexterindustries.com/new-brickpi3-lego-mindstorms/`).

En primer lugar, se barajó el uso de un LIDAR ya que es una opción robusta para la realización de mapeados 2D de entornos interiores donde el sensor proporciona buenos resultados. Debido a que la realización del mapa no es el único objetivo de este trabajo, sería necesario complementar al LIDAR con una cámara que realizase la detección de vehículos. Por ello se planteó utilizar el último modelo de cámara de Raspberry Pi, que es muy económica, alrededor de 25€. Sin embargo, el precio del LIDAR disparaba el coste del proyecto por encima de los 1000€, descartando esta posibilidad. Ambos sensores se muestran en la Figura 2.3 (a).

La segunda opción que se planteó fue emplear una cámara RGB-D que fuese capaz de realizar ambos cometidos, tanto el mapeado del entorno como la detección de vehículos y de esta forma conseguir una mayor compactación del hardware necesario. Este tipo de cámaras combinan un sensor de infrarrojos para la detección de profundidad y un sensor RGB para la obtención de la imagen en color. Los dos cámaras RGB-D estudiadas para su utilización fueron la Microsoft Kinect y la ASUS Xtion PRO Live, mostradas en las Figuras 2.3 (b) y 2.3 (c) respectivamente. Ambas son similares en cuanto al coste, pero la ASUS Xtion PRO Live presenta una serie de características que determinaron que fuese escogida para la realización del proyecto.

La principal diferencia es el tamaño de ambas, donde la Asus Xtion presenta unas dimensiones de 178 mm x 51 mm x 38 mm, mientras que Microsoft Kinect tiene unas

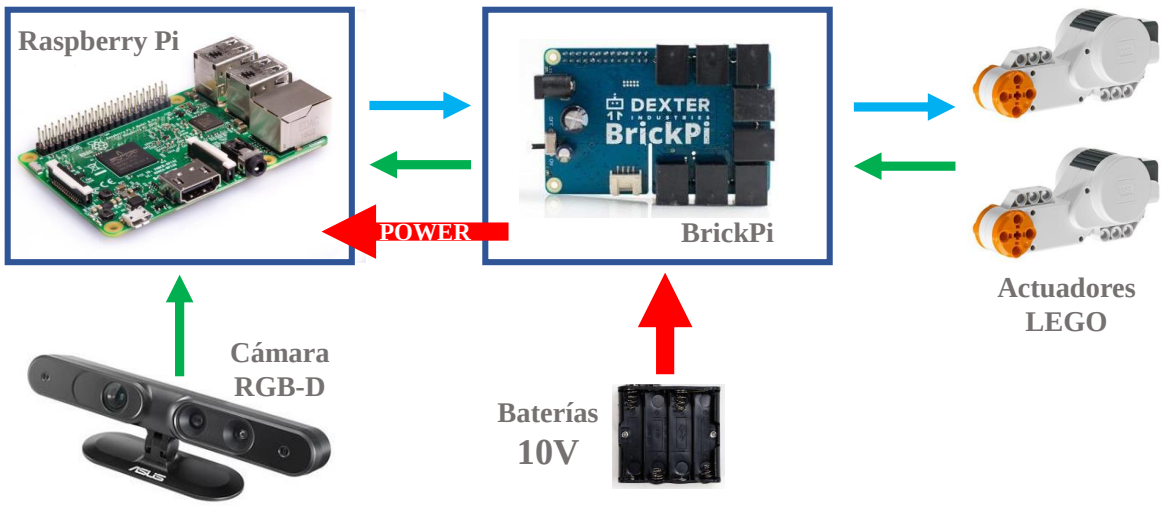


Figura 2.4: Esquema de conexión del hardware de la plataforma robótica.

[3]LIDAR Hokuyo, imagen de `https://www.hokuyo-aut.jp/`
RaspiCAM, imagen de `https://www.raspberrypi.org/products/camera-module-v2/`
Microsoft Kinect, imagen de `https://en.wikipedia.org/wiki/Kinect`
Asus Xtion PRO Live, imagen de `https://www.asus.com/es/3D-Sensor/Xtion_PRO_LIVE/`

dimensiones de $305\,mm$ x $76\,mm$ x $64\,mm$. Esta diferencia es proporcional a la diferencia en el peso donde la ASUS Xtion presenta un peso de $0,225\,Kg$, mientras que la Microsoft Kinect tiene un peso de $1,360\,Kg$. Dado que la plataforma robótica es de pequeñas dimensiones la diferencia presentada es notoria y determinante en la elección. Además, la Asus Xtion no requiere de alimentación externa, con el único requerimiento de conexión vía USB, por su parte la Microsoft Kinect sí requiere de alimentación externa, lo que conlleva mayor consumo y peso en la plataforma robótica. Por último, la cámara de ASUS presenta mayor invarianza en la precisión de las medidas de profundidad [12], lo que supone la obtención de mejores resultados.

La Tabla 2.1 recoge el coste aproximado de los elementos que componen la plataforma robótica diseñada. El coste total refleja la diferencia con otras plataformas robóticas, como el ejemplo de TurtleBot, cuyo coste aproximado, equipando la plataforma con sensores de gama media, es de más de 1000€.

Producto	**Precio**
Raspberry Pi 3 Model B	37,90 €
BrickPi3	86,50 €
Asus Xtion PRO Live	160,00 €
Motores y piezas LEGO	65,00 €
Coste Total	**349,40 €**

Tabla 2.1: Coste aproximado del hardware seleccionado para el desarrollo del proyecto.

Con los elementos seleccionados se realizan las conexiones tal como se muestra en la Figura 2.4. De este modo la cámara RGB-D se conecta directamente a la Raspberry Pi 3, mientras que los actuadores del sistema se conectan al módulo BrickPi3 el cuál hace de intermediario con la Raspberry Pi 3. Por último, mencionar que el sistema se alimenta directamente a través de BrickPi3 con $10\,V$. Desarrollando el diseño estructural con el hardware escogido se obtiene la plataforma robótica de la Figura 2.5.

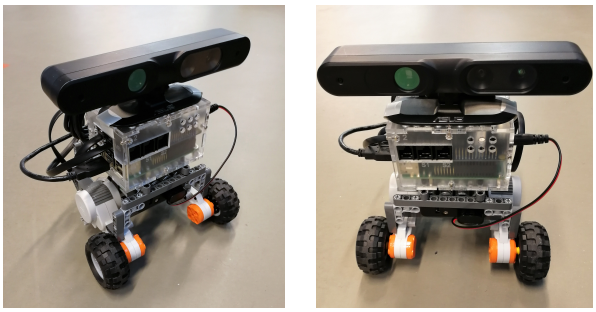


Figura 2.5: Plataforma robótica elaborada en el proyecto.

2.2. Software

Una vez seleccionados los distintos elementos que conforman el hardware del proyecto, es hora adecuar el software a las posibilidades que nos ofrece el hardware y a los requerimientos del trabajo. Como punto de partida se analizan los distintos sistemas operativos que se consideró utilizar. Posteriormente, se describen ROS, entorno de trabajo utilizado, OpenCV, biblioteca empleada para la detección de vehículos y los lenguajes de programación utilizados.

Sistema Operativo. A la hora de escoger un sistema operativo adecuado para este proyecto, se barajó entre dos distribuciones de GNU/Linux y por tanto, de código libre. Estos dos sistemas operativos son Raspbian y Ubuntu, los cuales tienen sus ventajas e inconvenientes. Por ejemplo, Raspbian tiene mayor apoyo por parte de los desarrolladores para su ejecución en Raspberry Pi, lo que supone un mayor rendimiento que Ubuntu en el mismo computador. Además, las bibliotecas de Dexter Industries para BrickPi están desarrolladas sobre Raspbian aunque admiten la compatibilidad con Ubuntu. Sin embargo, la diferencia más importante que repercutió en nuestra decisión fue que Raspbian soporta peor las versiones recientes de ROS, por ello desde ROS recomiendan el uso de Ubuntu, sistema operativo empleado en el proyecto. La versión utilizada en el proyecto es Ubuntu 16.04.

ROS. Como se ha mencionado anteriormente ROS es el acrónimo de *Robot Operating System* (Sistema operativo Robótico), un entorno de trabajo de código abierto empleado para desarrollar de forma simplificada proyectos complejos en plataformas robóticas. Este permite obtener una aplicación eficiente y robusta, siendo capaz de mantener la modularidad, la reutilización de código y la integración del hardware empleado.

En este proyecto se comenzó trabajando con la versión de ROS Indigo, que era hasta ese momento la más estable. Sin embargo, dicha versión estaba centrada en Ubuntu 14.04, por lo que finalmente se decidió actualizar a la versión recomendada para Ubuntu 16.04, ROS Kinetic, con el fin de establecer a su vez la compatibilidad con la versión de OpenCV utilizada para la detección de vehículos.

Como ya se mencionó en el capítulo de introducción, en la *wiki* de ROS hay una gran cantidad de tutoriales que hacen más fácil la adaptación a su uso. A continuación se exponen los aspectos más representativos del funcionamiento de ROS:

- El software de ROS está organizado en *packages*, que proporcionan una manera fácil de reutilizar el software. La descripción del *package* se establece en el *package manifest*, un archivo XML que recoge el nombre del *package*, la versión, autores y las dependencias con otros *packages*. Un *package* puede contener bibliotecas independientes, software a terceros, archivos de configuración, pero sobre todo contiene los *nodes*.

- Los *nodes* son procesos que realizan una tarea específica. Los *nodes* se comunican entre sí gracias al *node master*. La comunicación puede ser mediante *topics*, que permiten una comunicación unidireccional entre los *nodes*, mediante un sistema de suscripción y publicación a través de *messages*. Los *services* son una alternativa a los *topics*, debido a que en ocasiones el sistema unidireccional de comunicación no es el adecuado. Los *services* se basan en una solicitud y una respuesta a través de un par de *messages*.

  Generalmente en un proyecto se tienen varios *nodes*, por ejemplo, en este proyecto se tiene un *node* para la cámara, un *node* para la construcción del mapa, otro *node* para el control de los motores y así con todos los elementos. El esquema global de funcionamiento se detalla en el capítulo 3.

Ejemplificando el funcionamiento de ROS, comentar que en este proyecto es necesario un *node* para llevar a cabo la localización y el mapeo simultáneos. Una de las opciones de uso es el *node* perteneciente al *package* GMapping. Para su funcionamiento necesita dos tipos de información: la odometría y los escaneos de un láser. Para obtener esa información el *node* de GMapping se suscribe al *topic* */odom* y al *topic* */scan* respectivamente. Y a su vez, el *node* de GMapping publica el mapa construido en el *topic* */map*.

Figura 2.6: Esquema de alto nivel del software utilizado en el proyecto. El *node* de detección de vehículos (basado en OpenCV) está escrito en C++, al igual que el *node* de SLAM. Mientras que el *node* de control del movimiento está escrito en Python.

OpenCV. Para la realización de la aplicación de detección de vehículos se hace uso de la biblioteca de código abierto OpenCV, la cual es compatible con GNU/Linux, sistema operativo empleado en el proyecto. En este caso se ha hecho uso de la versión OpenCV 3.4.1 para llevar a cabo la detección de vehículos, lo que favoreció el uso de ROS Kinetic debido a su compatibilidad con las versiones de OpenCV 3, mientras que ROS Indigo presenta compatibilidad con OpenCV 2, dificultando el uso de las versiones recientes de OpenCV.

Lenguajes de Programación. En el proyecto se han utilizado dos lenguajes de programación: C++ y Python. La Figura 2.6 muestra de forma conceptual los lenguajes utilizados en el proyecto. Por ejemplo, la detección de vehículos se llevó a cabo con el software en C++ de OpenCV, mientras que para el control de los motores se utilizó la biblioteca para el módulo BrickPi3 de Dexter Industries, que está escrita en el lenguaje Python. Además de programar en Python el sistema de movimiento, también se ha programado la monitorización para visualizar el rendimiento del sistema. Por último, mencionar que los *packages* utilizados de ROS para realizar la tarea de SLAM o la teleoperación están programados en C++.

Capítulo 3

Arquitectura del sistema

En este capítulo se describe el proyecto desarrollado, el cual puede dividirse fundamentalmente en tres módulos. El primero controla el movimiento del robot, el segundo lleva a cabo la elaboración del mapa y el tercero se encarga de la detección de los vehículos. A pesar de la división realizada, los tres módulos se comunican entre sí para desarrollar las distintas tareas. En la Figura 3.1 se muestra, de forma conceptual, un diagrama de los principales bloques del sistema, así como las relaciones entre ellos. En verde se encuadra el módulo de desplazamiento, compuesto por el *node* de teleoperación y el *node* de movimiento, que se encargan del control del robot y de los cálculos de la odometría. El módulo de mapeado (encuadrado en rojo) además de la odometría, requiere de la información del escaneo para publicar el mapa, mientras que el módulo de detección, encuadrado en azul, publicará la posición 2D de los vehículos.

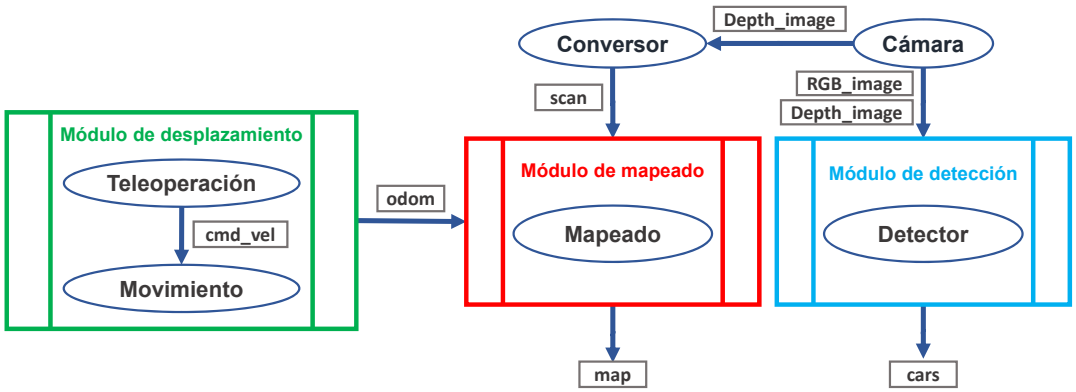


Figura 3.1: Esquema conceptual de los módulos del sistema. En elipses los *nodes* del sistema y en rectángulos los *topics* que emplean los *nodes* para comunicarse entre sí.

3.1. Módulo de desplazamiento

Movimiento. El *node* movimiento tiene una doble función, por una parte se encarga de enviar las consignas de velocidad al motor derecho e izquierdo y por otra parte desarrolla los cálculos de la odometría para estimar la localización del robot durante la navegación.

Como se muestra en la Figura 3.1, para establecer las **velocidades** de ambos motores este *node* se suscribe al *topic /cmd_vel* y de esta forma obtiene las referencias de velocidad lineal y velocidad angular que han sido publicadas por el *node* teleoperación. La comunicación con los motores se lleva a cabo por medio del módulo BrickPi3 que provee la función *set_motor_dps* para establecer la velocidad de ambos actuadores en grados por segundo.

Como el robot construido es de tracción diferencial (Figura 3.2), las velocidades de las ruedas se obtienen a partir de las velocidades lineal y angular del robot tal como se muestra en (3.1) donde v es la velocidad lineal, ω es la velocidad angular del robot, L es la distancia entre ejes de ambas ruedas y r es el radio de las ruedas.

$$\begin{bmatrix}\omega_d\\ \omega_i\end{bmatrix} = \begin{bmatrix}1/r & L/2r\\ 1/r & -L/2r\end{bmatrix}\begin{bmatrix}v\\ \omega\end{bmatrix} \Rightarrow \begin{cases}\omega_d = \frac{v}{r} + \frac{wL}{2r}\\ \omega_i = \frac{v}{r} - \frac{wL}{2r}\end{cases} \tag{3.1}$$

Por otra parte, este *node* también se encarga de realizar los cálculos de la **odometría**. Para ello, se emplean las velocidades lineal y angular reales del robot, obtenidas por medio de la función *get_motor_encoder* de la biblioteca de BrickPi3. Esta función permite obtener la posición del encoder de cada motor que sirven para calcular la velocidad de cada rueda. A partir de las velocidades de ambas ruedas, se obtienen las velocidades lineal y angular de la plataforma robótica como se muestra en (3.2). Estas velocidades permiten obtener la localización del robot, es decir, la posición y orientación en 2D.

$$\begin{bmatrix}v\\ \omega\end{bmatrix} = \begin{bmatrix}r/2 & r/2\\ L/2 & -L/r\end{bmatrix}\begin{bmatrix}\omega_d\\ \omega_i\end{bmatrix} \Rightarrow \begin{cases}v = \frac{r\omega_d}{2} + \frac{r\omega_i}{2}\\ \omega = \frac{r\omega_d}{L} - \frac{r\omega_i}{L}\end{cases} \tag{3.2}$$

Para obtener la localización del robot se calcula el desplazamiento producido en un intervalo temporal y de esta forma obtener las variaciones de posición y orientación producidas tal como se muestra en (3.3), donde la aproximación del incremento sobre el arco (ΔS) solo es válida para variaciones de ángulo ($\Delta\theta$) pequeñas o nulas:

$$\begin{cases}\Delta\theta = \omega\Delta t\\ \\ \Delta S \approx v\Delta t \Rightarrow \begin{cases}\Delta X = \Delta S\cos(\theta + \frac{\Delta\theta}{2})\\ \Delta Y = \Delta S\sin(\theta + \frac{\Delta\theta}{2})\end{cases}\end{cases} \tag{3.3}$$

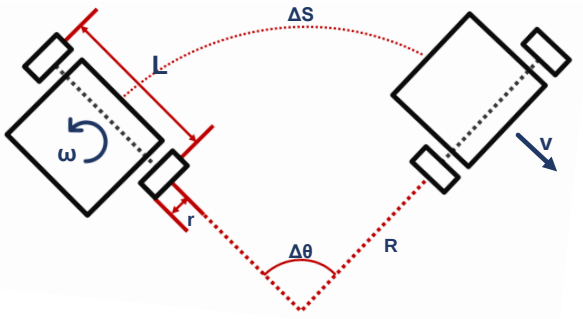


Figura 3.2: Modelo cinemático de un robot de tracción diferencial.

Los nuevos parámetros de posición y orientación son publicados en el *topic /odom* mediante un mensaje de tipo *nav_msgs/Odometry* para que pueda ser utilizado por el módulo de mapeado. La posición y orientación constituyen un mensaje tipo *geometry_msgs/Pose* compuesto a su vez por un mensaje *geometry_msgs/Point* (x, y, z) y un mensaje *geometry_msgs/Quaternion* que recoge la transformación de la orientación θ de ángulos de Euler a cuaternios.

Teleoperación. El *package* de teleoperación utilizado es *teleop_twist_keyboard* [13] y su principal funcionalidad es publicar la referencia de velocidad, tanto lineal como angular, para que el *node* de movimiento pueda ejecutar el desplazamiento de la plataforma robótica. El funcionamiento del *node* teleoperación consiste en leer si se ha pulsado una de las teclas predeterminadas que modifican la velocidad de referencia. Una posibilidad es que se desee un incremento o decremento de la velocidad lineal o angular, en ese caso las referencias de velocidades actuales son multiplicadas por un determinado factor de escala. Otra posibilidad es que se requiera un tipo movimiento distinto, los cuales pueden ser avance, retroceso, giro a izquierdas, giro a derechas... como se muestran en la Figura 3.3.

Para publicar la referencia de velocidad lineal y velocidad angular establecida, el *node* teleoperación utiliza un mensaje tipo *geometry_msgs/Twist* que se compone de dos vectores de tres componentes, uno para la referencia de velocidad lineal y otro para la referencia de velocidad angular. Este mensaje permite publicar las referencias de velocidad en el *topic /cmd_vel*, al cual se suscribe el *node* movimiento.

El hecho de teleoperar el movimiento de la plataforma robótica es una de las razones del uso de un computador como controlador remoto. Además de emplear el computador remoto como medio de teleoperación del movimiento, también sirve para la visualización gráfica del mapa construido (mediante la herramienta RViz) y la monitorización del rendimiento del sistema.

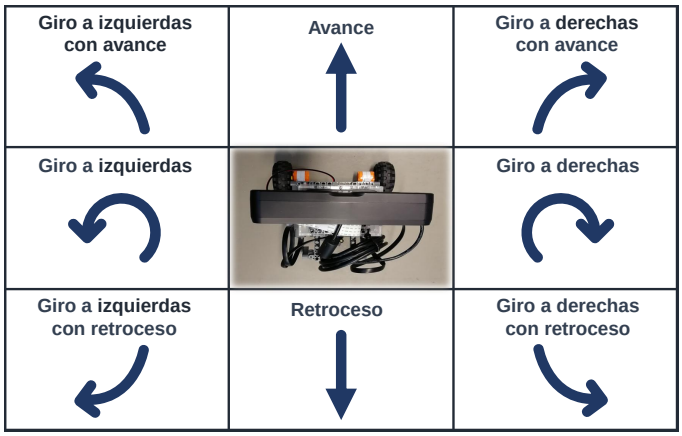


Figura 3.3: Diagrama de los posibles movimientos a realizar con la teleoperación.

3.2. Módulo de mapeado

En está sección se detalla el módulo de mapeado, constituido por los métodos de mapeado empleados para la construcción de un mapa 2D del entorno con localización del robot en el mismo (SLAM). Los dos métodos integrados y evaluados en este proyecto son GMapping [1] y Cartographer [3], que permiten la realización de la tarea de SLAM a partir del escaneo de un láser y de la información de la odometría. Como se comentó en la sección 1.4, Hector SLAM [2] proporciona generalmente peores resultados que estos dos métodos, por ello se descartó su uso.

Los dos métodos evaluados tienen en común el uso de un láser en la implementación del SLAM, sin embargo, como se mencionó en la sección 2.2 en este proyecto se hace uso de una cámara RGB-D. Para posibilitar el empleo de la cámara como láser se emplea el *package* de conversión de datos *depthimage_to_laserscan* [14]. Este *package* permite la conversión de la imagen de profundidad de la cámara a escaneo láser. El funcionamiento básico de una cámara RGB-D así como la conversión de la información se encuentran explicados en el anexo B.

GMapping. Como se ha mencionado anteriormente GMapping permite llevar a cabo SLAM en 2D. Para ello requiere información en forma de escaneo láser, que es obtenida a partir de la imagen de profundidad gracias al *node* conversor, que publica el escaneo en el *topic /scan*. Además el *node* de GMapping llamado *slam_gmapping* hace uso de la odometría para la localización del robot en el mapa creado, el cual se publica en el *topic /map* por medio de un mensaje *nav_msgs/OccupancyGrid*. Respecto a la configuración del funcionamiento de GMapping se ha estudiado la incidencia de diversos parámetros

como son la tasa de actualización del mapa, la compensación del error de la odometría, la configuración del filtro de partículas y la resolución del mapa.

En primer lugar, se ha estudiado tanto el intervalo de actualización temporal del mapa publicado a través de *map_update_interval* como los parámetros que determinan el desplazamiento necesario para llevar a cabo un nuevo escaneo, *linear_update* y *angular_update*. En segundo lugar, se ha analizado la compensación del error procedente de la odometría, tanto en traslación (*srr y srt*) como en rotación (*str y stt*). Para concluir se ha trabajado con mapas de distinta resolución modificando el parámetro *delta* y se ha estudiado el comportamiento del sistema ante una modificación del número de partículas del filtro (*particles*), ya que GMapping estima la localización del robot mediante un filtro de partículas *Rao-Blackwellized*.

Cartographer. La integración de Cartographer en ROS permite llevar a cabo SLAM en 2D y 3D. Debido a la limitación de cómputo por parte de Raspberry Pi, en este proyecto se ha trabajado con SLAM 2D.

Cartographer permite trabajar con odometría o sin ella, aunque con la finalidad de obtener mejores resultados en este proyecto se ha hecho uso de la misma mediante la suscripción del *node* de Cartographer al *topic /odom*. Por otra parte, Cartographer permite trabajar con varios sensores a la vez pero al igual que para GMapping el estudio realizado se ha particularizado para un láser horizontal. Para obtener la medida de escaneo el *node* de Cartographer se suscribe al *topic /scan* publicado por el *node* conversor. El mapa generado se publica en el *topic /map* a través de un mensaje *nav_msgs/OccupancyGrid*. En el caso de la configuración de Cartographer se ha estudiado la incidencia de parámetros relacionados con la optimización de la localización, la validez de emparejamiento y la resolución del mapa.

En primer lugar, para obtener un resultado fiable se ha estudiado tanto la confianza de la posición obtenida a través del SLAM (*local_slam_pose_rotation_weight* y *local_slam_pose_translation_weight*) como la confianza en la posición de la odometría (*odometry_rotation_weight* y *odometry_translation_weight*). Estos parámetros permiten configurar la influencia de las informaciones recibidas dando más importancia a aquellas en las que se confié más. Por otra parte, con objeto de obtener mejores resultados se ha estudiado la incidencia de parámetros relacionados con los emparejamientos, por ejemplo, la puntuación mínima necesaria para que un emparejamiento sea correcto (*min_score*) o la dificultad de un nuevo emparejamiento de desbancar al obtenido anteriormente. Al igual que para GMapping, en Cartographer se ha trabajado con distintas resoluciones de mapa por medio del parámetro *resolution* para establecer la comparativa entre ambos métodos como se muestra en el análisis de la sección 4.2.

3.3. Módulo de detección

Clasificador en cascada basado en Haar *features*. En esta última sección se describe el módulo relacionado con la detección de vehículos. Para la consecución de la tarea de detección de vehículos, en este proyecto se ha hecho uso del clasificador AdaBoost (*Adaptative Boosting*) basado en *boosting* [15], algoritmo que consiste en obtener un clasificador robusto como suma de clasificadores débiles. El término *Adaptative* hace referencia al enfoque de entrenamiento centrándose en los datos que fueron mal clasificados con el fin de corregirlos. Para la obtención de estos clasificadores débiles se realiza la extracción de las características más representativas de una imagen. En este caso se hace uso del descriptor Haar para la extracción de características, que estudia la variación de intensidad en las dos direcciones de la imagen. Para evaluar de manera eficiente esta variación de intensidad se hace uso de máscaras como las que muestra la Figura 3.4 que nos permiten calcular lo que se conoce como Haar *features*. Además se incluye el concepto de clasificador en cascada, para conseguir una evaluación más eficiente. Consiste en no evaluar siempre todas las características, sino hacerlo por etapas. Las primeras etapas del clasificador realizan un análisis general, lo que posibilita el descarte de las imágenes que con seguridad no son el objeto a detectar, mientras que las últimas etapas realizan un análisis más exhaustivo, es decir, con más características.

Entrenamiento e integración del clasificador Haar. Para adaptar e integrar dicho método en nuestro proyecto hemos particularizado la detección del clasificador para el reconocimiento de vehículos. Para ello se ha entrenado el clasificador con cientos de imágenes positivas (imágenes que contienen el objeto a detectar) y negativas (imágenes que no contienen el objeto a detectar). En la Figura 3.5, se muestran encuadrados en azul ejemplos de imágenes positivas empleadas en el entrenamiento y en rojo se encuentran encuadrados los ejemplos negativos. En segundo lugar, encuadradas en amarillo están las imágenes de validación, es decir, aquellas imágenes que se han empleado para comprobar los parámetros del clasificador. Por último, encuadradas en verde las imágenes de testeo, empleadas para obtener el rendimiento del clasificador.

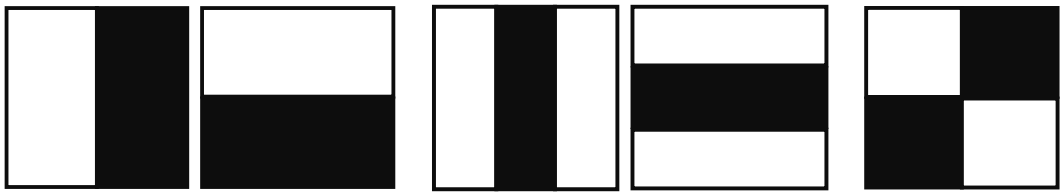


Figura 3.4: Ejemplos de máscaras utilizadas para las Haar *features*.

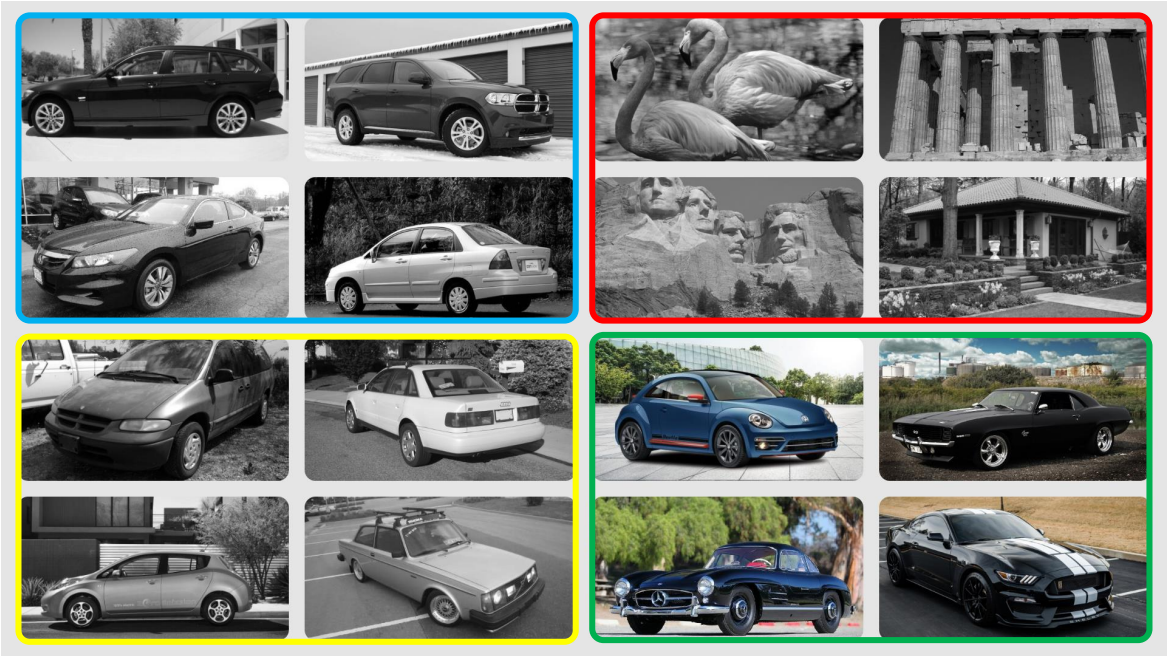


Figura 3.5: Ejemplos de imágenes utilizadas en el reconocimiento de vehículos.

En la sección 4.3 se muestran los resultados obtenidos para clasificadores con configuraciones distintas. En las distintas configuraciones se han modificado los parámetros *numStages* y *maxFalseAlarmRate* del clasificador con el objetivo de obtener el mejor rendimiento posible. Cuando se trabaja con un mayor número de etapas (*numStages*) se puede trabajar de forma progresiva; por ejemplo, aplicando 1, 5, 10, 25 y 50 características en las primeras cinco etapas del entrenamiento del clasificador lo que reduce el tiempo de cómputo en las etapas iniciales. Sin embargo, si se trabaja con un excesivo número de etapas se puede llegar a producir el sobreajuste de dicho parámetro, empeorando los resultados. Por su parte, el parámetro *maxFalseAlarmRate* se utiliza para definir la cantidad de características necesarias para que el clasificador tenga una buena tasa de acierto a la vez que descarta los negativos lo más rápido posible. Todos los clasificadores se han obtenido al entrenar con imágenes en grises y reducidas a 100 x 40 *píxeles*, con el fin de reducir el tiempo necesario para llevar a cabo el entrenamiento.

Para la integración del clasificador en ROS se ha modificado el *package People* [16] destinado en su origen a la detección de personas. Este *package* permite el uso del nodo detector que se suscribe a la información de la cámara, tanto de la imagen a color (RGB) como de la imagen de profundidad. La posición en 2D de los vehículos detectados se publica en el *topic* */cars*. Mediante la visualización gráfica en el computador remoto se muestra la posición del vehículo en el mapa del entorno llevado a cabo por el módulo anterior.

3.4. Integración

En esta sección se muestra el diagrama de la Figura 3.6 que representa el funcionamiento global del sistema, representando los *nodes* (elipses), los *topics* (rectángulos) y las relaciones entre ellos. El gráfico del funcionamiento del sistema se ha obtenido por medio del *package rqt_graph* [17]. En este caso el diagrama se ha obtenido con el uso de GMapping dedicado a la tarea de SLAM, aunque de forma similar se podría obtener un diagrama el uso de Cartographer en dicha tarea. En el diagrama se muestran las dependencias de los *nodes* para un correcto funcionamiento. Por ejemplo, se puede ver como una de las entradas del *node slam_gmapping* es */scan*, que es publicado por parte del node *depthimage_2_laserscan* que requiere la imagen de profundidad (*depth/image_raw*).

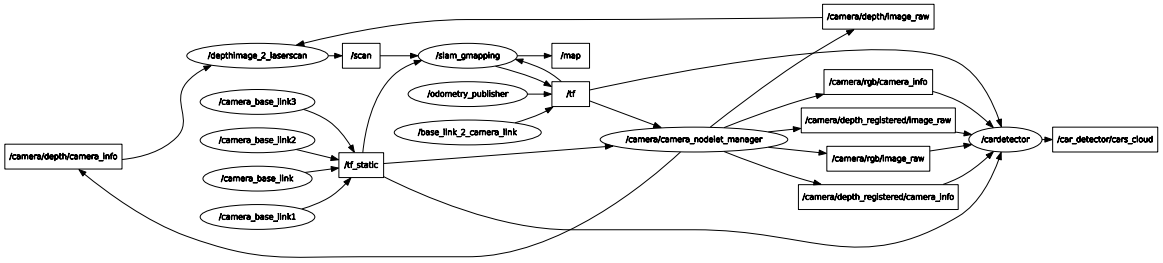


Figura 3.6: Gráfico del funcionamiento global del sistema, donde los *nodes* están representados con elipses y los *topics* están representados con rectángulos.

Capítulo 4

Evaluación y Experimentación

En este capítulo se redactan los resultados obtenidos en los distintos experimentos realizados para evaluar el comportamiento del sistema desarrollado, así como la correcta consecución de las tareas pretendidas.

El entorno de trabajo donde se han llevado a cabo las distintas pruebas ha sido el laboratorio del grupo de Robótica, Percepción y Tiempo Real del Instituto Universitario de Investigación en Ingeniería de Aragón. En dicho laboratorio se ha reproducido un entorno modelo en el que se han efectuado los experimentos con el fin de comparar los resultados bajo repetibilidad del escenario. El entorno modelo creado se muestra en la Figura 4.1.

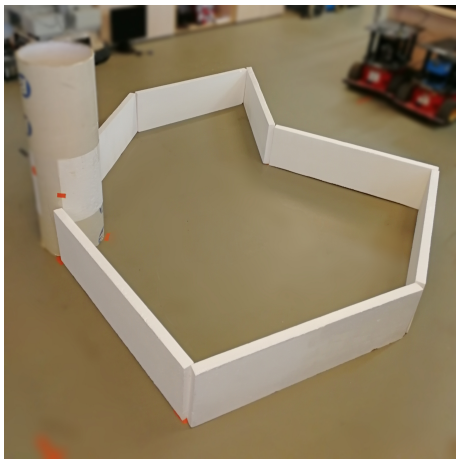


Figura 4.1: Mapa modelo recreado en el laboratorio.

Para desarrollar un análisis completo del proyecto se han analizado los distintos bloques implementados, extrayendo datos concluyentes de cada uno de ellos. En primer lugar, se exponen los resultados del módulo de desplazamiento, centrados en el análisis de los errores en la odometría del robot. En segundo lugar, se establece un análisis comparativo entre los métodos de SLAM estudiados experimentalmente, GMapping y Cartographer. En la búsqueda de un análisis completo en esta segunda parte de ensayos se ha evaluado tanto el rendimiento de la Raspberry Pi 3 como la precisión del mapa obtenido. Por último, se ha evaluado el módulo de detección de objetos analizando las tasas de acierto obtenidas ante diversas configuraciones.

4.1. Análisis de la odometría

Como parte del estudio del módulo de desplazamiento se analizan los datos procedentes del cálculo de la odometría. En el capítulo 3 se mencionó que la odometría es calculada y publicada en el *topic /odom* por el *node* movimiento y además dicho *topic* es una de las entradas del módulo de mapeado para así obtener una mejor solución, por ello es necesario evaluar el impacto que puede tener el error procedente de la odometría en el comportamiento del sistema. Para evaluar dicho error se analiza una trayectoria simple, que es representativa del error obtenible en desplazamientos lineales y angulares.

La trayectoria evaluada puede dividirse en tres tramos. El primer tramo consiste en un desplazamiento lineal de un metro en la dirección del eje X. En el segundo tramo se lleva a cabo un giro de 90° positivos, es decir, un giro a izquierdas. En el tercer tramo se efectúa un desplazamiento lineal de un metro en la dirección del eje Y.

En la Figura 4.2 se muestran los resultados obtenidos en dos ensayos al efectuar la trayectoria en forma de *"L"*. La línea azul continua y la línea verde continua son las representaciones de los datos obtenidos por medio del cálculo de la odometría. Esta líneas representan la trayectoria que la plataforma robótica cree haber efectuado. Por su parte, la línea roja discontinua es la representación de la trayectoria real que ha seguido el robot. Se observa que se produce error en la odometría, pero lo importante es concluir si dicho error repercute en el comportamiento del sistema.

En la Tabla 4.1 se recogen los datos obtenidos al concluir los distintos tramos de las trayectorias efectuadas. De los datos extraídos en el ensayo se concluye que el error obtenido en el desplazamiento lineal del primer tramo está dentro del rango de $\pm 0,1\,m$ en la dirección de avance, es decir, la dirección del eje X, mientras que en

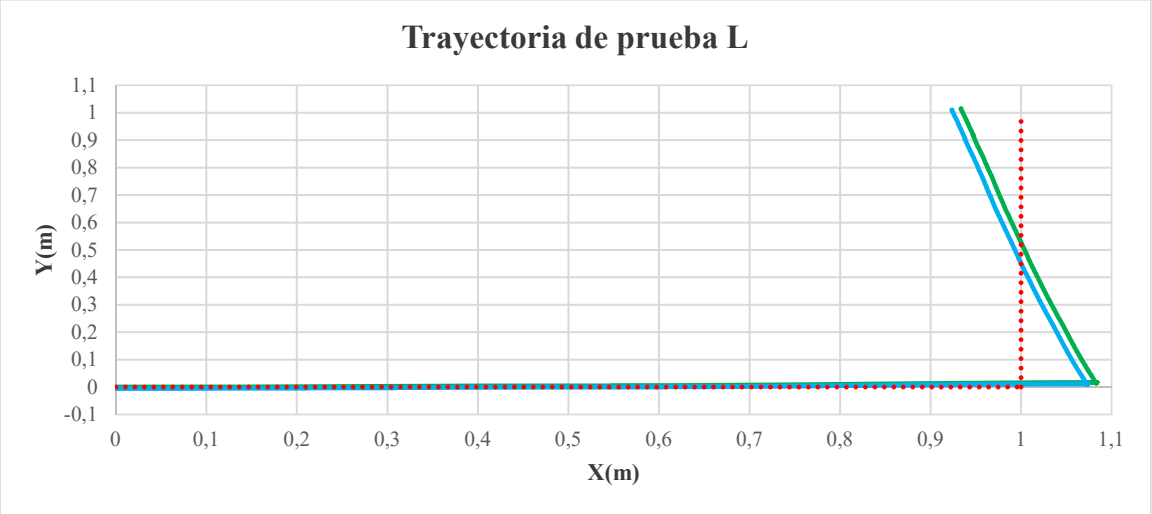


Figura 4.2: Representación de las trayectorias de prueba. Las líneas continuas verde (L_1) y azul (L_2) son las trayectorias obtenidas de los cálculos de la odometría y la línea roja discontinua es la trayectoria real.

la dirección del eje Y el error es despreciable al ser del orden de $\pm 0,02\,m$. Por tanto, tenemos un error aproximadamente del 10 % en desplazamientos lineales. Por su parte, al efectuarse el giro de $1,57\,rad$ (90°) los errores están en el rango de $\pm$0,16 rad, que corresponde también a un error del 10 %.

En los ensayos efectuados se ha estudiado la incidencia del error en la odometría en el comportamiento global del sistema, en particular, en la construcción del mapa dado que GMapping y Cartographer hacen uso de la odometría. En estos ensayos se presta especial atención a la influencia del error en la orientación del robot ya que influye directamente en el error de la posición del robot. Como se comentó en la sección 3.2, GMapping y Cartographer permiten corregir de una forma u otra dichos errores: GMapping actúa compensando el error, mientras que Cartographer permite establecer un grado de confianza en la odometría para evitar su uso cuando ésta sea imprecisa.

Trayectoria	Tramo	Localización real			Localización ideal			Error absoluto		
		x (m)	y (m)	θ (rad)	x' (m)	y' (m)	θ' (rad)	x -x'	y -y'	θ -θ'
L_1	1	1,0829	0,0157	0,0000	1,0000	0,0000	0,0000	0,0829	0,0157	0,0000
	2	1,0829	0,0157	1,7259	1,0000	0,0000	1,5700	0,0829	0,0157	0,1559
	3	0,9337	1,0157	1,7150	1,0000	1,0000	1,5700	-0,0663	0,0157	0,1450
L_2	1	1,0715	0,0153	0,0000	1,0000	0,0000	0,0000	0,0715	0,0153	0,0000
	2	1,0715	0,0153	1,7317	1,0000	0,0000	1,5700	0,0715	0,0153	0,1617
	3	0,9226	1,0149	1,7025	1,0000	1,0000	1,5700	-0,0774	0,0149	0,1325

Tabla 4.1: Error procedente de la odometría en las trayectorias de prueba

Los errores de odometría son frecuentes en las plataformas robóticas y sus orígenes son múltiples, por ejemplo, puede proceder de posibles desigualdades en el diámetro de las ruedas, la desalineación de las ruedas, el desplazamiento por suelos irregulares o el deslizamiento de las ruedas debido a derrapes. Particularizando para este proyecto, es probable que los errores puedan derivar de la construcción de un robot de bajo coste, donde el diámetro y la alineación de las ruedas no es precisa. El suelo sobre el que se han realizado las pruebas es antideslizante pero presenta irregularidades en su superficie, lo que puede ser origen de error en la odometría.

De este primer ensayo se concluye que los errores de la odometría son admisibles en este caso de estudio dado que GMapping y Cartographer permiten minimizar su incidencia. Pese a ser válido para el desarrollo de este proyecto, en un estudio futuro sería recomendable optimizar la odometría con objeto de obtener mejores resultados.

4.2. GMapping vs Cartographer

El segundo estudio de interés es el análisis comparativo entre dos integraciones de ROS para localización y mapeo simultáneos, GMapping y Cartographer. En primer lugar, se ha llevado a cabo el análisis del rendimiento de la plataforma robótica durante la ejecución de las tareas del sistema, en particular se ha evaluado el rendimiento de la Raspberry Pi 3 ya que el objetivo primario de este trabajo reside en comprobar las capacidades de un robot de bajo coste basado en este tipo de computadores. En segundo lugar, se efectúa el análisis de la calidad del mapa elaborado, en términos de precisión del mismo.

Temperatura. Uno de los parámetros estudiados en el rendimiento es la temperatura de la CPU, debido a que un elevado valor de esta puede repercutir en daños irreversibles sobre el computador. En la Figura 4.3 se muestra la representación gráfica de la evolución de la temperatura en la CPU durante la ejecución de las tareas pretendidas, donde la elaboración del mapa es efectuada por GMapping o Cartographer.

En la representación gráfica se observa que la temperatura en reposo de la CPU se encuentra en torno a $55°C$. Esta temperatura se debe a la integración de la Raspberry Pi 3 junto con el módulo BrickPi3 en el interior de una caja de metacrilato debido a la búsqueda de mayor robustez estructural, ya que fuera de la misma se experimentan temperaturas en reposo en torno a $35°C$.

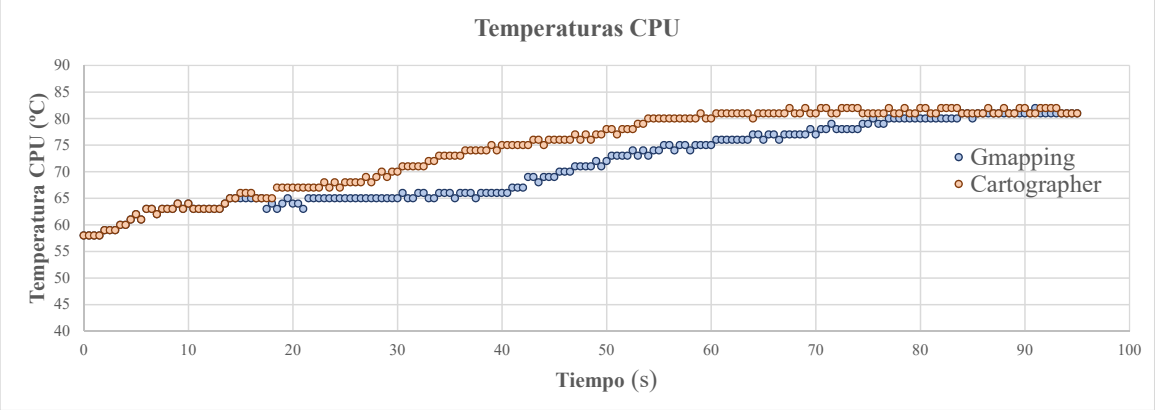


Figura 4.3: Representación de la evolución temporal de la temperatura en la CPU.

En la evolución de la temperatura se observa un incremento desde la ejecución de las tareas, momento que se produce aproximadamente a los cuatro segundos. Las diferencias apreciables son mínimas, aunque durante el uso de Cartographer se alcance antes la temperatura máxima experimentada, esto se produce tan solo unos segundos antes. Con ambos métodos se obtienen unas temperaturas máximas similares, en torno a los 80ºC. Dicha temperatura es el límite admisible recomendado por el fabricante, por ello sería recomendable disminuirla estableciendo una mejor refrigeración del sistema a la vez que se mantiene la robustez del mismo.

Aunque se experimentan temperaturas elevadas durante la ejecución de los ensayos se ha comprobado que esto no repercute en la frecuencia de la CPU, la cual se mantiene a su máximo valor de 1,2GHz durante los experimentos realizados.

Memoria. El segundo parámetro estudiado es el porcentaje de memoria RAM utilizada debido a la limitación que presenta en este aspecto el computador utilizado. En la Figura 4.4 se muestra la representación gráfica de la evolución de la memoria RAM en uso. Hay que tener en cuenta que se trata de una representación absoluta, es decir, tiene en cuenta todas las tareas en ejecución, sin embargo, la única diferencia entre los dos ensayos es el uso de GMapping o Cartographer, por tanto es una muestra directa de la diferencia entre ambos.

Al igual que para la temperatura, se observa en el gráfico (Figura 4.4) que aproximadamente a los cuatro segundos comienzan a ejecutarse las tareas y a los pocos segundos se alcanza el valor permanente de memoria RAM en uso. En el caso de Cartographer el porcentaje de RAM en uso está en torno al 40 %, mientras que para GMapping se encuentra en torno al 44 %.

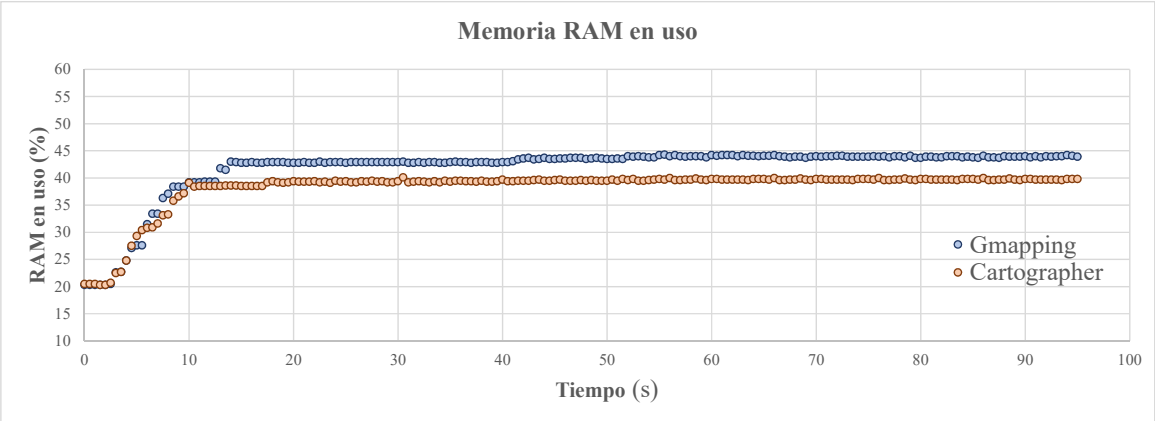


Figura 4.4: Representación de la evolución temporal de la memoria RAM en uso.

Para establecer un análisis más detallado de la incidencia sobre la memoria RAM se adjunta la Tabla 4.2, que recoge el porcentaje de uso desglosado. De esta forma se obtiene directamente la diferencia de requerimientos de memoria RAM de ambos métodos. El porcentaje de memoria RAM utilizada en la ejecución del SLAM por parte de Cartographer es del $2,3\,\%$, mientras que en el caso de GMapping es del $4,2\,\%$.

Programa/Comando	Gmapping	Cartographer
Roscore	3,0	3,0
Rosmaster	2,9	2,9
Rosout	0,8	0,8
Roslaunch	3,1	3,1
Nodelet	9,5	8,4
Transformaciones	4,5	4,5
Conversión de imagen a láser	1,6	1,6
Odometría y telemanipulación	9,0	9,0
SLAM	**4,2**	**2,3**
Otros	5,4	4,4

Tabla 4.2: Porcentajes desglosados de la memoria RAM en uso.

Tiempos. El último ensayo del rendimiento de la Raspberry Pi 3 se centra en el tiempo que la CPU se dedica a la gestión de código del usuario, mencionar que el resto del tiempo se dedica a la ejecución de código del sistema y a la gestión de las interrupciones. Al igual que sucedía con la representación gráfica de la memoria RAM en uso, la Figura 4.5 muestra el porcentaje de todo el código a nivel de usuario. Este código incluye los cálculos de la odometría, la realización de las tareas de SLAM, la detección de vehículos, etc. Sin embargo, debido a que la única diferencia se encuentra en el método de SLAM utilizado, los resultados muestran la diferencia directa entre Cartographer y GMapping.

De los análisis del rendimiento se extrae que los requerimientos de GMapping y Cartographer son similares. Sin embargo, Cartographer tiene un comportamiento menos exigente sobre el computador utilizado, ya que el sistema está en torno un 10 % menos del tiempo ejecutando código del usuario. Una vez concluido estos ensayos es hora de evaluar la calidad del mapa obtenido con ambos métodos.

Precisión de los mapas. Para evaluar la calidad en la elaboración del mapa se han medido en el entorno de experimentación varias distancias de referencia (distancias de las paredes y distancias entre las esquinas anotadas en la Figura 4.6). En los ensayos efectuados se ha trabajado con una resolución de $0,01\,m/píxel$. La Figura 4.7 muestra un ejemplo de los resultados obtenidos con ambos métodos, donde a priori se observa que el mapa obtenido con GMapping tiene una calidad superior al obtenido durante el uso de Cartographer. Sin embargo, para comprobar cuál de los dos métodos elabora un mapa que se acerca más al modelo real se procede a medir las distancias para compararlas con las medidas reales del entorno de trabajo y así evaluar el error producido.

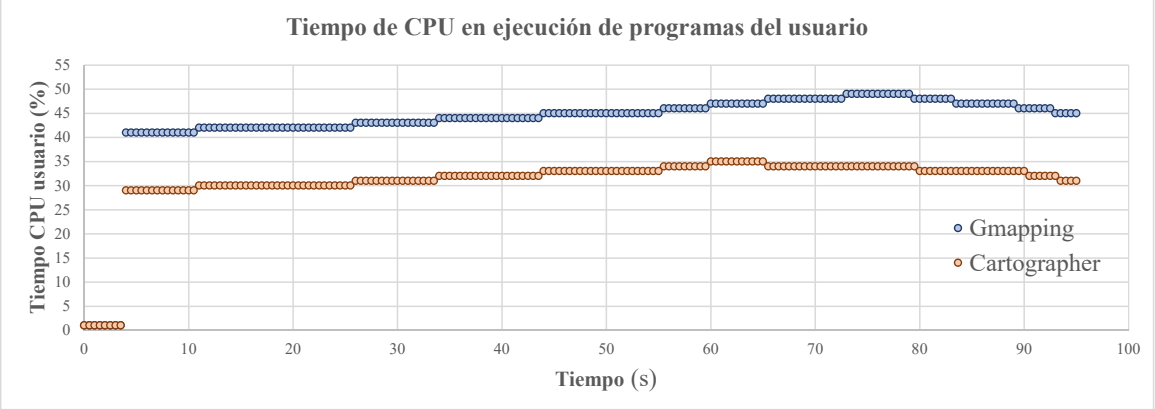


Figura 4.5: Porcentaje de tiempo de la CPU dedicado a la gestión de código de usuario.

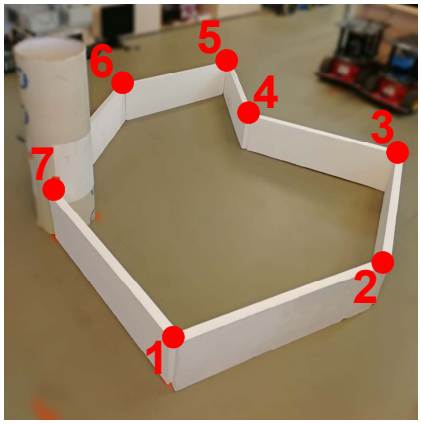


Figura 4.6: Entorno de experimentación con anotación de esquinas, tomadas como referencia en las medidas interiores.

La Tabla 4.3 muestra las medidas reales así como las distancias medias obtenidas para cada uno de los métodos en los diferentes ensayos. De los datos extraídos se certifica una mejor solución bajo el uso de GMapping, que muestra una menor desviación respecto las medidas reales para la mayoría de las distancias medidas.

De este ensayo se concluye que la precisión en la en la elaboración del mapa es superior con el uso de GMapping, pero la capacidad de realización de SLAM en 3D por parte de Cartographer es motivo de consideración en el desarrollo de futuras aplicaciones. Por otra parte, de los ensayos se concluye que es más perjudicial el error de odometría en GMapping. Esto se debe a que cuando falla el emparejamiento de los escaneos, GMapping se fía de la odometría y hace uso de ella.

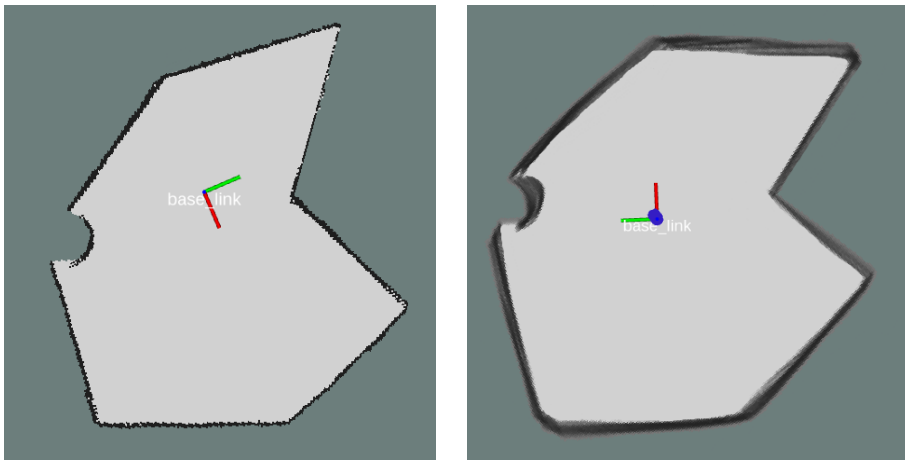


Figura 4.7: Desde la izquierda: mapa de GMapping y de Cartographer a 0,01$m/píxel$

Tramo	Distancia real (cm)	**Gmapping** Media de distancias (cm)	**Cartographer** Media de distancias (cm)	**Gmapping** Error (cm)	**Cartographer** Error (cm)
1--3	177	174	195	3	-18
1--4	170	161	177	9	-7
1--5	255	248	264	7	-9
1--6	210	216	218	-6	-8
2--4	140	137	123	3	17
2--6	220	221	209	-1	11
4--6	97	105	105	-8	-8
Paredes	100	104	99	-4	1

Tabla 4.3: Error obtenido en la construcción del mapa con GMapping y Cartographer.

4.3. Análisis de la detección de vehículos

Otro objetivo del proyecto era la inclusión de anotación semántica en el mapa construido. Para ello se lleva a cabo la detección de vehículos, que es posible mediante el uso de un clasificador lo suficientemente ligero como para ejecutarse en la plataforma desarrollada. En esta sección se analizan los resultados obtenidos en el entrenamiento de la detección de vehículos. Como se comentó en la sección 3.3, en este proyecto se hace uso de un clasificador Haar [8], para cuyo entrenamiento es necesario un conjunto de muestras positivas y negativas, debido a ello se hace uso de *datasets* que contienen los ejemplos necesarios [18] [19]. En el entrenamiento del clasificador se ha incidido en el estudio de la influencia de dos parámetros. El primero de ellos es el número de etapas de las cuales consta el entrenamiento, *numStages*. El segundo define la cantidad de características necesarias para que el clasificador tenga un comportamiento adecuado, *maxFalseAlarmRate*.

La Tabla 4.4 muestra los resultados obtenidos con diversas configuraciones cuando se comprueba su eficacia sobre el conjunto de imágenes de testeo, es decir, imágenes de vehículos que no forman parte del conjunto de imágenes de entrenamiento. El mejor resultado obtenido (encuadrado en rojo) corresponde al clasificador finalmente utilizado en este trabajo para realizar la detección de los vehículo.

Para extraer resultados concluyentes del clasificador entrenado se analiza tanto el *recall*, que representa la relación de clasificaciones positivas reales respecto todas las muestras positivas, como la precisión, que representa la relación de clasificaciones

Clasificador	StageNum	MaxFalse	Detecciones Correctas	Detecciones Falsas	Recall	Precision	F-measure (%)
Cars_v1	5	0,5	162/200	56	0,81	0,74	77
Cars_v2	10	0,5	168/200	32	0,84	0,84	84
Cars_v3	15	0,5	184/200	37	0,92	0,83	87
Cars_v4	20	0,5	154/200	34	0,77	0,82	79
Cars_v5	15	0,7	176/200	19	0,88	0,90	89
Cars_v6	15	0,9	162/200	24	0,81	0,87	84

Tabla 4.4: Evaluación de los clasificadores obtenidos para distintas configuraciones.

positivas reales respecto todas las clasificaciones dadas como positivas. Como este trabajo tiene un enfoque global, buscamos el equilibrio de ambos parámetros expresado por medio de la métrica *F-measure*, que se calcula como detalla la ecuación 4.1.

$$F - measure = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{4.1}$$

Los cuatro primeros clasificadores dan muestra de la incidencia del número de etapas, donde la mejor tasa de aciertos se obtiene para 15 etapas donde la $F = 87\,\%$, destacar que pese a establecer una configuración con 20 etapas el resultado obtenido es peor lo que supone un sobreajuste de dicho parámetro.

Los siguientes clasificadores dan muestra de la influencia de *maxFalseAlarmRate*, para ello se comparan *Cars_v3*, *Cars_v5* y *Cars_v6*. El clasificador *Cars_v5* con $maxFalseAlarmRate = 0,7$ proporciona un *recall* inferior que el clasificador con $maxFalseAlarmRate = 0,5$. Sin embargo, la precisión es mayor dado que se producen 18 detecciones falsas menos (muestras falsas clasificadas como positivas), por ello el clasificador *Cars_v5* con $F = 89\,\%$ resulta ser el mejor clasificador obtenido y por ello el empleado finalmente en la detección de vehículos del proyecto. Por otra parte, se observa de nuevo en el clasificador *Cars_v6* que se experimenta un sobreajuste, esta vez del parámetro *maxFalseAlarmRate*.

En conclusión, el clasificador obtenido con un *recall* del 88 % y una precisión del 90 % tiene un comportamiento válido en este caso de estudio, donde la detección de vehículos no deja de ser un objetivo complementario al verdadero objetivo de estudio de las posibilidades de una plataforma robótica de bajo coste.

4.4. Aplicación final

Al obtener unos resultados satisfactorios en los ensayos efectuados es posible llevar a cabo una aplicación final que agrupe las tareas desarrolladas en el proyecto. La aplicación final consiste en construir el mapa del entorno de experimentación a la vez que el robot se localiza en dicho mapa y además incluir la anotación semántica de los vehículos reconocidos en el entorno. El resultado final de la aplicación desarrollada se muestra en la Figura 4.8.

Para la correcta consecución de la aplicación final es necesaria la integración simultánea de las distintas tareas comentadas en la memoria. En primer lugar, es necesario disponer de un robot dotado de movilidad, lo que se consigue mediante el control de los actuadores tal como se ha comentado en la sección 3.1. Con un robot dotado de movilidad es posible llevar a cabo el cálculo de la odometría, que es fundamental para mejorar los resultados en la realización de la tarea de SLAM, de acuerdo con lo mencionado en la sección 3.2. Por otra parte, además de la odometría, el módulo dedicado a realizar la tarea de SLAM necesita la información del escaneo del entorno. Para obtener este escaneo es necesario convertir los datos de profundidad procedentes de la cámara RGB-D a escaneos láser.

Por último, es necesario llevar a cabo la detección de los vehículos, para lo cuál ha sido necesario el entrenamiento previo del clasificador utilizado. El *node* encargado de la detección permite obtener la posición en 2D del vehículo detectado respecto del robot, la cual es publicada en el *topic /cars*. La visualización gráfica en RViz permite mostrar el mapa obtenido así como la posición de los vehículos, los cuales se representan mediante círculos tal como muestra la Figura 4.8.

En la Figura 4.9 se muestra un ejemplo de la trayectoria seguida durante la ejecución de la aplicación propuesta para construir un mapa del entorno. Al llevar a cabo el escaneo con una cámara RGB-D no es posible obtener la distancia de obstáculos que se encuentran a menos de $0,30\,cm$ del robot, debido al rango válido de medida del sensor. Por ello, se evita la aproximación excesiva a los límites del entorno de experimentación, concentrando la trayectoria que sigue el robot en el centro de dicho entorno. También es importante que la trayectoria tenga una combinación de traslaciones y rotaciones, para obtener un mejor modelo del entorno. Además dichos desplazamientos deben realizarse a una velocidad adecuada, evitando movimientos bruscos que repercuten de forma negativa en el resultado obtenido.

La Figura 4.10 muestra distintos pasos intermedios del proceso de realización de mapa anotado. Las líneas rojas representan la sección del entorno escaneada por el

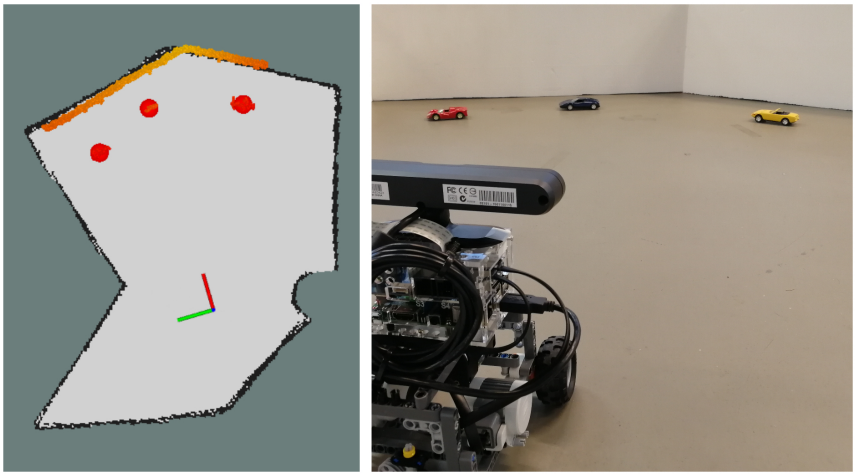


Figura 4.8: Construcción del mapa con anotación semántica de vehículos detectados.

sensor. En el paso 3 de dicha figura se percibe que la construcción final del mapa presenta alguna inconsistencia. Esto se soluciona en el paso siguiente, donde se lleva a cabo el cierre de bucle del algoritmo de SLAM, al haber sido detectada una zona del entorno ya visitada. Por último, se activa el *node* de detección de vehículos, lo que permite obtener la posición de los vehículos en el mapa tal como muestra el paso 5.

De este modo, se concluyen los resultados experimentales del proyecto, donde se ha certificado el correcto funcionamiento de la plataforma robótica elaborada basada en Raspberry Pi 3 al ejecutar de manera simultánea múltiples tareas en ROS. El cumplimiento del objetivo establecido en el inicio del proyecto, abre camino al estudio de otras aplicaciones sobre la plataforma robótica elaborada, tal como se comenta en el capítulo 5.

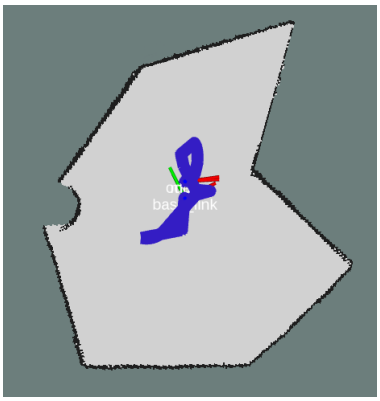


Figura 4.9: Construcción del mapa incluyendo la trayectoria realizada.

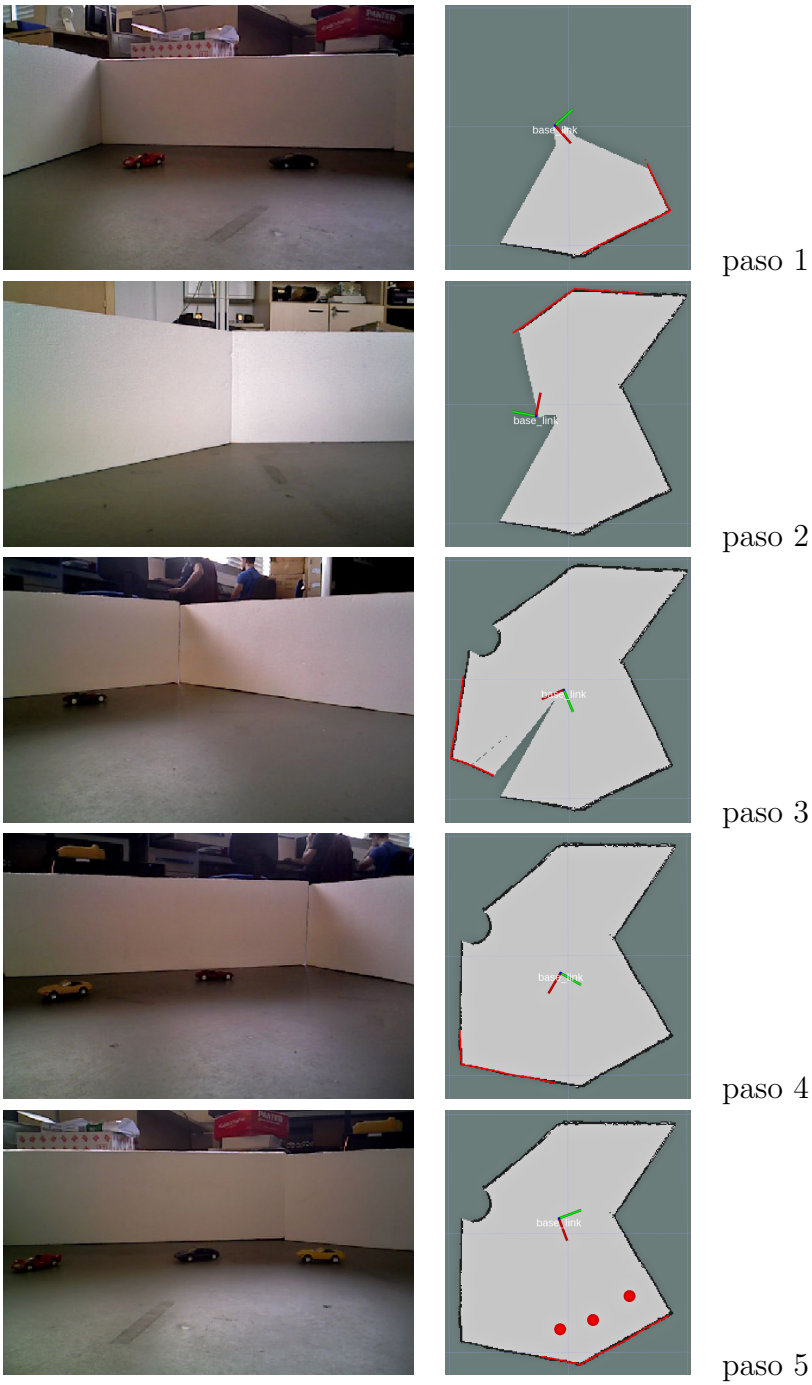


Figura 4.10: Proceso completo de construcción del mapa anotado. Imagen capturada por el robot en distintos instantes de tiempo (izq) y mapa construido hasta ese momento (dcha). El paso 5 presenta el resultado final, donde esta activo el sistema de detección/anotación de coches.

Capítulo 5

Conclusiones

Conclusiones. En este trabajo se han integrado en un robot de bajo coste tareas ejecutadas por robots de alta gama en entornos reales. Estas tareas consisten en la construcción de un mapa del entorno y la localización del robot en el mismo, además de la anotación semántica en el mapa construido de vehículos detectados. El principal objetivo era comprobar las posibilidades de ejecución de dichas tareas en un entorno similar al que se puede darse durante la realización de asignaturas de la Universidad de Zaragoza y el resultado obtenido es satisfactorio.

Respecto a las conclusiones de los métodos empleados y los ensayos realizados, se ha constatado el conocimiento previo del uso de la odometría como información complementaria. Esto se debe a que el error procedente de la odometría impide que sea la única información a utilizar, especialmente en aplicaciones que se demoran en el tiempo puesto que el error crece proporcionalmente. Por ello es necesario complementarla con la información extraída por medio de los sensores correspondientes.

El rendimiento de la Raspberry Pi 3 bajo el uso de los dos métodos estudiados experimentalmente es adecuado. Destacar la evaluación de las temperaturas experimentadas que se sitúan en el límite permitido lo que evidencia la posible mejora de refrigeración de la plataforma robótica. Respecto a la calidad del mapa construido por ambos métodos se ha podido concluir que GMapping provee una precisión mayor bajo la misma configuración de resolución. Sin embargo, también se ha comprobado que Cartographer presenta un comportamiento menos exigente, lo que permite llevar a cabo la tarea de SLAM en menor tiempo, dado que el emparejamiento de los escaneos es más rápido.

En cuanto a la detección de objetos se ha conseguido llevar a cabo la ejecución de dicho objetivo con unos resultados válidos en cuanto a rapidez, ya que se logra llevar a cabo la detección de vehículos a la vez que se construye el mapa del entorno. Además la efectividad de acierto en los resultados es positiva en un entorno sencillo como el evaluado. Sin embargo, en el futuro habría que integrar sistemas de aprendizaje automático más avanzados ya que esto es solo una prueba de realización de este tipo de tareas.

Trabajos futuros. Este proyecto abre el camino a futuros estudios sobre plataformas robóticas de bajo coste, sobre las cuáles podría integrarse una reconstrucción del entorno 3D, por ejemplo, con el uso de Cartographer. Esto podría ser posible si se consigue optimizar el sistema para evitar el calentamiento excesivo de la plataforma robótica. Otra opción de estudio complementario sería estudiar como integrar en el sistema algoritmos de reconocimiento más recientes y potentes, como las redes neuronales, para la detección de objetos. Por último, se podría estudiar la integración de sensores adicionales, por ejemplo, el LIDAR comentado en la sección 2.1 o una unidad de medición inercial (IMU) dado que Cartographer permite su uso como una entrada de información complementaria.

Conclusiones personales. Por último, a nivel de conclusión personal mencionar que durante el desarrollo de este trabajo he aplicado y profundizado en técnicas aprendidas en diversas asignaturas, como robótica o visión por computador. Además he adquirido conocimiento acerca de ROS, un entorno frecuente en proyectos robóticos. Para concluir, este trabajo me ha servido para darme cuenta de los problemas que surgen desde las etapas iniciales en proyectos reales, problemas a los cuales me he tenido que enfrentar con el fin de obtener los resultados deseados dentro del plazo establecido.

Bibliografía

[1] Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard. Improved Techniques for Grid Mapping with Rao-Blackwellized Particle Filters. *IEEE Transactions on Robotics*, 23:34–46, 2007.

[2] Stefan Kohlbrecher, Johaness Meyer, Oskar von Stryk, and Uwe Klingauf. A Flexible and Scalable SLAM System with Full 3D Motion Estimation. In *Proc. IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*, 2011.

[3] Wolfgang Hess, Damon Kohler, Holger Rapp, and Daniel Andor. Real-Time Loop Closure in 2D LIDAR SLAM. In *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, pages 1271–1278, 2016.

[4] Hugh Durrant-Whyte and Tim Bailey. Simultaneous localization and mapping: part I. *IEEE Robotics Automation Magazine*, 13(2):99–110, 2006.

[5] Maha Al-Nesf, Hamza Aagela, and Violeta Holmes. An Asus Xtion Pro Based Indoor Mapping Using a Raspberry Pi With Turtlebot Robot. *23rd International Conference on Automation and Computing (ICAC)*, pages 1–5, 2017.

[6] B. M. F. da Silva, R. S. Xavier, T. P. do Nascimento, and L. M. G. Gonsalves. Experimental evaluation of ROS compatible SLAM algorithms for RGB-D sensors. In *Proc. Latin American Robotics Symposium (LARS) and 2017 Brazilian Symposium on Robotics (SBR)*, pages 1–6, 2017.

[7] K. Kamarudin, S. M. Mamduh, A. Y. M. Shakaff, S. M. Saad, A. Zakaria, A. H. Abdullah, and L. M. Kamarudin. Method to convert Kinect's 3D depth data to a 2D map for indoor SLAM. In *Proc. IEEE 9th International Colloquium on Signal Processing and its Applications*, pages 247–251, 2013.

[8] Paul Viola and Michael Jones. Rapid object detection using boosted cascade of simple features. In *Proc. IEEE Conference on Computer Vision and Pattern Recognition*, 2001.

[9] Viren Pereira, Vandyk Amsdem Fernandes, and Junieta Sequeira. Low cost object sorting robotic arm using Raspberry Pi. In *Proc. Global Humanitarian Technology Conference-South Asia Satellite (GHTC-SAS)*, 2014.

[10] Karel Horak and Ludek Zalud. Image Processing on Raspberry Pi for Mobile Robotics. *International Journal of Signal Processing Systems*, 4(2):1–5, 2016.

[11] S. S. A. Abbas, P. O. Jayaprakash, M. Anitha, and X. V. Jaini. Crowd detection and management using cascade classifier on ARMv8 and OpenCV-Python. In *2017 International Conference on Innovations in Information, Embedded and Communication Systems (ICIIECS)*, pages 1–6, 2017.

[12] Hussein Haggag, Mohammed Hossny, Despina Filippidis, Douglas Creighton, Saeid Nahavandi, and Vinod Puri. Measuring depth accuracy in RGBD cameras. In *Proc. Signal Processing and Communication Systems (ICSPCS), 7th International Conference*, 2013.

[13] Graylin Trevor Jay. Generic Keyboard Teleop for ROS. `http://wiki.ros.org/teleop_twist_keyboard`.

[14] Chad Rockey. Converts a depth image to a laser scan for use with navigation and localization. `http://wiki.ros.org/depthimage_to_laserscan`.

[15] Yoav Freund and Robert E. Schapire. A Short Introduction to Boosting. In *Proc. Sixteenth International Joint Conference on Artificial Intelligence*, pages 1401–1406, 1999.

[16] Caroline Pantofaru. Algorithms related to detecting and tracking people using various robot sensors. `http://wiki.ros.org/people`.

[17] Dirk Thomas. Graphical user interface plugin for visualizing the ROS computation graph. `http://wiki.ros.org/rqt_graph`.

[18] Shivani Agarwal, Aatif Awan, and Dan Roth. UIUC Image Database for Car Detection. `http://cogcomp.org/Data/Car/`.

[19] Jonathan Krause, Michael Stark, Jia Deng, and Li Fei-Fei. 3D Object Representations for Fine-Grained Categorization. In *4th International IEEE Workshop on 3D Representation and Recognition (3dRR-13)*, Sydney, Australia, 2013.

Anexos

Anexo A

Manual de uso

En este manual se detallan las instrucciones de instalación del software necesario para llevar a cabo el proyecto desarrollado. También, se explican los pasos a seguir para conectar la plataforma robótica (en adelante, Robot) al ordenador de control remoto (en adelante, PC). Por último, se comentan las instrucciones para llevar a cabo las distintas tareas del proyecto de forma conjunta en una misma aplicación. El objetivo de este manual es permitir el uso de los resultados de este proyecto como punto de partida en futuros estudios de índole similar.

A.1. Instalación del software

En esta sección se describen los pasos necesarios para la instalación del software utilizado en el proyecto. Los comandos de instalación descritos a continuación son válidos indistintamente del computador utilizado, con excepción de la instalación de Cartographer para Raspberry Pi 3 (Robot) como se explica en la presente sección. En este caso los comandos de instalación están destinados a un uso de ROS Kinetic, en el caso de utilizar otra versión de ROS modificar Kinetic por la versión correspondiente. De acuerdo con la instalación requerida ase abre un terminal y se ejecuta:

1. **Instalación de ROS:**
   sudo apt-get install ros-kinetic-desktop-full
2. **Instalación de controlador de cámara RGB-D:**
   sudo apt-get install ros-kinetic-openni2-camera ros-kinetic-openni2-launch

3. **Instalación de Hector SLAM:**
   sudo apt-get install ros-kinetic-hector-slam

4. **Instalación de GMapping:**
   sudo apt-get install ros-kinetic-openslam-gmapping

5. **Instalación de Cartographer:**
   Para llevar a cabo la instalación de Cartographer en Raspberry Pi 3 es necesario realizar un *swap file*, es decir, un archivo de memoria en el disco duro complementario a la memoria RAM, para ello se introduce:

   - dd if=/dev/zero of=/swapfile bs=1M count=2G
   - chown root:root /swapfile → Se elige el propietario
   - chmod 0600 /swapfile → Se elige que solo *root* pueda leer y escribir
   - mkswap /swapfile → Se elige el *swap file* creado como memoria *swap*
   - swapon /swapfile → Activación del *swap file*
   - swapoff /swapfile → Desactivación cuando ya no sea necesario

   Una vez se tiene el *swap file* se procede a la instalación:

   - sudo apt-get update
   - sudo apt-get install -y python-wstool python-rosdep ninja-build
   - mkdir cartographer_ws
   - cd cartographer_ws
   - wstool init src
   - wstool merge -t src https://raw.githubusercontent.com/googlecartographer/cartographer_ros/master/cartographer_ros.rosinstall
   - wstool update -t src
   - src/cartographer/scripts/install_proto3.sh
   - rosdep update
   - rosdep install --from-paths src --ignore-src --rosdistro=kinetic -y
   - catkin_make_isolated --install --use-ninja
   - source install_isolated/setup.bash

6. **Instalación de la biblioteca de Dexter Industries:**
   sudo curl https://raw.githubusercontent.com/DexterInd/Raspbian_For_Robots/master/upd_script/fetch_brickpi3.sh | bash

7. **Instalación de OpenCV y el *package People*:**
   Para la instalación de OpenCV ejecutar en consola las siguientes líneas:

   - sudo apt-get install cmake
   - sudo apt-get install python-devel numpy
   - sudo apt-get install gcc gcc-c++
   - sudo apt-get install git
   - git clone https://github.com/opencv/opencv.git
   - mkdir build
   - cd build
   - cmake ../
   - make -j 1
   - sudo make install

   Para la instalación del *package People* ejecutar los siguientes comandos:

   - mkdir detector_ws
   - cd detector_ws
   - mkdir src
   - src
   - git clone https://github.com/wg-perception/people
   - cd detector_ws
   - catkin_make
   - source ./devel/setup.bash

8. **Instalación de rqt:**
   sudo apt-get install ros-kinetic-rqt ros-kinetic-rqt-common-plugins

A.2. Comunicación del PC con el Robot

Para establecer la comunicación entre el computador de control remoto (PC) y la plataforma robótica (Robot) es necesario seguir los pasos descritos a continuación:

1. Conectar el PC y el Robot a la misma red, en el caso del Robot hacerlo mediante conexión inalámbrica.

2. Abrir un terminal en el PC y el Robot donde ejecutar *ifconfig* para visualizar las direcciones IP de ambos sistemas.

3. En un terminal ejecutar *sudo vim /etc/hosts* para ingresar las direcciones IP obtenidas junto con el nombre de usuario del computador correspondiente.

De esta forma las dos máquinas se conocen entre sí por nombre de modo que podemos realizar la conexión entre ambas por medio del protocolo SSH.

A.3. Lanzamiento de la aplicación

En esta sección se describen los pasos a seguir para la ejecución de las tareas desarrolladas en el proyecto. Las siguientes instrucciones requieren de la configuración previa de los diversos métodos puestos en práctica, así como del entrenamiento del clasificador empleado en la detección de objetos. Mencionar que la ejecución de instrucciones en el Robot se hace a través del PC por medio de la conexión SSH descrita anteriormente. El proceso para la ejecución de la aplicación es el siguiente:

1. Abrir diversos terminales del Robot en el PC mediante *ssh pi@jelen* (nombre de usuario del Robot) en los que llevar a cabo los siguientes pasos:

   - Iniciar estableciendo el maestro y diversos parámetros de ROS por medio de *roscore*.
   - Ejecutar el controlador de la cámara a través de *roslaunch openni2_launch openni2.launch*.
   - Iniciar el nodo *movement* comentado en la presente memoria por medio de *python movement.py*.
   - La teleoperación del Robot se lleva a cabo gracias a *rosrun teleop_twist_keyboard teleop_twist_keyboard.py*.
   - Ejecutar el *node* de SLAM a través de *roslaunch raspi_2D_gm.launch* en la ruta */home/pi/gmap_ws/* en el caso de GMapping. En el caso de utilizar Cartographer ejecutar *roslaunch raspi_2D.launch* en la ruta */home/pi/goocar_ws/src/cartographer_ros/launch/*.
   - Lanzar el detector de vehículos por medio de *roslaunch car_detector.rgbd.launch* en la ruta */home/pi/bridge_ws/src/cars/car_detector/launch/*.

2. Abrir un terminal en el PC en el que llevar a cabo:

   - Abrir la interfaz de visualización gráfica de RViz para obtener el mapa generado y los objetos detectados. El siguiente comando abre la interfaz con la configuración utilizada a través de *rosrun rviz rviz -d raspi_2D.yaml*.
   - Como posibilidad complementaria se puede visualizar el esquema de nodos y *topics* del sistema por medio de *rosrun rqt_graph rqt_graph*.

Anexo B

Cámara RGB-D

Como se ha comentado en la presente memoria, en este proyecto se hace uso de un sensor de profundidad. En concreto, se hace uso de una cámara RGB-D (Figura B.1), que combina un sensor infrarrojos para la obtención de profundidad y un sensor RGB para obtener la imagen a color. Para el correcto funcionamiento de estos sensores es necesaria una calibración previa, que permite obtener los parámetros intrínsecos de la cámara. Como en este proyecto se hace uso del *package Openni2* no se ha llevado a cabo la calibración de la cámara, ya que provee modelos de cámara predeterminados listos para usar con una precisión suficiente para el desarrollo de este proyecto. En el caso de requerimiento de mejora de la precisión es aconsejable realizar una calibración exhaustiva.

En la Figura B.2 se muestran las imágenes obtenidas con la cámara, tanto de profundidad como de color. De esta forma se observa que los píxeles más oscuros corresponden a objetos más cercanos a la cámara y los píxeles blancos representan datos inválidos ya sea porque están fuera del rango de la cámara (8 metros) o existan problemas de reflejos, sombras y transparencias.

Figura B.1: Cámara RGB-D utilizada: Asus Xtion PRO Live.

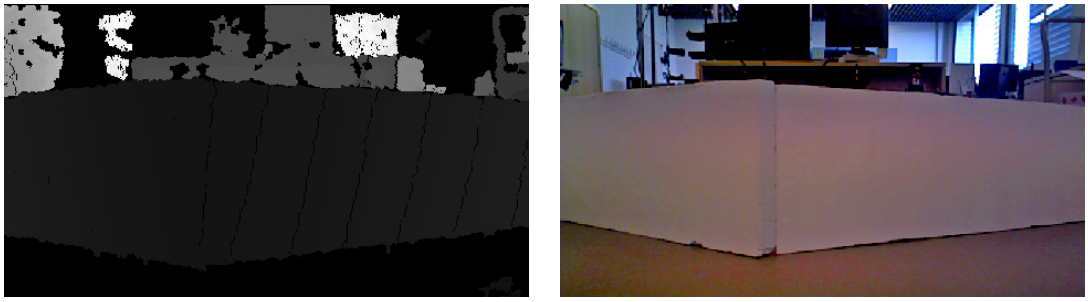


Figura B.2: Desde la izquierda: imagen de profundidad y color captadas por la cámara.

Sin embargo, los *packages* de ROS empleados en el proyecto para realizar la localización y el mapeo simultáneos en 2D requieren que sus datos de entrada sean del tipo *laserscan*. Por tanto, para poder emplear la cámara RGB-D como un LIDAR es necesario convertir la información captada por el sensor al formato adecuado. Para ello, se ha hecho uso del *package depthimage_to_laserscan*. Con el uso de este *package* se consiguen los datos en el formato adecuado para un correcto funcionamiento. A continuación, se expone a grandes rasgos como se realiza la conversión de los datos:

Para comenzar, mencionar que fundamentalmente existen dos métodos de conversión. El primero consiste en realizar una lectura completa del plano obtenido, recorriendo las n filas de la imagen y las m columnas. El método de conversión se basa en el hecho de que para evitar las colisiones hay que tener en cuenta los objetos más cercanos. De este modo la medida láser obtenida sera el valor mínimo de z de cada columna de la imagen de profundidad (B.1), donde cada píxel es un vector que contiene la profundidad. Y al suponer que no existen inclinaciones lateral y longitudinal de la cámara z es la distancia a la que se encuentra el obstáculo. Estas suposiciones son razonables en proyectos robóticos centrados en la navegación en entornos interiores.

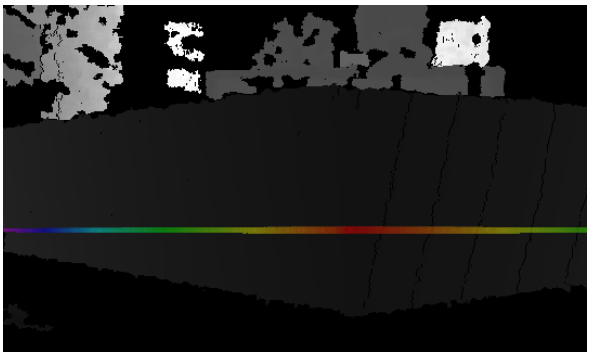


Figura B.3: Imagen de profundidad convertida a escaneo láser, donde los tonos rojos simbolizan obstáculos cercanos y tonos azules obstáculos lejanos.

$$z_i = \ min(z_{0,i}, z_{1,i}, ..., z_{n,i}) \tag{B.1}$$

El segundo método consiste en realizar un corte de la imagen a una determinada altura, generalmente la altura del centro óptico de la cámara. De esta forma se obtiene la profundidad de los píxeles de una línea horizontal. Cada método tiene sus ventajas e inconvenientes. El primer método permite comprobar obstáculos a diferentes alturas lo cual es interesante en plataformas con desplazamiento vertical. Sin embargo, se puede llegar a reconocer como obstáculo un elemento por el cual una plataforma terrestre pasaría por debajo y no le afectaría. Debido a esto, en el proyecto se hace uso del segundo método, que consiste en efectuar un corte tal como se muestra en la Figura B.3. Hay que tener en cuenta se puede llegar a detectar como obstáculo el suelo, por ello hay que establecer la cámara a una determinada altura y limitar su rango de alcance.