Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías Industriales

Detección de señales y planificación de trayectorias en un robot móvil

Autor: Juan Manuel León Muñoz
Tutor: José Ramón Domínguez Frejo
Tutor Externo: Ramón Andrés García Rodríguez

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

Sevilla, 2021

Proyecto Fin de Carrera
Ingeniería de las Tecnologías Industriales

Detección de señales y planificación de trayectorias en un robot móvil

Autor:
Juan Manuel León Muñoz

Tutor:
José Ramón Domínguez Frejo
Tutor Externo:
Ramón Andrés García Rodríguez

Dpto. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2021

Proyecto Fin de Carrera: Detección de señales y planificación de trayectorias en un robot móvil

Autor:	Juan Manuel León Muñoz
Tutor:	José Ramón Domínguez Frejo
Tutor Externo:	Ramón Andrés García Rodríguez

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

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2021

El Secretario del Tribunal

A mis padres Manuel y Rosario

A mis hermanos Curro y Jaime

A mis eternos compañeros Paco y Oscar

A mis educadores y maestros

A Carolina

Agradecimientos

Sirvan estas líneas para agradecer a todos aquellos que me han ayudado a hacer posible terminar esta gran carrera.

En primer lugar, como no podía ser de otra forma agradezco a mis padres Manuel y Rosario todo el apoyo, la motivación, la confianza y la sabiduría que durante estos años no han dejado de confiarme ni un momento. A mi hermano Curro gran artífice de este logro, gracias por marcar el camino como buen hermano mayor que siempre has sido. A mi hermano Jaime por hacer más livianas las duras tardes de estudio con tu simpatía y tu música.

También es de agradecer el incondicional soporte que ha supuesto una persona tan especial como Carolina durante estos años, solo ha tenido palabras buenas y motivadoras hacia mí, forzándome siempre a dar el máximo y seguir peleando por lograr mis metas.

Imposible no acordarme ahora de mis compañeros de carrera Paco y Oscar, siempre fueron los que me soportaron en los peores días. Entre los 3 hemos formado un equipo en el que siempre confiábamos ante cualquier adversidad. Nunca dejamos de creer en nosotros mismos y siempre estuvimos ahí los unos para los otros, sin importar el momento o las circunstancias. Gracias de veras porque sin vosotros todo esto nunca hubiera sido posible.

Gracias a la Universidad de Sevilla y en especial al Departamento de Automática por prestarme el material necesario para desarrollar el proyecto sin el cual nada de esto hubiera sido posible.

Por último, me gustaría agradecer la labor realizada por tantos y tantos profesores de la escuela que durante este tiempo me han transmitido conocimientos e ilusión a partes iguales, me siento eternamente en deuda por ayudarme a ser la persona que soy ahora mismo, consciente del mundo que me rodea y un poco más conocedor de cómo se desarrolla todo en esta vida.

Juan Manuel León Muñoz

Alumno de la Escuela Técnica Superior de Ingeniería de la Universidad de Sevilla

Sevilla, 2021

Resumen

El proyecto ha sido realizado con el Rosbot 2.0, robot móvil desarrollado por la empresa Husarion que funciona a través de la programación mediante ROS (Robot Operating System). El dispositivo fue escogido debido a que el departamento cuenta con varios ejemplares recién adquiridos y considero una gran oportunidad el poder familiarizarme mejor con el mundo de la robótica en particular y la programación en general.

El objetivo de este robot es educacional, cuenta con múltiples opciones de desarrollo, tiene motores independientes en cada una de las ruedas, un sensor LIDAR, una cámara RGBD además de varios sensores de infrarrojos. Todo ello se incluye para ofrecer la posibilidad de tener una primera toma de contacto con elementos que se usan de manera real en el mundo de la robótica actual.

Además, el desarrollo del proyecto es interesante por el creciente uso de robots móviles en la industria actual. El hecho de poder planificar su trayectoria a distancia, mapear entornos de gran superficie, y obtener tanto imagen como video en tiempo real, nos abre un amplio margen de posibilidades. Gracias a todo este interés cada vez es más frecuente la investigación en estos temas y el desarrollo de los mismos.

En mi trabajo me he centrado en el uso de la cámara RGBD junto con el sensor LIDAR. La idea principal es la implementación de un sistema detección de señales de tráfico que reaccione ante las mismas. Unido al desarrollo e implementación del mapeado de entornos mediante el LIDAR. Para posteriormente planificar trayectorias específicas en los mismos.

Para ello se realizará una presentación del entorno de trabajo, así como una descripción del robot. Seguidamente se presentará la estructura de funcionamiento de ROS. A continuación, se desarrollarán los algoritmos encargados de la detección de señales fundamentada en el nodo “Find_Object”. Para seguir, explicaremos cómo es posible la realización del mapeado de entornos basado en el nodo “Gmapping”. Antes de acabar, será necesario presentar el funcionamiento de la planificación de trayectorias que trabaja a través del nodo “move_base”. Este se encarga de dirigir comandos a la base con el fin de diseñar su trayectoria.

El proyecto se fundamentará en la puesta en marcha de todos estos nodos dentro de sus algoritmos particulares, para posteriormente ser capaces de hacer que funcionen todos en consonancia con el fin de alcanzar el objetivo. Con esto se llevará a cabo una familiarización con el entorno de trabajo de ROS. Así como un aprendizaje en el funcionamiento del robot tanto interna como externamente.

Para finalizar, se mostrarán los resultados obtenidos con todas estas herramientas para demostrar el correcto funcionamiento de nuestros algoritmos. Seguidamente se ofrecerá una visión general del proyecto mediante las conclusiones a las que he llegado después de realizarlo. Como elementos extra se añadirán dos anexos, un manual de usuario y una librería con todos los códigos empleados explicados línea a línea.

Abstract

The project has been realized with the RosBot 2.0, a mobile robot developed by Husarion, it works by the programing with ROS (Robot Operating System). The device was chosen because the department bought some of these ones recently. I think is a great opportunity to understand the world of robotic and continue learning about programing.

The main target of the robot is the education, it has so many options for developing, it has engines in every wheel, a LIDAR sensor, an RGBD camera, and some infrared sensors. All these options are included to have the opportunity of working for the first time with devices used in the actual robotic.

In addition, the development of the project is interesting due to the increasing use of mobile robots in today's industry. The fact of being able to plan it trajectory in the distance, mapping big area environments and obtain image and video in real time, offer us a big range of possibilities. Due to all this interest research on these topics their development is becoming more frequent.

On my work I have used overall the RGBD camera and the LIDAR sensor. The main target is to create a program able to detect traffic signals and to react at this moment to the detection, joined to the mapping of certain environments and the path planning to reach a destination.

First of all, I will present the work environment, as well as a description of the robot. After this, the ROS operating structure Will be presented, also the algorithms responsable of the signal's detection base don the node "Find_Object". To continue, I will explain how it's posible to carry out the environment's mapping base don the "gmapping" node. Before finishing, it will be necessary to present the operation of the trajectory planning that Works trough the "move_base" node. This one send commands to the base in order to design their trajectory.

The project will be based on the start-up of all these nodes within their algorithms, to later be able to make them all work in accordance with the purpose of achieving the objective. With this, a familiarization with the ROS work environment will be carried out. As well as learning in the operation with the robot both internally and externally.

Finally, the results obtained with all these tools will be shown to demonstrate the correct operation of our algorithms. Next, an overview of the project will be offered through the conclusions I have reached after carrying it out. As extra elements, two annexes will be added, a user manual and a library with all the codes used explained line by line.

Índice

ÍNDICE DE FIGURAS

Índice de videos

Notación

c.t.p.	En casi todos los puntos
c.q.d.	Como queríamos demostrar
e.o.c.	En cualquier otro caso
e	número e
:	Tal que
L	Length
W	Width
H	Height
V	Voltios
L (x,y, σ)	Valor de cada punto en la matriz de la figura
D (x,y, σ)	Valor de la diferencia gaussiana
X	Valor en el eje de abcisas
Y	Valor en el eje de ordenadas
σ	Valor relativo a la orientación del punto
H (x, σ)	Matriz Hessiana
Lxx= (x, σ)	Producto de convolución de segundo orden derivativo

Glosario

ROS	Robot Operating System
LIDAR	Laser Imaging Detection and Ranging
CV	Computer Vision
SIFT	Scale Invariant Feature Transformation
SURF	Speed Up Robust Feature
ORB	Oriented FAST and Rotated BRIEF
FAST	Features Accelerated Scaner Test
BRIEF	Binary Robust Independent Elementary Features
PC	Personal Computer
RGBD	Red Green Blue and distance
SLAM	Simultaneous Location and mapping
SVM	Support Vector Machine
GPS	Global Positioning System
SBC	Asus Tinker Board
SSH	Secure Shell
DWA	Dynamic Window Approach

1 INTRODUCCIÓN

Desde el inicio de la robótica siempre se ha buscado simplificar el papel del trabajo del hombre, se ha tratado de emplear máquinas en las operaciones más costosas para así evitar el cansancio físico del ser humano. En la actualidad dichas tareas de las antiguas fábricas en las que se necesitaba de un potente grupo de personas para realizar la labor encomendada, ya son realizadas por máquinas robotizadas en su mayoría.

Es por ello por lo que la investigación y el desarrollo de la robótica en la actualidad se centra en facilitar la vida del hombre en lo cotidiano, de un tiempo a esta parte se emplea esta potente herramienta para convertir nuestras vidas en algo más sencillo, automatizar tareas, liberarnos de preocupaciones y permitirnos emplear nuestro valioso tiempo en proyectos y objetivos de una mayor importancia.

Es en este contexto dónde cobran importancia elementos como la detección de señales o la planificación de trayectorias, dos herramientas que permiten un desarrollo de manera autónoma de una máquina en tiempo real, capaz de reaccionar ante estímulos y responder ante cambios. Las aplicaciones que estos temas tienen en la vida actual son múltiples, nos encontramos hoy en día con numerosos proyectos de conducción autónoma o control del tráfico que constantemente investigan sobre estos ámbitos para desarrollar una mejor función en un futuro muy cercano.

1.1 Contextualización del proyecto

El trabajo realizado tiene como objetivo el conocimiento de las posibilidades que ofrece el Rosbot 2.0. El hecho de la adquisición de varios ejemplares por parte del departamento los insta a conocer hasta donde se puede llegar trabajando con estos dispositivos desde el punto de vista de un alumno más. Su posterior aplicación, puede ser sin duda la adaptación de unas prácticas de laboratorio para una divulgación realista y en primera persona de lo que significa la robótica.

En primer lugar. El Rosbot 2.0 ofrece numerosas posibilidades para trabajar con él, como por ejemplo usarlo desde nuestro propio portátil. En él solo debemos tener instalado el sistema operativo Ubuntu junto con el programa ROS (Robot Operating System), esto nos permite conectarnos al robot y controlarlo de manera telemática.

En segundo lugar, el dispositivo cuenta con un procesador y un sistema operativo integrado por lo que puede conectarse a un monitor, teclado y ratón y trabajar directamente con él sin necesidad de intermediarios. Esta es la opción que más se desarrolla en el trabajo, ofrece la mayor cantidad de facilidades posibles y beneficia el avance en el proyecto.

Además, un modelo similar (RosBot 2.0 Pro) va a ser usado en el contexto del proyecto OCONTSOLAR para la toma de medidas de radiación solar, este modelo cuenta con un mejor procesador y mejores prestaciones en los sensores que incluye. Esto lo hace perfecto para el desarrollo de su función en su nuevo trabajo, donde se le dará un uso profesional y contribuirá a un importante proyecto.

El proyecto busca desarrollar nuevos métodos de control usando sensores móviles incorporados en drones y en los propios RosBot 2.0, esto se realiza de cara a la optimización de plantas solares usando estimaciones y predicciones de la radiación espacial. Además, los resultados podrán ser reutilizados en sistemas de control de tráfico mantenimiento de edificios energéticos y agricultura.

Por último, contamos con la posibilidad de utilizar un simulador integrado dentro del propio programa ROS llamado Gazebo. Esta es la opción que menos desarrollaremos ya que al tener el robot físicamente no tiene mayor importancia la simulación de proyectos.

1.2 Objetivo del Trabajo

El objetivo de este proyecto se fundamenta en los siguientes elementos:

- Detección y el mapeo de un entorno determinado por parte del Robot a través de su sensor LIDAR.
- Configuración y diseño de dicho mapa mediante el programa Rviz incluido dentro del paquete ROS.
- Planificación de una trayectoria específica para ser recorrida por el robot.
- Implementación en el sistema de la detección de varias señales de tráfico para que el dispositivo sea capaz de reconocerlas y posteriormente actuar en consecuencia cuando las detecte.

El trabajo previo del que se parte es el incluido dentro de los propios tutoriales de la página web oficial del producto (Husarion) así como la web de soporte de ROS donde también ofrecen información en la que me baso para el desarrollo de este trabajo.

Para cumplir con el objetivo debemos ser capaces de manejar con soltura el programa ROS y aprender a programar y manipular el robot con facilidad. En primer lugar, se trabajará en profundidad el aprendizaje y la posterior detección de imágenes. Seguidamente, se añadirá el mapeo de entornos y el diseño de trayectorias a recorrer.

Para todo ello deberemos sumergirnos en el programa ROS y familiarizarnos con su funcionamiento, su manera de estructurarse, de funcionar y trabajar para así poder finalmente desarrollar con el menor de los problemas el objetivo marcado al principio del trabajo.

1.3 Procedimiento a seguir

Con el único objetivo de realizar el trabajo previamente explicado, se debe seguir una metodología que va a pasar a resumirse a continuación, siendo aclarada esta en detalle posteriormente, los pasos que debemos recorrer son los siguientes:

- Aprendizaje básico sobre el robot y sus funciones (método de carga, funcionamiento de botones, indicaciones de leds…).
- Instalación de Ubuntu en una partición de la memoria del PC.
- Instalación de ROS en nuestro portátil.
- Aprendizaje básico sobre el funcionamiento del programa ROS mediante su wiki.
- Obtención de códigos básicos para el funcionamiento del Robot a través de Husarion.
- Conexión entre PC y robot además de probar el control de este mediante el teclado del PC.
- Aprendizaje, implementación y detección de imágenes (señales de tráfico) por parte del robot a través de su cámara RGBD
- Detección y mapeo de un determinado entorno por parte del robot a través de su sensor LIDAR.
- Diseño y planificación de trayectoria a seguir por parte de nuestro dispositivo.

Por tanto, la memoria del proyecto constará de varias partes, la primera incluirá el resumen e introducción de nuestro trabajo en el capítulo 1, seguidamente se hará una breve exposición de las nociones de la detección de señales y el seguimiento de trayectorias a lo largo de la historia en el 2º capítulo. Posteriormente se analizará cada uno de los pasos del procedimiento explicado con anterioridad.

Empezaremos analizando en profundidad el robot que vamos a utilizar en el capítulo 3, para ello se hará un análisis del hardware en el apartado 3.1 y el software en el apartado 3.2. Explicando brevemente los distintos sensores, motores y cámaras con los que cuenta. Para entender correctamente la situación de partida desde la que comenzamos se desarrollará el punto 3.3.

Posteriormente se explicará el funcionamiento de ROS en el 4º capítulo. Partiendo desde su instalación, requisitos previos y primeros pasos en el apartado 4.1. Para seguir analizando sus distintas capas y su método de trabajo en el subcapítulo 4.2. Además, en este mismo apartado se analizarán las herramientas utilizadas en el entorno de ROS.

A continuación, se comentarán los pasos a seguir para poder cumplir con los objetivos explicados, empezando con los distintos comandos para la configuración de nuestro espacio de trabajo, para seguir analizando los archivos ejecutables y documentos launchs necesarios en cada una de nuestras temáticas.

Se dedicará el quinto capítulo a la detección de objetos, centrándose este en el aprendizaje en el subcapítulo 5.2 y la posterior detección de imágenes que en nuestro caso básicamente serán señales de tráfico dentro del apartado 5.4. Seguidamente trataré de analizar en el mismo las librerías utilizadas, las funciones tomadas de OpenCV, además de aquellas de creación propia.

Tras esto se desarrollará la temática del mapeo de un entorno y la posterior planificación de trayectorias en el capítulo nº6, haciendo hincapié en los sensores y dispositivos utilizados para la misma, además de explicar cada uno de los comandos, programas y nodos que intervienen en el mismo. Se separará el mapeo de entornos de la planificación de trayectorias en los subcapítulos 6.3 y 6.4 respectivamente.

Para continuar, realizaremos un análisis de los resultados obtenidos en nuestro trabajo dentro del capítulo nº7, comentando si se cumplen los objetivos marcados al principio. Además de subrayar las posibilidades y aplicaciones que nuestro trabajo puede llegar a conseguir en un futuro si se continúa desarrollando el mismo. Se diferenciarán los resultados de visión de aquellos de mapeado y planificación de trayectorias en los apartados 7.1 y 7.2.

Para terminar, se realizarán unas conclusiones en el capítulo 8 donde puedan valorarse las limitaciones del proyecto además de las posibles mejoras futuras y su aplicación en la realidad. Se incluirán los errores que se han apreciado en el desarrollo de nuestro trabajo.

Dentro del anexo B se incluirán los códigos empleados para cada una de las funciones previamente explicadas y se tratará de comentar todo aquello que pueda ser de difícil entendimiento sin aclaraciones. Además, se desarrollará una guía de procesos necesarios para la realización del proyecto, conocido como manual de usuario en el anexo A.

2 NOCIONES BÁSICAS SOBRE VISIÓN ARTIFICIAL MAPEADO Y PLANIFICACIÓN DE TRAYECTORIA

A continuación, se realizará una breve introducción sobre las temáticas principales del proyecto, además de un análisis de ejemplos previos y un posterior uso de estas en la actualidad. Nos centraremos en la visión artificial, más concretamente en la detección de objetos dentro de videos en tiempo real. Posteriormente se comentará la planificación de trayectorias dentro de un entorno previamente reconocido a través de un mapeado.

2.1 Detección de Objetos

La visión artificial es una disciplina técnica que consiste en el análisis y la comprensión de imágenes del mundo real, para la posterior extracción de información alfanumérica sobre las mismas que pueda ser implementada y tratada desde un ordenador. Su funcionamiento se basa en el mismo método empleado por la visión humana y trata de adaptarla al mundo virtual.

La detección de objetos es la parte de la visión artificial que se centra en la detección de elementos particulares dentro de una imagen o un video como se muestra en la Figura 2-1. La detección cuenta con dos secciones bien diferenciadas, la primera de ellas es la extracción de características de los propios objetos que conforman una imagen. Posteriormente la segunda parte se centra en la búsqueda de los objetos tratando de analizar coincidencias con las características previamente extraídas.

Para la detección de características se utilizan numerosos descriptores de imágenes, algunos de ellos son elaborados modelos matemáticos basados en matrices, histogramas que analizan la luz, el color o incluso la distancia al objeto. Además, se emplean técnicas más avanzadas como el HOG que se basa en los histogramas, pero los clasifica mediante gradientes de características explícitas.

Para la clasificación de dichas características y su posterior reconocimiento, se utilizan técnicas de aprendizaje máquina. Como por ejemplo, la regresión logística que trata de predecir el resultado a obtener a través de predictores implementados en su sistema, o algunos más desarrollados como el Support Vector Machine (SVM) que se fundamenta en el mismo principio, pero a través de una serie de algoritmos es capaz de aprender de manera automática.

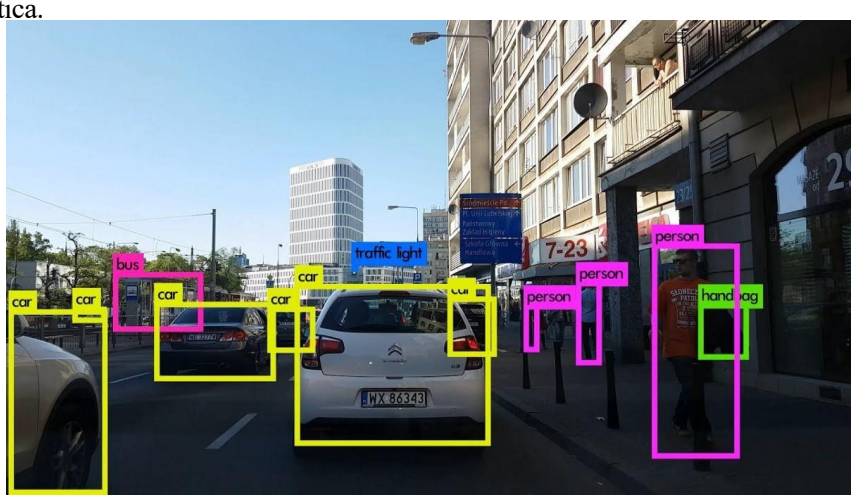


Figura 2-1 ejemplo de visión artificial (Fuente: [2])

Particularmente, en el entorno de ROS la visión artificial será desarrollada en su mayor parte por la función Find_object_2d, esta puede utilizar 3 algoritmos de implementación diferentes de la mano de la biblioteca de códigos de OpenCV, todo ello contribuirá a que una cámara cualquiera sea capaz de detectar un objeto y publicarlo dentro de un tópico de ROS.

Básicamente nuestra función se basa en dos pilares. Por un lado, la detección de puntos clave, aquellos capaces de captar los elementos más representativos de la imagen para poder ser comparados luego. Por otro lado, tenemos los descriptores. Estos se encargan de definir la orientación de la imagen, la rotación, la escala en la que se encuentra y su distancia al objetivo.

A continuación, se desarrollarán en profundidad los 3 algoritmos básicos que se utilizan en la visión artificial:

- Scale Invariant Feature Transformation (SIFT): desarrollada por Lowe en 2004 [**3**] se utiliza para extraer características relevantes de las imágenes en cuanto a la cantidad de información de su entorno. Como la existencia de bordes o texturas. La mayoría de los algoritmos de visión se basan en la detección de esquinas. Estos son invariantes ante las rotaciones, es decir, que si la imagen es sometida a giros pueden encontrarse los mismos puntos. Sin embargo, no funcionan de igual manera ante el escalado de imágenes. Este problema se soluciona con este nuevo algoritmo que se mantiene invariante ante la escala. Su desarrollo se fundamenta en el filtro de la diferencia gaussiana. Este funciona de la siguiente manera:

$$D(x, y, \sigma) = L(x, y, k_i\sigma) - L(x, y, k_j\sigma)$$

  En esta fórmula, se trata de buscar puntos de L (matriz de cada imagen) en los que no varía la rotación. Para ello, se modifica el valor de sigma (parámetro descriptor de la orientación) y se seleccionan los puntos que obtengan un menor valor al realizar esta diferencia. Como se ha mencionado antes, este filtro de gauss se ejecuta para la detección de bordes, realiza dos desenfoques en la imagen con diferentes radios y sustrae las dos versiones para obtener un resultado final. Se considera bastante rápido porque puede aplicarse mediante diferentes métodos. Los parámetros a definir para el mismo son los radios de desenfoque utilizados.

  El segundo paso trata de localizar una serie de puntos clave que son refinados mediante un filtro eliminando aquellos que tengan un bajo contraste. En tercer lugar, el algoritmo es capaz de establecer la orientación de dichos puntos clave a través de la graduación de la imagen local. Por último, cuenta con un generador que computa la posición de cada punto clave en la imagen local basado en su graduación y orientación.

- Speed Up Robust Feature (SURF): desarrollado por Herbert Bay en 2006 [27]. Este algoritmo es una variación de SIFT ya que fue desarrollado a partir del primero. En este caso la determinación de la escala se realiza mediante una serie de cajones filtrantes (box filters) en lugar de usar la diferencia gaussiana. Este algoritmo es capaz de obtener una representación visual de la imagen y extraer información detallada del contenido.

  Además, esto permite el uso en diferentes escalas al mismo tiempo. Empleando un método conocido como la pirámide. El algoritmo utiliza un detector BLOB que se basa en la matriz hessiana para la determinación de puntos de interés. Esta puede definirse de la siguiente forma:

$$H(x, \sigma) = \begin{pmatrix} Lxx(x, \sigma) & Lxy(x, \sigma) \\ Lxy(x, \sigma) & Lyy(x, \sigma) \end{pmatrix}$$

Donde $Lxx = (x, \sigma)$ es la convolución del segundo orden derivativo $\partial x / \partial x^2 g(\sigma)$ con la imagen en el punto x, y de manera similar para $Lxy = (x, \sigma)$ y $Lyy = (x, \sigma)$.

Para la orientación usa un sistema de respuesta de ondículas que es un tipo especial de transformada matemática que representa una señal dilatada de una onda finita, tanto en vertical como en horizontal.

Para la detección de imágenes SURF utiliza también la respuesta de ondículas, que captura los puntos que rodean a aquellos que son clave, dividiéndolos en subregiones donde se toma la ondícula y posteriormente es representada para obtener la función descriptora del algoritmo.

Se usa también para la detección el signo del laplaciano, para resaltar los puntos de interés, ya que es capaz de distinguir aquellos con más brillo dentro de fondos oscuros. En caso de coincidencia de características, solo se compararán imágenes si cuentan con el mismo tipo de contraste (basado en el signo laplaciano) lo que permite una detección más rápida.

- Oriented FAST and Rotated BRIEF (ORB): desarrollado por Ethan Rublee en 2011 [26] esta técnica mezcla el algoritmo FAST (Features Accelerated Segment Test) junto con el BRIEF (Binary Robust Independent Elementary Features). El algoritmo FAST se centra en la detección de puntos mientras que BRIEF es un descriptor de estos.

  FAST no detecta la orientación ni la rotación de la imagen, sino que, calcula la intensidad de parches con una esquina ubicada en el centro del objeto. Posteriormente, calcula la dirección del vector que une el centro de la imagen con el centro del parche estudiado. Dándonos así la orientación del mismo.

  Por otro lado, el descriptor BRIEF, actúa pobremente si se produce una rotación en el plano, por tanto, las matrices de rotación se calculan utilizando la orientación hallada mediante FAST y luego son los descriptores BRIEF los que se dirigen conforme a esta.

  El algoritmo FAST, dado un píxel de una matriz, compara su brillo con el de los 16 píxeles circundantes que se encuentran en un círculo pequeño alrededor del píxel original. Estos se clasifican en tres clases (más claros, más oscuros o similares) si más de ocho píxeles son más claros u oscuros se selecciona como punto clave. Siendo así posible la detección de bordes de una imagen.

  Posteriormente, BRIEF toma todos los puntos clave encontrados y los convierte en un vector de características binarias para que juntos puedan representar un objeto. Es por ello que dicho vector también se conoce como descriptor de características.

En nuestro caso, la función “Find_Object_2d” utiliza el algoritmo SIFT descrito previamente tanto para la extracción de puntos clave como para la de descriptores. La función para extraer ambos se toma de la biblioteca de OpenCV. Posteriormente se buscan coincidencias de dichos puntos con los 2 puntos vecinos que se encuentren más cerca de la imagen a comparar.

Este proceso se realiza mediante el algoritmo ORB explicado con anterioridad. Se comparan los descriptores y puntos clave de los vecinos de la imagen propuesta con los originales de la imagen guardada. Con el fin de establecer una homografía o detección de coincidencia, además se muestra el número de elementos comunes de ambos. Por último, si hay coincidencia se configura una manera de mostrarla gráficamente que en nuestro caso es un rectángulo que rodea la imagen como se muestra en la figura 2.2.

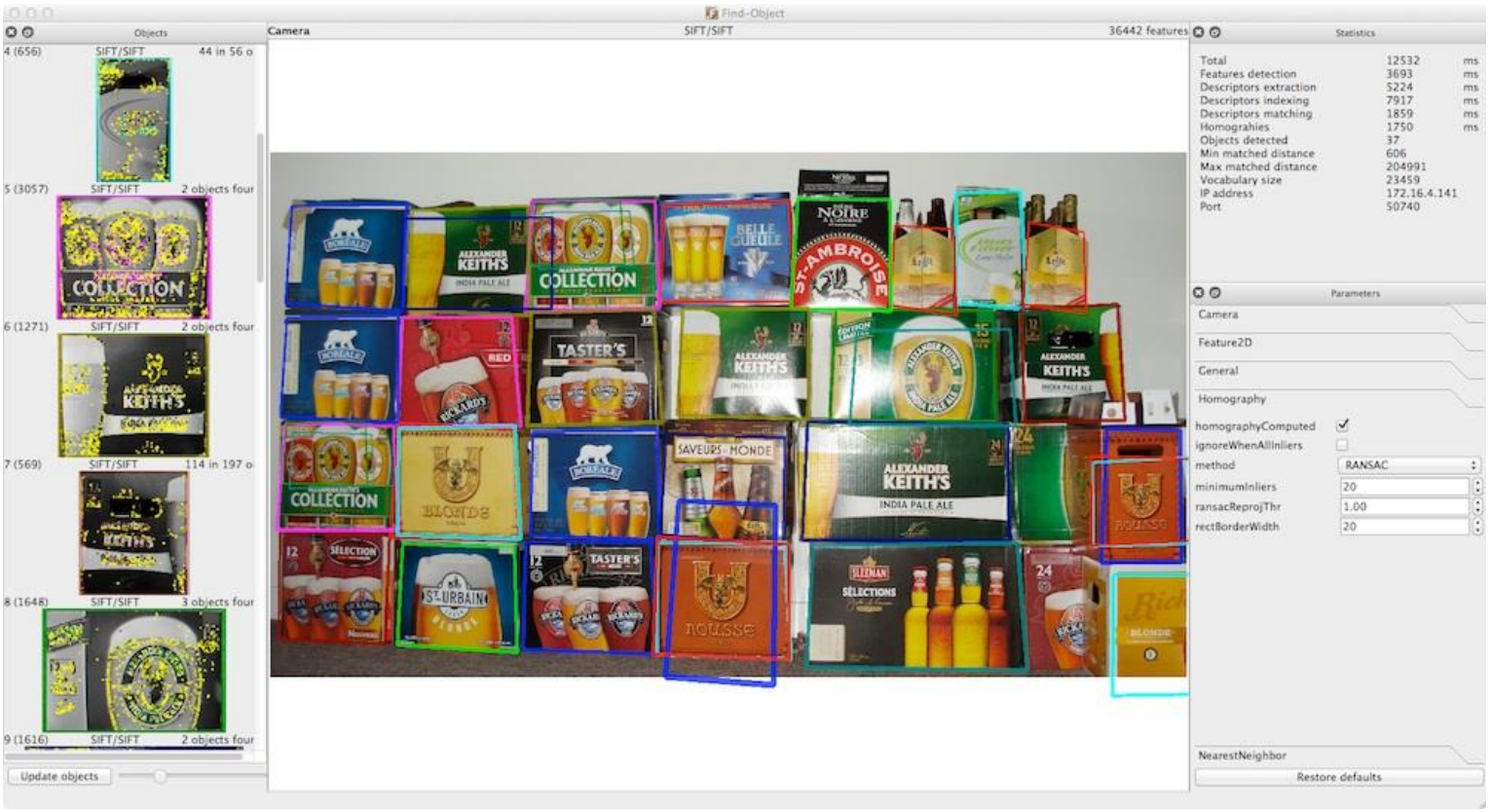


Figura 2-2 ejemplo de detección de objetos en la función "Find_Object" (Fuente: [3])

2.2 Mapeado de entornos

El mapeado de un entorno consiste en la realización de un mapa en 2 dimensiones donde se introduzcan los distintos obstáculos que se hayan encontrado recorriéndolo. Todo ello suele realizarse de múltiples formas como por ejemplo mediante el uso de un sensor LIDAR.

Este es un dispositivo que emite un láser constantemente a su alrededor siendo capaz de recibirlo de nuevo pasado un tiempo. Debido al retraso entre la emisión de la señal y la recogida de la misma, se determina la distancia desde el dispositivo emisor al objeto en cuestión. Realizando el mismo procedimiento de manera repetida somos capaces de modelar un mapa completo de un determinado espacio.

Particularizando su funcionamiento, el objetivo es la creación de una nube de puntos como la mostrada en la figura 2-3, que representa los objetos que hay a su alrededor. Para conocer las coordenadas de la propia nube de puntos debemos conocer la posición del sensor dentro de la misma por lo que también debería incluirse un sistema GPS integrado.

Según el tipo de laser que se integre en el LIDAR este puede ser de varios tipos:

- LIDAR de pulsos: el mapeado se realiza midiendo el retraso entre una señal emitida y su posterior recepción tras impactar en un objeto o superficie del entorno.
- LIDAR de medición de fase: en este se emite un láser continuo, cuando se recibe la señal reflejada se mide la diferencia de fase entre la emitida y la de llegada.

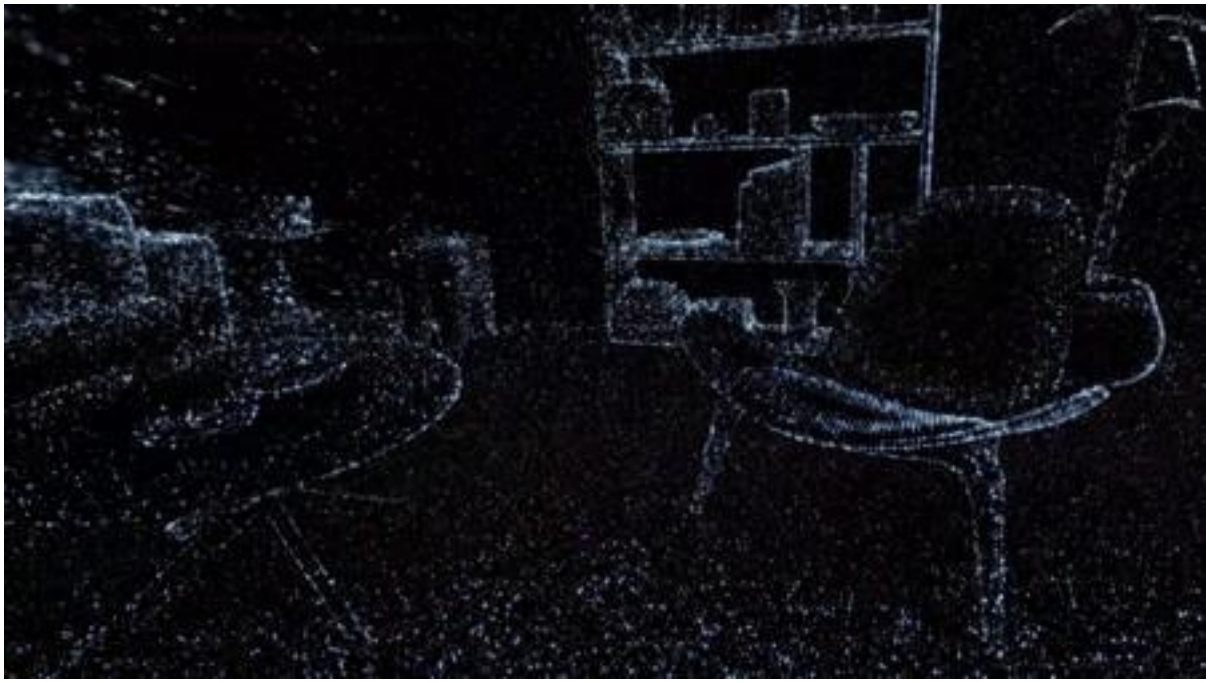


Figura 2-3 nube de puntos generada por LIDAR (Fuente: [4])

2.2.1 Fundamentos del nodo "RPLidar"

En particular, para nuestro proyecto, el paquete que da soporte al sensor LIDAR y nos muestra sus detecciones de manera gráfica es el "RPlidar" de ROS. Este da soporte tanto a los RPLIDAR A1 como a los A2 y A3. En nuestro caso, trabajaremos con el RPLIDAR A2. Este es un sensor de bajo coste utilizado para robots que trabajen en zonas de interior y cuya función sea la de la navegación SLAM que se explicará posteriormente en el proyecto.

El procesamiento de imagen es bastante rápido y es tratado mediante un paquete desarrollado por RoboPeak. Un equipo de investigación y desarrollo de plataformas robóticas fundado en 2009 y conformado por ingenieros de software, electrónicos y expertos en redes. Por tanto, se considera ideal para el mapeado de entornos locales, ya sean casas o habitaciones, así como pequeños espacios cubiertos.

El principal nodo usado para su desarrollo será el "RPlidarNode" que toma los resultados directamente del LIDAR en crudo, para posteriormente convertirlos al formato de ROS como mensaje del tipo "LaserScan" que pueden ser interpretado por el sistema. Posteriormente estos mensajes son publicados en el tópico "scan".

2.2.2 Fundamentos del nodo "gmapping"

La traslación de los datos publicados en este tópico al mapa es llevada a cabo mediante un nodo llamado "slam_gmapping" creado por Brian Gerkey, este es el encargado de crear un mapa de ocupación de celdas en 2 dimensiones a través de la información provista por el sensor en primera instancia. El funcionamiento del nodo se fundamenta en el "GMapping" que se centra en el uso de un filtro de partículas Rao-Blackwellized que realiza una marginación de la distribución de la probabilidad de su espacio de estado.

Este se considera un medio eficaz para solventar el problema del mapeo SLAM. El filtro utiliza una serie de partículas en las que cada una cuenta con un mapa individual y propio del entorno, por tanto, se considera algo muy importante reducir el número de partículas que se presentan. Para ello se propone un enfoque que calcula la distribución de una propuesta en concreto teniendo en cuenta no solo el movimiento del robot sino también sus observaciones más recientes.

Este hecho aclara la posición del dispositivo en cada momento. Además, el algoritmo es capaz de seleccionar aquellas áreas que quedan peor definidas y escogerlas para volver a realizar un muestreo posterior con el fin de completar la información faltante. Esto reduce el número de partículas necesarias para el mapeo y por tanto reduce el problema del exceso de información.

2.2.3 Fundamentos del nodo "tf"

Para la obtención de la posición del dispositivo en cada momento en nuestro proyecto, vamos a basarnos en el nodo "Tf". Creado por el equipo Tully Foote de la biblioteca de ROS. Este, nos permite hacer un seguimiento de una serie de marcos coordinados a través del tiempo. Su funcionamiento se basa en la en la transformación de diferentes puntos o vectores entre cualquiera de los marcos coordinados en cualquier momento del tiempo.

Esto nos da la posibilidad no solo de saber la posición del dispositivo en cada momento, sino que también nos permite de manera más exacta hacer un seguimiento de cualquier parte de nuestro robot. Si contamos con un brazo robótico, por ejemplo, podremos saber dónde se encuentra la base, la articulación 1, la pinza final … como se muestra en la figura 2-4.

En nuestro caso el nodo "tf" irá asociado a la base del robot ya que no cuenta con numerosas piezas que puedan ser interesantes de cara a un seguimiento. Como se ha explicado previamente el nodo no solo guarda la información de la posición actual, sino que también provee la localización del mismo en instantes anteriores. Además, es capaz de coordinarse con los marcos del propio mapa y situar el dispositivo dentro del mismo.

Como bien hemos dicho, el funcionamiento del nodo se basa en los marcos. Estos, se definen como sistemas coordinados, en ROS siempre se representan en 3 dimensiones. Por tanto, el nodo, lo que estudia es la posición de una serie de puntos con 3 coordenadas dentro de un marco. La relación de posición de los distintos marcos se realiza mediante una matriz que representa 6 grados de libertad. Es decir, una traslación seguida de una rotación.

La traslación la representa un vector de 3 coordenadas cada una identificada con un eje del espacio, mientras que la rotación es definida mediante una matriz. Con este sistema el nodo registra por completo el movimiento que realiza un dispositivo desde un marco al siguiente. Guardando así tanto la información relativa al proceso de traslación, como aquella referida a la rotación del mismo.

Para el uso de toda esta información previamente descrita, el nodo "tf" está encargado de publicar tanto la matriz de rotación como el vector de traslación en el tópico "pose". Esta se publica en forma de mensaje geométrico provisto por el nodo "geometry_msgs" desarrollado por Tully Foote. Lo que nos permite adaptar tanto la matriz como el vector a un formato utilizable por ROS para su adaptación y desarrollo.

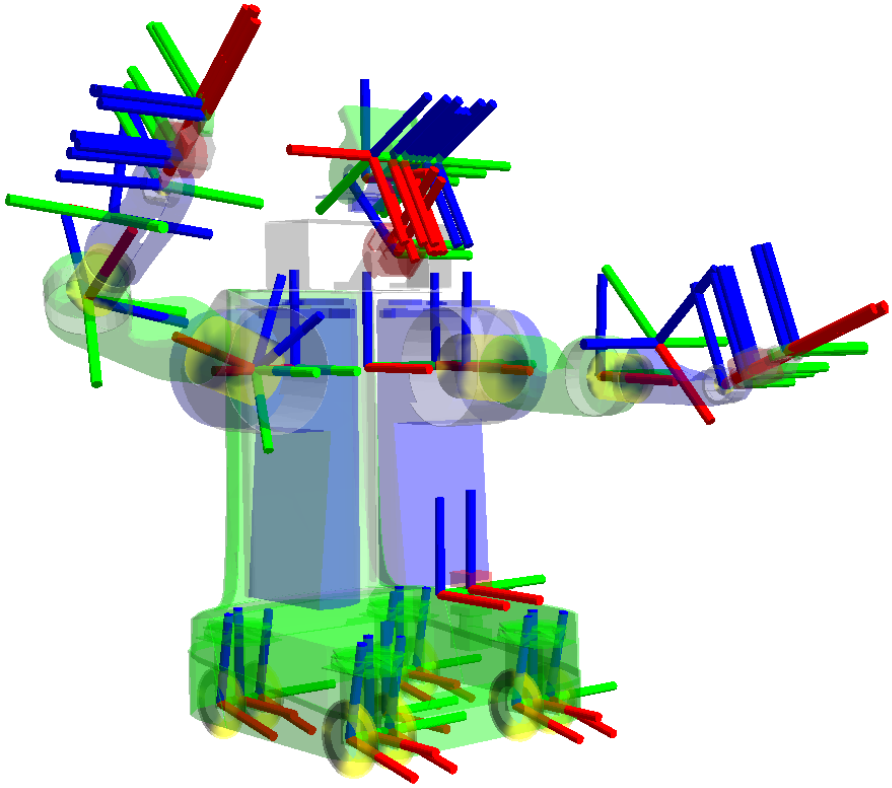


Figura 2-4 ejemplo de localización de identificadores en un robot móvil (Fuente [6])

2.3 Planificación de trayectorias

En este apartado procederemos a desarrollar el fundamento en el cual se basa la planificación de trayectorias dentro de nuestro proyecto. Entendiendo la planificación como la capacidad de un dispositivo de trasladarse desde una posición de origen hasta otra de destino sin colisionar con ningún obstáculo y siendo capaz de diseñar la trayectoria más eficiente posible a través de un mapa.

La mayor parte del desarrollo de la planificación de trayectorias dentro de un robot móvil se centra en la búsqueda y obtención de estrategias de control automático. Con el fin de que el dispositivo sea capaz de elaborar rutas adecuadas y seguras. Es decir, sin colisionar con nada y lo más eficientes posibles dentro de las posibilidades de desplazamiento de nuestro dispositivo.

Por tanto, para poder llevar a cabo una correcta planificación, es indispensable el conocimiento total del modelo cinemático y dinámico del dispositivo. Con ello seremos conocedores de las limitaciones a las que se enfrenta y cuáles son sus posibilidades de uso. Ya que, solo serán realizables aquellos movimientos que puedan ser efectuados por el móvil sin llevarlo a posiciones límite.

La planificación de trayectorias se basa en dos elementos, el primero de ellos es la traslación del robot a la posición idónea para la posterior manipulación. Es decir, en un brazo robótico significaría el desplazamiento de las principales articulaciones hasta la posición en la que la pinza o elemento final sea capaz de alcanzar la posición deseada.

El segundo se corresponde con el control de cada uno de los movimientos descritos en el primer fundamento. Con el fin de que el efector final alcance la posición deseada sin sufrir ningún percance en el proceso. En nuestro caso, la planificación de trayectorias es llevada a cabo en su gran mayoría por el nodo "move_base" de ROS. Creado por Eitan Marder en 2009.

2.3.1 Fundamentos del nodo "move_base"

Este Nodo provee una implementación de una acción que dada una posición de destino en el mundo tratará de alcanzarla desplazando hasta ella la propia base móvil. Para ello, el nodo funciona de manera conjunta con un planificador a nivel local, y otro a nivel global. Además, es capaz de trabajar con los mapas de obstáculos que previamente han sido creados durante el mapeado de entornos.

Dentro de estos mapas se diferencia también la existencia de uno a nivel local, que se refiere a aquello que el dispositivo percibe en el momento exacto donde se estudia, y otro a nivel global que es el elaborado recorriendo la totalidad del mundo en el que se mueve nuestro robot.

Su código y funcionamiento particular se explicará de manera más detallada en el apartado correspondiente de planificación de trayectorias. Aun así, el concepto básico de este nodo es que hace funcionar a la vez una serie de procesos como la rotación, la traslación y los giros; que siempre deben mostrar si están disponibles o no, es decir, si se encuentra algún obstáculo bloqueando alguno de estos movimientos.

Cuando estos procesos se encuentran disponibles, se le pasa la información al proceso central de navegación. Este es capaz de tomar decisiones y aplicar cada uno de los procesos que corresponde para acercarse cada vez más a la posición final deseada. Si ninguno de los mismos se encuentra disponible, se produce un reset de la trayectoria y se vuelve sobre su propio camino hasta ser capaz de diseñar una alternativa.

Este proceso central de navegación es regulado por el nodo "nav_core", que se desarrollará en el apartado de planificación. Es el encargado de proveer una interfaz para la navegación específica del dispositivo. Ofrece un planificador tanto a nivel global como a nivel local.

A nivel global, el nodo se fundamenta en los siguientes algoritmos:

- "Global_Planner" planificador de trayectorias global, rápido y flexible basado en la interpolación.
- "Navfn" planificador de trayectorias basado en celdas que utiliza la navegación para diseñar una trayectoria específica.
- "Carrot_Planner" planificador simple que dado un punto específico trata de desplazar al robot lo más cerca posible incluso si se trata de un obstáculo.

A nivel local, el nodo se fundamenta en los siguientes algoritmos:

- "Base_Local_Planner" provee la implementación del Dynamic Window Approach (DWA) [28]. Método empleado en la evitación de obstáculos a partir de la búsqueda de un entorno seguro y la elección de la solución óptima en dicho espacio.
- "DWA_Local_Planner" implementación del método DWA mucho más fácil y limpia. Es más flexible y sus ejes son variables.

Cabe decir, que este nodo funciona solo ante obstáculos fijos. Es decir, aquellos que son móviles se consideran imposibles de esquivar por nuestro dispositivo. Además, es reseñable que puede ser capaz de detectar que posiciones son alcanzables y cuáles no, cuando va a comenzar a moverse reconoce su entorno local y si no encuentra salida por dos veces declara inalcanzable el objetivo

3 DESCRIPCIÓN DEL ROBOT

En este capítulo voy a presentar el robot con el que la mayoría del trabajo ha sido desarrollado. Este es el modelo ROSbot 2.0, desarrollado por la empresa Husarion e impulsado mediante la plataforma de programación ROS. A continuación, se realizará un análisis de los distintos componentes que forman parte del dispositivo tanto a nivel de hardware como a nivel de software.

3.1 Descripción del Hardware

El Rosbot 2.0 es un robot móvil 4x4 con conducción autónoma, cuenta con 4 ruedas independientes con un motor en cada una de ellas, un sensor LIDAR, una cámara Astra RGBD y varios sensores de distancia, velocidad y orientación, además de contar con una autonomía de 3 horas con su batería.

Sus medidas básicas contando con la cámara y el LIDAR son 200x235x220mm [L x W x H]. Su peso total es de 2.84 kg. El cuerpo está compuesto por una placa de aluminio de 1.5mm de grosor, y se mueve a una velocidad máxima de 1m/s de manera lineal y a una velocidad radial máxima de 420 ë/s de manera rotacional.

Figura 3-1 perspectivas del ROSbot2.0 (Fuente: [1])

A continuación, se realizará una descripción específica de cada uno de los componentes de nuestro robot de manera más específica:

- 4 sensores infrarrojos de distancia modelo VL53L0X con un rango de funcionamiento que alcanza más allá de los 200cm.
- Un controlador de tiempo real modelo CORE2 basado en el microcontrolador STM32F407.
- 4 motores Xinhe XH-25D con 6V DC y una velocidad máxima de giro de 5000rpm.
- Un sensor IMU que sirve como acelerómetro, giroscopio y velocímetro que funciona para los 9 ejes en los que se desplaza el robot.
- 3 baterías recargables de 3500mAh de capacidad con un voltaje nominal de 3.7V estas pueden ser fácilmente sustituibles por unas del mismo tipo.
- Una antena conectada directamente al módulo ASUS Tinker Board del wifi y unida mediante un cable específico al SBC.
- Un SBC del tipo ASUS Tinker Board con 2GB de RAM , 4x1.80 Ghz y 32Gb de memoria en una micro SD, este utiliza el sistema operativo Ubuntu customizado para el uso de ROS.

- Un LIDAR modelo RpLidar A2 360 grados con un rango mayor de los 8 metros de alcance. Provee un escaneo de campo de 360º con una tasa de giro de entre 5 y 10Hz, es capaz de detectar elementos a más de 16 metros. Este soporta entregar de 4000 a 8000 muestras por segundo. por cada rotación provee unas 360 muestras y debido a que la frecuencia va de 2 a 10Hz el dispositivo la soporta sin problema. El propio sensor es capaz de adaptar la velocidad de giro según el entorno lo solicite.
- Una cámara RGBD modelo Orbbec Astra, sistema con posibilidad de visión en 3 dimensiones, con sistemas de color VGA, con seguimiento de gran rango y diseñada para ser compatible con aplicaciones ya existentes.

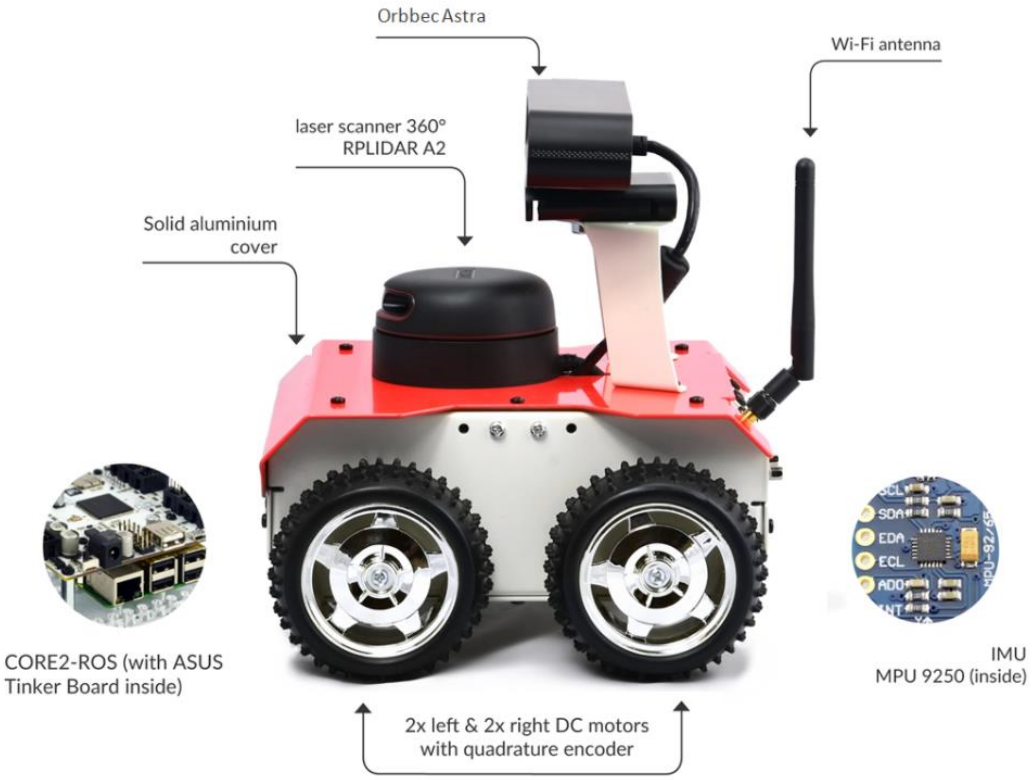


Figura 3-2 vista lateral y posicionamiento de dispositivos en el ROSBot2.0 (Fuente: [1])

3.2 Descripción del Software

El software de nuestro dispositivo puede ser divido en dos partes bien diferenciadas:

- La primera de ellas es un sistema de bajo nivel que trabaja con un controlador de tiempo real (CORE2) y puede ser desarrollado y utilizado mediante el codificador Visual Studio.

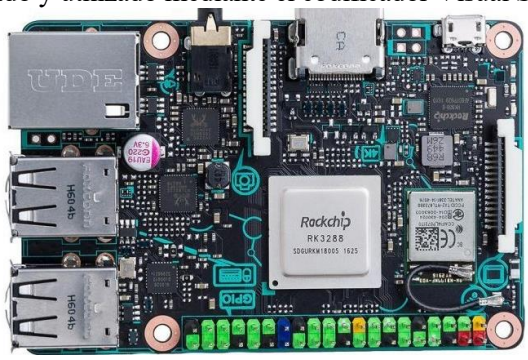


Figura 3-3 controlador de tiempo real core2 (Fuente: [1])

- La segunda parte es el sistema operativo con el que cuenta por defecto, este es el Ubuntu 18.04 que funciona dentro del SBC (Asus Tinker Board) y contiene todos los componentes necesarios para trabajar con ROS de manera inmediata. Dentro del SBC se incluye la propia MicroSD de 32Gb donde está instalado el sistema operativo y sirve como memoria para el dispositivo.

Figura 3-4 Logo Ubuntu 18.04 (Fuente: [7])

- Como se ha explicado previamente el sistema funciona a través de ROS, una plataforma de programación que cada año saca una versión diferente. En general siempre hay varias versiones funcionando al mismo tiempo. En nuestro caso, utilizaremos la versión ROS Melodic Morenia, lanzada el 23 de mayo de 2018 de manera más estable.

Figura 3-5 Logo ROS Melodic Morenia (Fuente: [8])

3.3 Trabajos Previos

Debido a la reciente adquisición de estos dispositivos por parte del departamento son escasos los trabajos de fin de carrera realizados con ellos. Al principio de este proyecto, se planteó la posibilidad de trabajar mediante Matlab en lugar de ROS para manipular el dispositivo. Ya que, un alumno estaba tratando de adaptar el funcionamiento y control del robot a este programa.

Sin embargo, el trabajo aún no estaba finalizado y por tanto no conté con el mismo para el desarrollo de este proyecto. Si utilicé como referencia el trabajo de fin de grado del alumno Manuel Mora [19] que utilizó un robot móvil diferente (Summit-Xl) para desarrollar un sistema de mapeado. Por el contrario, él usó el simulador Gazebo incluido dentro del programa ROS en lugar del robot físico, lo cual hacía algo menos servible éste como ejemplo.

Figura 3-6 Logo Husarion (fuente: [1])

Por lo tanto, en su gran mayoría la información previa al desarrollo de este proyecto ha sido obtenida por mi cuenta a través de Foros oficiales de soporte de la empresa Husarion, tutoriales incluidos dentro de la propia página de la WIKI de ROS, tutoriales específicos añadidos a la web oficial de la empresa Husarion y cursos incluidos dentro de la biblioteca OpenCV.

Figura 3-7 Logo Comunidad ROS (fuente: [8])

4 ROS. ESTRUCTURA Y FUNCIONAMIENTO

En esta parte de la memoria nos centraremos en definir el funcionamiento y la estructura del programa ROS. como ya sabemos, esta herramienta sirve como sistema operativo de robots. En nuestro caso, utilizaremos ROS Melodic Morenia, cuya instalación ya está hecha por defecto dentro de nuestro dispositivo principal el ROSbot 2.0.

Sin embargo, es posible que su instalación en nuestro PC sea también necesaria por lo que en el anexo A se incluirá una guía paso a paso para la misma [3]. Posteriormente nos centraremos en las posibilidades que ofrece esta plataforma para nuestro robot en particular.

4.1 Conceptos Básicos

ROS (Robot Operating System) es un framework para el desarrollo de software dedicado a robots. Este como su propio nombre indica ejerce la función de sistema operativo para los dispositivos robóticos. Fue creado en el año 2007 y en la actualidad provee servicios como el control de dispositivos de bajo nivel, la implementación de funcionalidades comunes, el paso de mensajes entre procesos y el mantenimiento de paquetes.

Su estructura se fundamenta en la arquitectura de grafos. Donde el procesamiento de los mismos se realiza en los nodos. Estos pueden recibir y mandar mensajes de sensores, control o estados. Actualmente funciona en sistemas UNIX (Ubuntu) y también puede instalarse en sistemas Windows, aunque se trabaja para adaptarlo al resto de sistemas operativos.

4.1.1 Desarrollo Gráfico

Los nodos previamente nombrados representan todos los procesos que tienen lugar en ROS. Estos son capaces de comunicarse entre sí gracias a los Tópicos considerados como los bordes de los distintos procesos. El proceso principal se denomina ROS Master y es el que hace todo esto posible ya que se encarga de la creación e inicialización de nodos además de la conexión entre los mismos. Esto se resume en la figura 4-1 gráficamente.

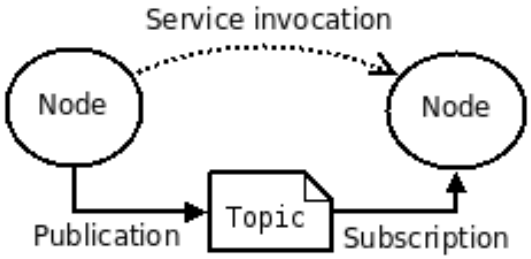


Figura 4-1 estructura básica de funcionamiento de ROS (Fuente: [9])

La arquitectura básica se considera descentralizada ya que los mensajes o llamadas no pasan necesariamente por el ROS Master, sino que este solo se encarga de crear la conexión para que puedan funcionar de manera independiente entre sí. Esto permite la posterior conexión con PCs en el exterior.

Dichos nodos son elementos ejecutables, normalmente programados en c++ o en Python. Pueden funcionar de manera conjunta utilizando archivos launch. Además, estos son capaces de publicar información en los tópicos para que esta sea enviada y distribuida (nodo "Publisher") o bien, suscribirse a un tópico para captar la información que emite (nodo "Suscriber").

Se cuenta también con un "parameter server" encargado de la gestión de la memoria y los archivos de manera ordenada. Es tan necesario para trabajar como el ROS Master y por tanto ambos se suelen iniciar de manera conjunta mediante la orden "ROSCORE". Esta debe ejecutarse siempre antes de empezar a trabajar con el programa para que todo sea inicializado.

Existe otra manera de compartir mensajes entre procesos y es a través de los servicios. Este método requiere un doble sentido en el paso de mensajes, por ello tendremos uno o varios nodos (clientes) que preguntarán ("request") a un solo server (nodo servidor) mandando este último una respuesta ("response"). Como se plasma en la figura 4-2.

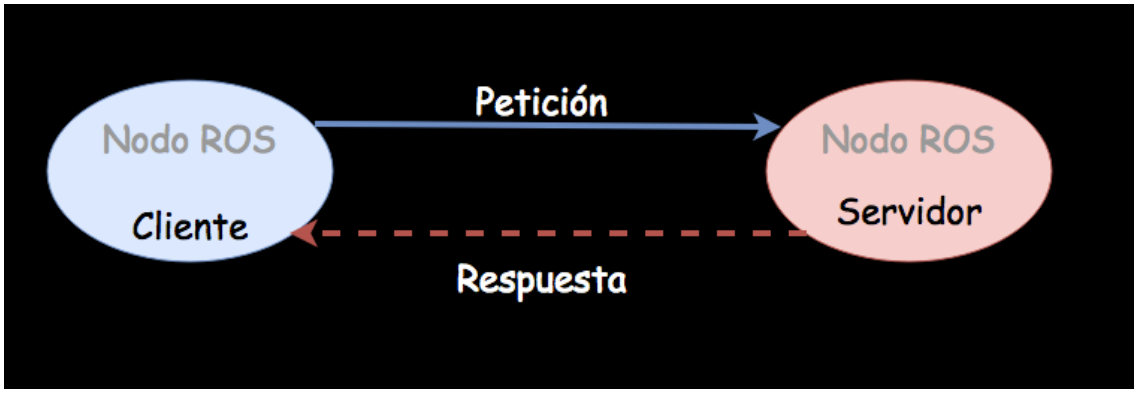


Figura 4-2 esquema de funcionamiento del protocolo Client-Server (Fuente: [9])

4.1.2 Gestión de Archivos

En primer lugar, para comenzar a trabajar con ROS, crear nodos, ejecutarlos y desarrollar sus distintas funciones, debemos tener un espacio de trabajo donde almacenar todo lo creado. Este se denominará por lo general ROS workspace y dentro nos encontraremos todos los archivos con los que trabajaremos.

Tras configurar nuestro espacio de trabajo deberemos crear paquetes ("packages"). Estos son unidades de almacenamiento donde se organizan los distintos códigos, ejecutables, librerías y. launchs necesarios para realizar una tarea específica.

Un nivel por encima de estos paquetes se encuentran los repositorios ("Repositories"), que aglutinan un conjunto de paquetes que cuentan con la misma versión y por tanto pueden ser ejecutados a la vez. A las agrupaciones de paquetes por funcionalidades específicas se les conoce como "metapackages".

Para la transmisión de información se utilizan mensajes ("messages") y archivos de tipo mensaje ("msg") dónde se guardan el tipo de dato de todos los campos de la información. Quedando esta totalmente definida, y los "services", archivos de tipo servicio ("srv") con los que se realizan las peticiones de información y se definen por completo.

Al crear un espacio de trabajo nos aparecerán 3 subniveles dentro del mismo como se muestra en la figura 4-3:

- Src: carpeta donde se ubican los códigos fuente (.cpp) de cada uno de los nodos que tenemos intención de ejecutar con posterioridad.
- Build: aquí se configurarán y se compilarán los distintos paquetes por medio de los archivos modificables CMakeList.txt y el comando "Cmake".
- Devel: unidad donde se almacenan las librerías y los archivos ejecutables del paquete.

Figura 4-3 Distribución de ficheros en el "workspace" (Fuente: [9])

Al crear un paquete también aparecerán 3 directorios dentro del mismo:

- Include: donde hallaremos las cabeceras de los códigos desarrollados posteriormente.
- Launch: carpeta en la que encontraremos los archivos launch, estos se utilizan para poder ejecutar más de un nodo a la vez. Destaca su simplicidad para ser creados y la gran capacidad de solvencia de probl
- emas que aporta su desarrollo.
- Src: directorio muy similar al ya explicado con anterioridad.

En la figura 4-4 se muestra el completo desarrollo de los directorios que pueden incluirse en nuestro workspace de manera gráfica con los tipos de archivos que pueden almacenarse dentro de cada repositorio.

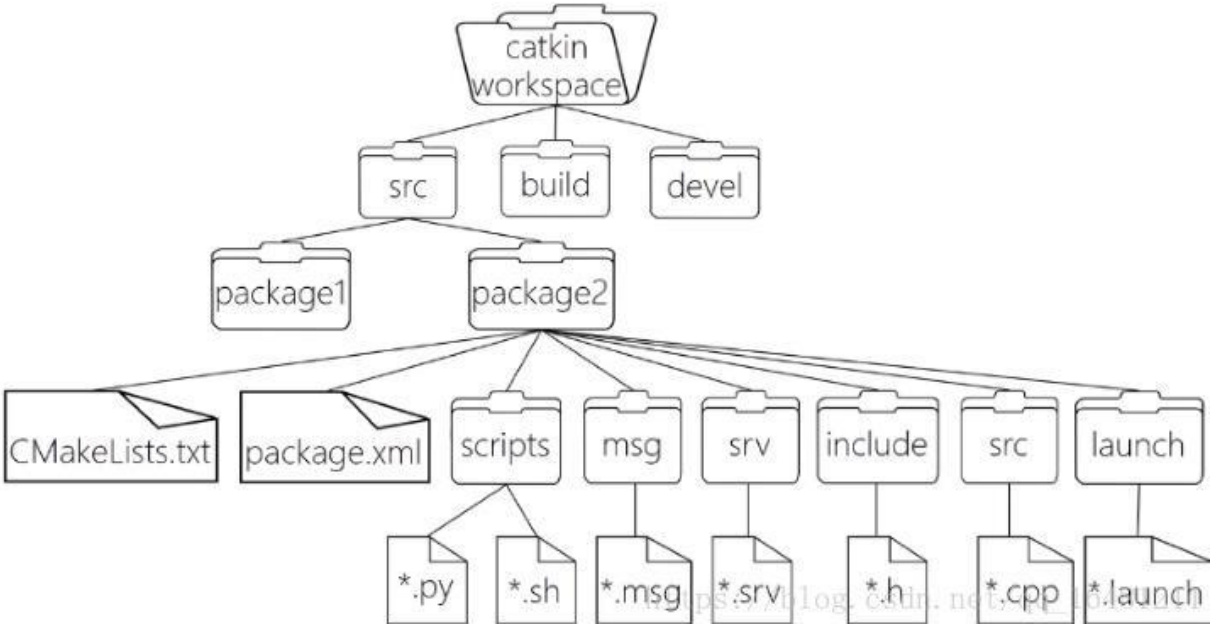


Figura 4-4 distribución de archivos en el sistema ROS (Fuente: [9])

4.1.3 Foros y comunidades

Como se ha mencionado previamente tanto el entorno de ROS como la propia empresa Husarion, son dos asociaciones que se esfuerzan en fomentar el trabajo en equipo y la ayuda mutua de cara a los posibles problemas que pudieran surgir.

Por tanto, cuentan con foros y librerías muy bien cuidadas y atendidas donde los propios trabajadores de estas empresas prestan su ayuda desinteresada y continua ante cualquier situación que pueda complicar nuestro trabajo.

Como referencia se dejan a continuación los diferentes recursos que se han utilizado para la resolución de dudas y aclaraciones necesarias.

- **Husarion Community [**10 **]:** esta página no es más que un foro de preguntas y respuestas dónde puedes tratar de resolver tus dudas buscándolas por categoría o incluso por fecha de publicación. Resulta muy útil el hecho de que sean los propios expertos de la empresa quienes respondan a las preguntas y solucionen los problemas.
- **Wiki de ROS [**11 **]:** Esta página sirve como biblioteca donde consultar cualquier tipo de duda sobre el desarrollo de programas en el entorno de ROS. Cuenta con cursos de formación, calendario de eventos, códigos abiertos y a nivel de software pone a nuestra disposición más de 2000 librerías de paquetes y herramientas comunes para el desarrollo y la depuración de programas en ROS.

4.2 Herramientas utilizadas en el entorno de ROS

Como bien se ha descrito con anterioridad el framework de ROS es muy amplio y pone a nuestra disposición numerosas herramientas de gran utilidad para el desarrollo de trabajos. En este apartado, remarcaremos aquellas más importantes y sobre todo las utilizadas para nuestro proyecto particular.

4.2.1 Find Object 2D

Este programa como su propio nombre indica es el empleado para la detección de imágenes en dos dimensiones directamente desde la cámara instalada en nuestro robot. La herramienta necesita una breve configuración inicial donde se le aclara a qué tópicos suscribirse para recibir información en tiempo real de la cámara. Su funcionamiento se muestra en la figura 4-5.

Una vez configurado pone a nuestra disposición los servicios de detección de imágenes realizados mediante elementos de referencia dentro cada objeto. Para su correcto uso se deben desarrollar dos fases diferentes:

- **Aprendizaje:** en esta parte será necesaria la muestra de la señal de forma repetida frente a la cámara tratando siempre de encontrar la posición en la que sea capaz de tomar más elementos de referencia, una vez hecho esto capturaremos la imagen y se almacenará en un repositorio local.
- **Reconocimiento:** esta segunda parte no muestra obligatoriamente la cámara por pantalla, sino que simplemente comienza a detectar las imágenes previamente incorporadas y muestra por pantalla el identificador que anteriormente se le ha asignado a cada imagen en la parte de aprendizaje.

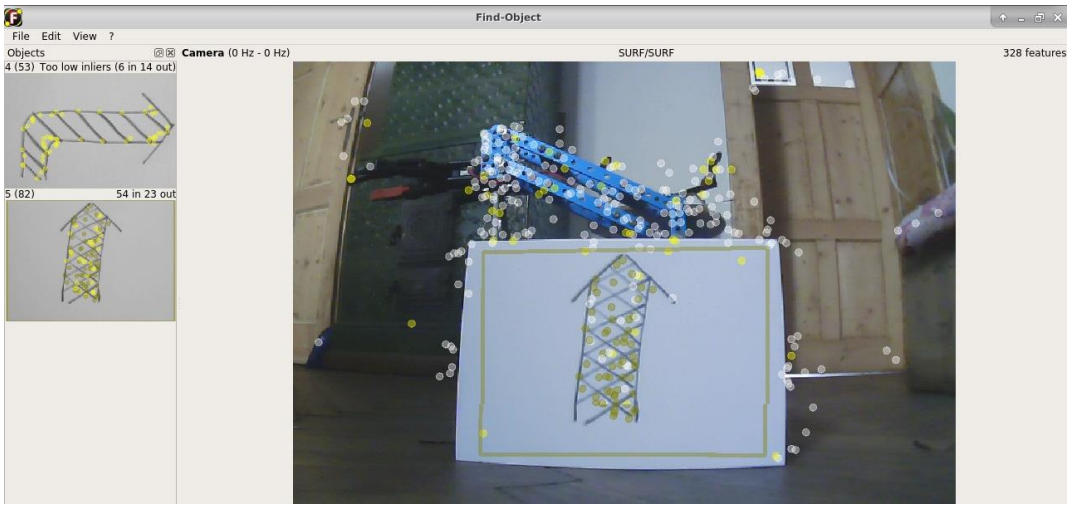


Figura 4-5 Ejemplo del programa Find Object 2d (Fuente: [12])

4.2.2 Rviz

Este programa nos muestra por pantalla información en tiempo real aportada por el robot, se utiliza principalmente para el mapeado de entornos y la posterior detección de obstáculos, su funcionamiento se basa en las distintas "Displays" que pueden ser añadidas de manera manual según aquello que nos interese plasmar.

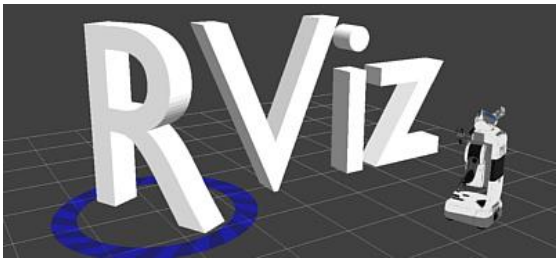


Figura 4-6 Logo Programa Rviz (Fuente: [13])

En nuestro caso funcionaremos principalmente con los siguientes "displays": "Tf" que se encarga de transformar la posición real del robot en datos para ubicarlo en el mapa en tiempo real. "LaserScan" que se encarga de tomar la información provista por el LIDAR para plasmarlo en el programa en forma de obstáculos y límites del entorno. "MAP" que muestra el desarrollo del mapa a tiempo real a medida que avanza el escaneo y "POSE" que realiza una aproximación de la trayectoria seguida por el robot en el entorno.

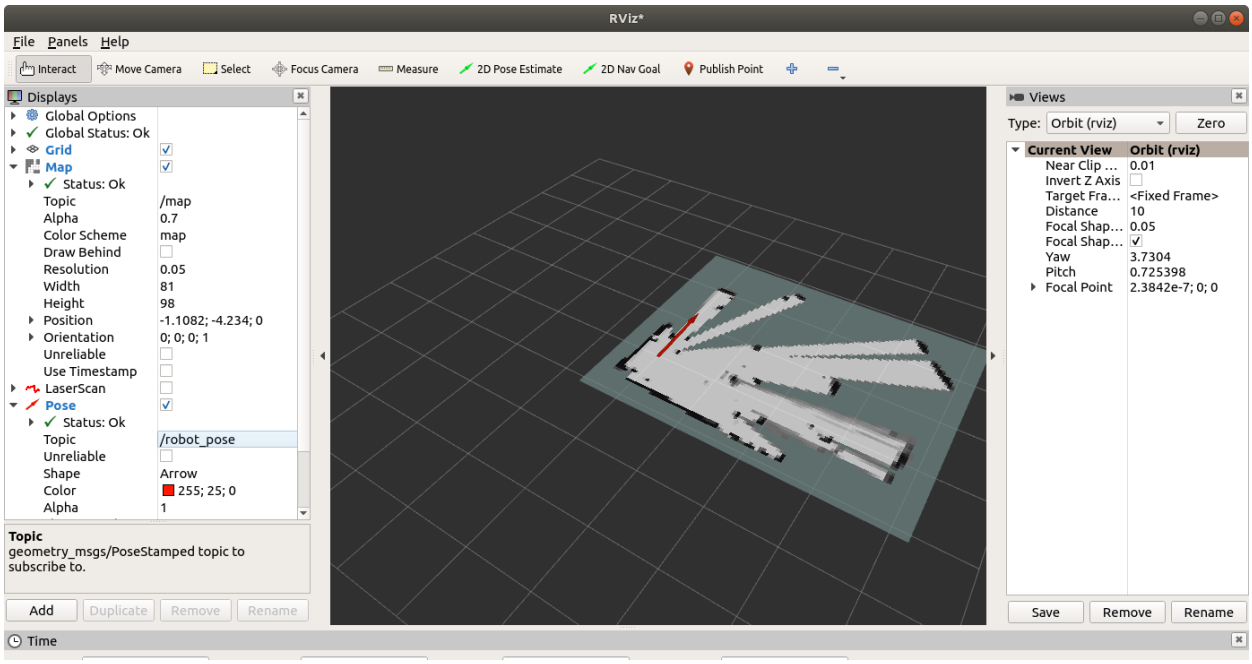


Figura 4-7 ejemplo funcionamiento Rviz (Fuente: [12])

Como se observa en la imagen 4-7, el sistema nos permite una vez creado el mapa un desplazamiento por el mismo desde una posición inicial que siempre debe guardarse hasta otra de destino que puede elegirse arbitrariamente. El propio programa ya detecta como obstáculos todo aquello con lo que pueda colisionar y es capaz de crear una trayectoria evitando estos elementos.

Aunque posteriormente se desarrollará el tema en profundidad, resulta interesante el hecho de que en nuestro robot, el LIDAR encargado de la detección de obstáculos y el mapeo se encuentra a una cierta altura. Es por ello por lo que devuelve ciertos errores al detectar la parte de debajo de sofás o mesas bajas. Esto podría solucionarse mediante el uso de infrarrojos o la propia cámara, pero se escapa a los objetivos del proyecto

4.2.3 RQT_graph

Esta herramienta nos ayuda a apreciar de manera más gráfica y sencilla la cantidad de nodos que se encuentran activos y la suscripción o publicación en los mismos mediante los tópicos entre ellos. Resulta bastante útil a la hora de comprobar y cerciorarse de un correcto funcionamiento del sistema, ya que, con un simple vistazo somos capaces de detectar si hay alguna suscripción errónea, o si se nos ha olvidado iniciar algún nodo.

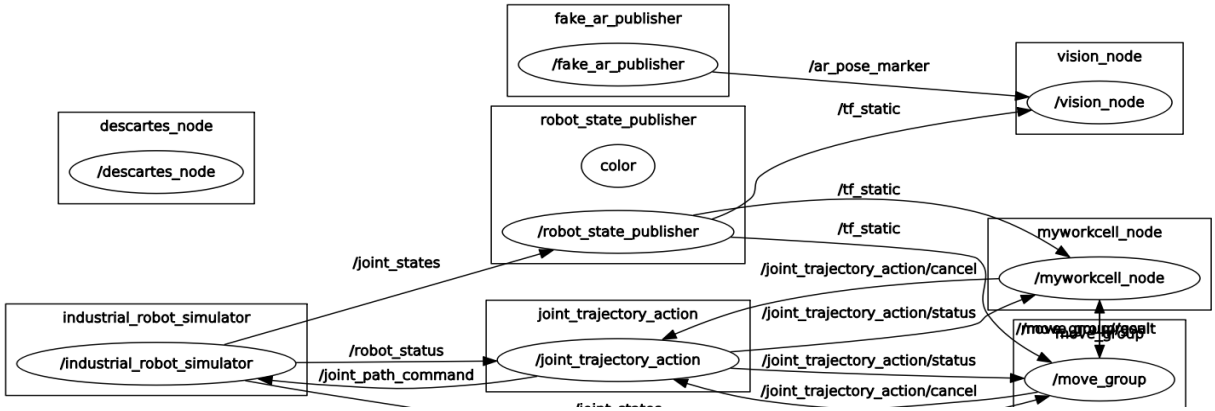


Figura 4-8 ejemplo funcionamiento RQT Graph (Fuente: [13])

4.3 ROSbot2.0 en ROS

En este apartado procederemos a explicar el trabajo previo y los primeros pasos para poder trabajar con el robot mediante ROS. Comenzaremos explicando la creación del espacio de trabajo para seguir con la inicialización de paquetes, nodos como archivos c++ y por último la incorporación de archivos launch.

Cabe destacar que en esta sección se describirán los pasos a seguir de manera no técnica quedando esta explicada en el anexo posterior donde se añaden las líneas de código específico para realizar cada función,

4.3.1 Configuración del espacio de trabajo

Para poder trabajar con ROS y nuestro robot necesitaremos configurar un repositorio local dentro de nuestro dispositivo con el único fin de almacenar todos nuestros archivos, librerías, nodos y paquetes que posteriormente serán compilados.

Crearemos por tanto una carpeta llamada "ros_workspace" y dentro de la misma añadiremos un fichero más llamado" src". Posteriormente, necesitaremos inicializar nuestro espacio por lo que nos moveremos a la carpeta "src" recién creada y ejecutaremos la orden "catkin_init_workspace".

Tras esto nos dirigiremos a la carpeta "ros_workspace", donde deberemos ejecutar la orden "catkin_make" esta actualizará nuestro espacio de trabajo. Será necesaria utilizarla cada vez que introduzcamos un nuevo paquete o nodo para ser reconocido y compilado para su correcto funcionamiento.

Una vez ejecutadas estas órdenes, en nuestro espacio de trabajo aparecerán dos carpetas más "build" necesaria para almacenar elementos usados durante la compilación y "devel" que guardará ficheros externos. Tras esto nuestro espacio de trabajo quedará completamente configurado y listo para su desarrollo. Como se muestra en la imagen 4-9

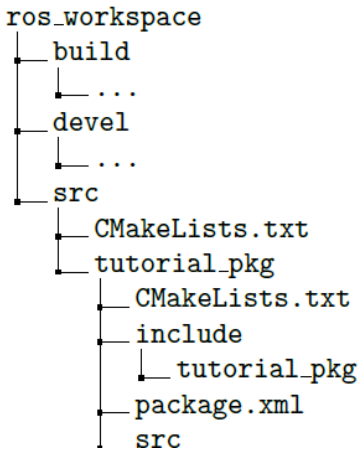


Figura 4-9 distribución fichero espacio de trabajo (Fuente: [12])

4.3.2 Creación de paquetes

Como se ha explicado previamente la distribución de los nodos en ROS se realiza mediante paquetes. Estos almacenan varios nodos y los ejecutan a la vez si se utiliza el package. En este proyecto necesitaremos crear varios paquetes para introducir una serie de nodos de creación propia para el funcionamiento de nuestro robot.

La creación de estos paquetes se realiza mediante la orden "catkin_create_pkg" seguido del nombre que deseemos ponerle y además de manera opcional puede añadirse entre corchetes los nombres de los paquetes necesarios para el desarrollo de nuestros nodos.

A continuación, después de ejecutar nuestros comandos previamente descritos habremos creado un nuevo fichero donde nos aparecerán dos archivos. El primero de ellos es el "CMakeLists.txt" muy importante dentro del paquete. Ya que, contiene las instrucciones para los nodos. Será necesario modificarlo posteriormente para declarar un nodo (archivo c++) como ejecutable y que el sistema pueda reconocerlo cuando sea llamado a funcionar. El segundo es el "package.xml" que contiene datos como el autor del paquete, la versión requerida para su funcionamiento y los paquetes necesarios para un correcto desarrollo. Este no se modifica.

4.3.3 Creación de nodos

Tras la configuración de nuestro espacio de trabajo y la posterior creación del paquete, somos capaces de crear y configurar nuestros nodos. Estos serán archivos c++ (.cpp) que se almacenarán en la carpeta "src" del paquete recién creado no del espacio de trabajo.

Podemos crear un archivo normal y depositarlo en el fichero explicado previamente o podemos utilizar la orden "touch" seguido de la dirección de almacenamiento y el nombre del archivo. Finalmente, el resultado será el mismo, la introducción del archivo dentro de la carpeta correspondiente.

Para cualquier nodo es indispensable seguir la siguiente estructura, en primer lugar, se añadirán las líneas correspondientes a la cabecera del archivo donde se declararán las librerías necesarias de ROS para el funcionamiento del nodo. Esta se realizará mediante #include, para seguir será necesaria la declaración de nuestra función principal con sus variables de entrada y salida como en cualquier otro archivo en "c".

Posteriormente se llamará a la inicialización del nodo mediante la orden "ros::init" que lo registrará en la tarea maestra. Se ejecutará también una comprobación de que todo sigue funcionando mediante la orden "ros::ok", mientras esto se desarrolle correctamente el nodo se seguirá ejecutando sin problemas.

A continuación, como he explicado previamente será necesaria la modificación del archivo CmakeLists.txt del paquete creado para la declaración de nuestro archivo como nodo. Este proceso se repetirá de igual manera cada vez que introduzcamos un nuevo nodo en el paquete.

En primer lugar, se dará nombre al proyecto y se establecerá la versión mínima necesaria para su funcionamiento. Seguidamente, se configurará una serie de elementos básicos mediante la orden "add_compile_options". A continuación, será necesario declarar el nodo como ejecutable mediante la orden "add_executable" seguido del nombre del proyecto y la ubicación del nodo. Por último, se declararán las librerías necesarias para dicho nodo.

Como se ha explicado previamente, antes de que todo esto pueda ejecutarse debe compilarse y actualizar el espacio de trabajo de cara a sistema. Por ello nos moveremos a la carpeta "ros_workspace" y ejecutaremos de nuevo la orden "catkin_make".

Si todo ha ido como debiera, esta orden no devolverá ningún fallo. Si devuelve alguno, deberemos revisar el código o alguno de los pasos explicados previamente ya que habremos cometido un error con seguridad. Antes de ejecutar el nodo, debemos configurar la fuente de partida y las variables de entorno mediante la orden "source ~/ros_workspace/devel/setup.sh" esta deberá ser utilizada antes de comenzar a ejecutar cualquier paquete o nodo ya que sin ella no será posible su funcionamiento.

Finalmente, para ejecutar el nodo recién creado solo debemos hacer uso de la orden "rosrun" seguida del nombre del paquete y del nombre del nodo. Para comprobar si todo funciona correctamente podemos ejecutar la orden "rosnode list" con la que debe aparecer toda la lista de nodos activos incluyendo este último recién creado. En la figura 4-10 se muestra un ejemplo de las diferentes opciones de funcionamiento ofrecidas por el programa incluyendo la de la lista de nodos activos.

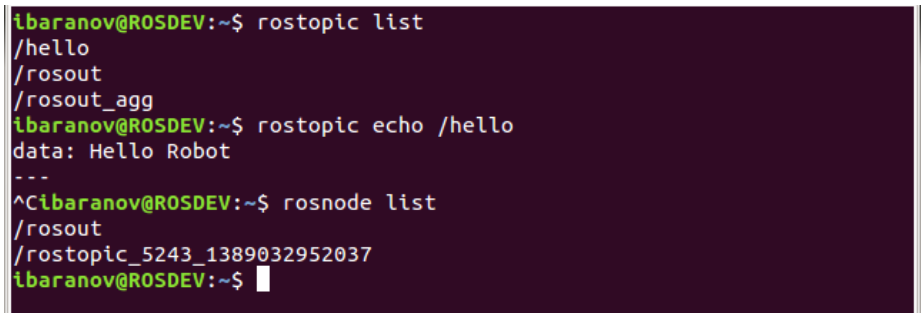


Figura 4-10 ejemplo funcionamiento ROS

4.3.4 Creación de archivos Launch

Como se ha explicado previamente, los archivos. launch son bastante interesantes porque son capaces de lanzar varios nodos al mismo tiempo. Es necesario por otro lado, configurar un espacio para todos estos archivos dentro del paquete.

Por tanto, lo primero que debemos hacer es crear una nueva carpeta dentro de nuestro paquete llamada "launch". Una vez hecho esto podemos comenzar a introducir archivos de este tipo en el interior y ejecutarlos mediante la orden "roslaunch" seguido del nombre del paquete y del nombre del launch.

Dentro del contenido de los archivos launch es necesario describir el paquete en el que se encuentra, el tipo de archivo que es, su nombre y el tópico al que se suscribe para comenzar a trabajar. Todo ello se añade antes y después de una serie de configuraciones y se hace una vez por cada nodo que queramos ejecutar.

4.3.5 Conexión remota

Dentro de los primeros pasos necesarios para la configuración de nuestro dispositivo en ROS tenemos la opción de conectar un ordenador de manera externa para controlar telemáticamente nuestro dispositivo. Para ello será necesario que el ordenador desde el que vaya a realizarse la conexión tenga instalado Ubuntu (18.04 en nuestro caso) además del programa ROS.

Si se cumplen los requisitos previos, el siguiente paso es cerciorarnos de que ambos dispositivos (robot y ordenador) están conectados a la misma red Wifi, de la que, por otro lado, necesitaremos su dirección IP. Para ello, debemos ejecutar el comando "ifconfig" y apuntar la dirección IPv4 (inet). Como muestra la imagen 4-11

```
jualeomun@ubuntu:~$ ifconfig
ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.180.129  netmask 255.255.255.0  broadcast 192.168.180.255
        inet6 fe80::61c5:6e24:af06:1245  prefixlen 64  scopeid 0x20<link>
        ether 00:0c:29:62:db:fe  txqueuelen 1000  (Ethernet)
        RX packets 108800  bytes 131903674 (131.9 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 51898  bytes 6142072 (6.1 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Figura 4-11 detección de la señal IP

Una vez hecho esto, solo deberemos ejecutar el comando "ssh husarion@xxx.xxx.xxx.xxx" donde en lugar de las x debemos introducir la dirección IP hallada previamente. Tras esto, se nos requerirá una contraseña que es "**Husarion**". Si todo se ha realizado correctamente nos aparecerá por pantalla un pequeño mensaje como el de la figura 4-12 y significará que ya estamos conectados. En adición, nos informa de la última conexión que tuvo el dispositivo.

```
jualeomun@ubuntu:~$ ssh husarion@192.168.1.66
husarion@192.168.1.66's password:
Last login: Mon Apr 26 09:36:54 2021 from 192.168.1.49
 _____     ____     _____  _              _     ___      ___
|  __ \   / __ \   / ____|| |            | |   |__ \    / _ \
| |__) || |  | || (___  | |__     ___    | |_     ) |  | | | |
|  _  / | |  | | \___ \ | '_ \   / _ \   | __|   / /   | | | |
| | \ \ | |__| | ____) || |_) || (_) || |_   / /_  _| |_| |
|_|  \_\ \____/ |_____/ |_.__/  \___/  \__| |____|(_)\___/

    Documentation: https://husarion.com/manuals/
```

Figura 4-12 ejemplo conexión remota ROS

Desde este momento podremos controlar de manera telemática nuestro robot. Esto será muy útil de aquí en adelante para el control del dispositivo por teclado o la muestra de resultados en tiempo real mientras el dispositivo funciona. A modo de ejemplo puede ejecutarse desde el ordenador algún nodo para ver si funciona o abrir incluso la cámara.

Cabe destacar que, como se ha explicado previamente, en ROS todo funciona mediante nodos. Por tanto, el nodo maestro es aquel que ha iniciado mediante “ROSCORE” el sistema, y este siempre suele ser el propio robot. Mediante la conexión ssh recientemente explicada lo que hacemos es introducir un nuevo nodo que cuenta con accesos y permisos a la ejecución de programas dentro del dispositivo.

5 DETECCIÓN DE SEÑALES

En este capítulo nos centraremos en describir y comentar el proceso que se ha llevado a cabo para que nuestro robot sea capaz de reconocer, detectar y actuar en consecuencia de una serie de señales de tráfico. El objetivo principal de esta sección es llegar a definir la inteligencia artificial, en su vertiente del reconocimiento de imágenes dentro de videos en tiempo real. Con el fin de dejar explicado el funcionamiento de nuestro sistema.

Comenzaremos, por tanto, analizando el método de reconocimiento visual con el que cuenta nuestro dispositivo. Además de la manera de configurar el mismo para el aprovechamiento y desarrollo de nuestro proyecto. Tras el aprendizaje de la detección, se pondrá a prueba su funcionamiento y se diseñará un ejecutable que sea capaz de tomar decisiones en consecuencia.

Por otro lado, será necesario aclarar el archivo principal que rige al robot y que se ejecuta cuando se detecta alguna imagen, este va a ser capaz de tomar decisiones y actuar en consecuencia de la señal apercibida a su alrededor.

5.1 Introducción

Como se ha explicado previamente el sistema de reconocimiento del robot funciona mediante su cámara RGBD integrada en el dispositivo. Esta es capaz de detectar imágenes mediante el reconocimiento de características básicas de las mismas, ya sean estas, puntos específicos, líneas, límites o colores.

La detección de imágenes cuenta con dos partes bien diferenciadas. La primera de ellas es el aprendizaje del objeto en cuestión. La señal se presenta delante del sistema de visión y se extraen de la misma una serie de elementos interesantes para su posterior reconocimiento. Esto debe realizarse con cada una de las señales que nos interese detectar.

La segunda parte es el reconocimiento, en esta el sistema funciona constantemente. Cada píxel de la cámara es analizado y tratado en busca de coincidencias con las imágenes previamente guardadas. Si finalmente se consigue el número mínimo de coincidencias necesarias, entonces se considera una imagen detectada. Si no, se continua con el proceso hasta que ocurra.

En esta sección necesitaremos usar como nodos básicos el “find_object_2d” utilizado para la detección de objetos en videos de tiempo real y el “Astra.launch” que se encarga del funcionamiento de la cámara y su configuración básica.

5.2 Aprendizaje de imágenes

Para esta primera parte se procederá al aprendizaje de objetos, en este caso señales a detectar por nuestro sistema. Cabe destacar que cualquier elemento puede servir para ser detectado, pero mientras más limites, colores, elementos y puntos característicos tenga más fácil será posteriormente su detección.

5.2.1 Configuración previa aprendizaje de imágenes

Lo primero que debemos hacer es ejecutar el nodo “find_object_2d” que por defecto está suscrito al tópico “image” y que deberemos cambiar al tópico “image_raw”. Para cambiar la suscripción de un nodo a otro tópico se usa la orden “remap” seguido del tópico original y el nuevo de destino. Para el correcto funcionamiento de la cámara debemos ejecutar el nodo “Astra.launch” que se encarga de su configuración e inicialización. Para automatizar todo este proceso se utiliza un archivo launch que es capaz de realizar todo lo descrito anteriormente de manera simultánea. Una vez ejecutemos este archivo, nos aparecerá una ventana del programa “find_object_2d” donde podremos apreciar como el sistema busca características de los objetos que percibe como muestra la imagen 5-1.

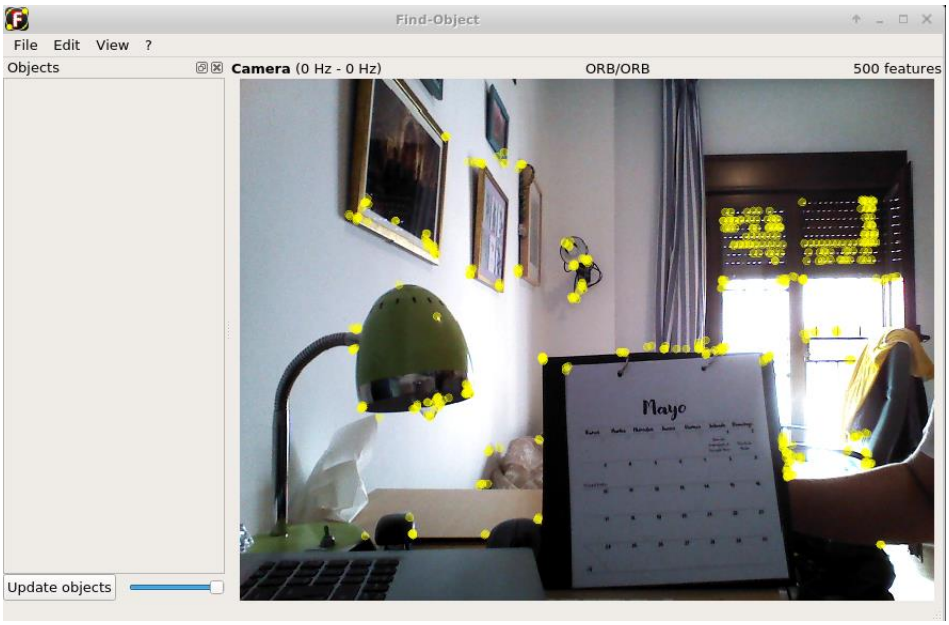


Figura 5-1 Ejemplo propio funcionamiento Find Object

5.2.2 Inclusión de objetos en "Find_Object_2d"

En nuestro caso y a modo de ejemplo utilizaremos una señal de limitación de velocidad a 40km/h como la que se muestra en la figura 5-2. Una vez la coloquemos delante de la cámara y percibamos que el sistema toma el mayor número de puntos de referencia posible (puntos amarillos) entonces abrimos "edit" y seleccionamos "add object". Con lo que nos aparecerá una ventana muy similar a la anterior, pero en blanco y negro en esta ocasión.

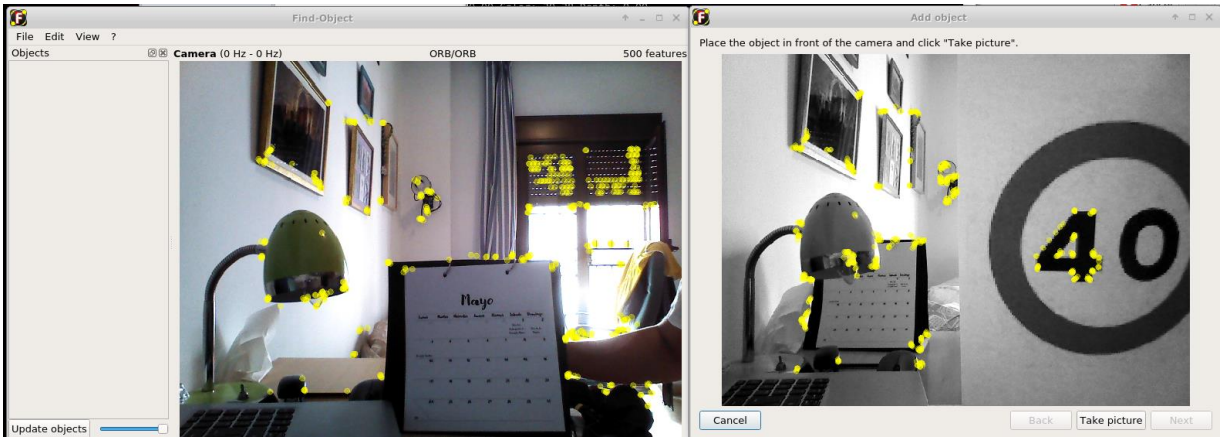


Figura 5-2 ejemplo captura de imagen en Find Object

Como puede observarse el programa es capaz de detectar con mayor facilidad los números que cualquier otra cosa, es por ello por lo que las señales alfanuméricas serán las que tengan una mayor facilidad de reconocimiento en nuestro proyecto. Tras cuadrar la imagen correctamente en la ventana de opciones nos permiten seleccionar el área de interés. Será por tanto necesario recortar la imagen para eliminar aquellos elementos inútiles dentro de la fotografía y centrarnos únicamente en la parte que incluye la señal u objeto deseado.

Este proceso debe ser llevado a cabo con cautela para evitar que se guarden elementos innecesarios en la memoria del objeto. Una correcta selección donde solo se toman puntos identificativos de la señal aparece en la imagen 5-3 añadida a continuación.

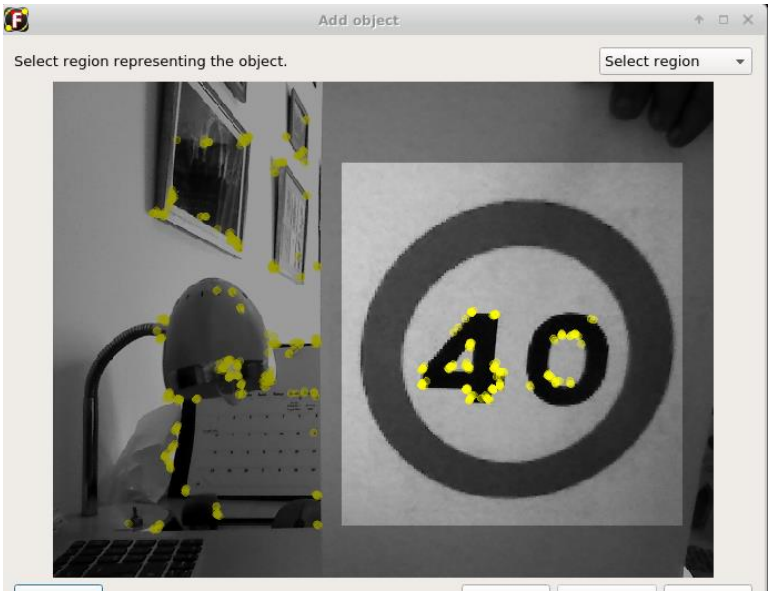


Figura 5-3 Detección de puntos característicos de una imagen

Una vez realizado este último paso solo debemos guardar el objeto y volveremos a la pantalla principal del sistema donde podremos apreciar que un nuevo elemento ha sido añadido. Además, si acercamos a la cámara la señal previamente mostrada, puede observarse como se detecta y se rodea de un color característico indicando que se han encontrado el mínimo de características comunes con la imagen enseñada previamente. Este proceso aparece gráficamente en la imagen 5-4

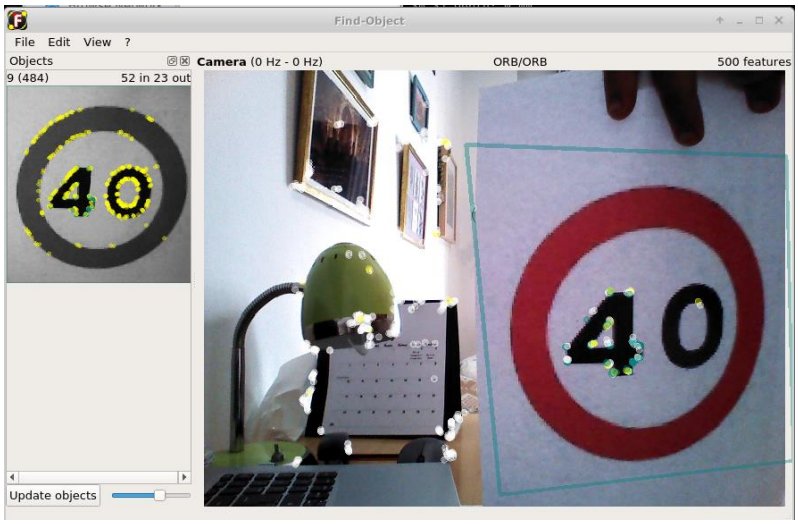


Figura 5-4 Detección de una imagen por parte del programa

Por último, solo deberemos guardar todos los objetos que hayan sido añadidos en un fichero local y guardar la dirección de este. Ya que, será necesaria para el desarrollo posterior del proyecto. Esto puede hacerse fácilmente en la pestaña de “file” donde seleccionaremos “save objects” y nos derivarán a la elección de la carpeta deseada para su almacenaje.

Es destacable el hecho de que el funcionamiento de la cámara más la detección de objetos, supone una gran ocupación de la RAM del dispositivo. Por tanto, muchas veces cuando tratamos de guardar los objetos, el sistema se queda algo colgado. Como recomendación puede ser necesario pausar la cámara en este momento. Dicha opción está disponible en la pestaña “file”, ya que con esto se hace más liviana la carga y se posibilita guardar las imágenes sin dificultad.

5.2.3 Funcionamiento del nodo "Find_object"

En cuanto al algoritmo utilizado por la función Find_object_2d, al que ya se hizo referencia en el capítulo dos de este mismo proyecto, si nos centramos en su funcionamiento en profundidad, cabe decir que, en primer lugar, toma las siguientes bibliotecas de la librería de OpenCV:

- "core" nos permite generar interfaces gráficas.
- "highgui" nos permite abrir fácilmente ventanas, mostrar imágenes, leer o escribir archivos relacionados con imágenes y principalmente operar con la cámara del dispositivo.
- "features2d" relaciona características bidimensionales, usa principalmente funciones de comparación y detección de puntos característicos basados en el algoritmo ORB explicado previamente. Cabe decir que los algoritmos que usa son de software libre.
- "nonfree" forma parte de la propia biblioteca de "features2d" pero en este caso incluye algoritmos del tipo SIFT y SURF que están protegidos por ciertas patentes.
- "calib3d" en ella encontramos funciones relacionadas con la reconstrucción de imágenes en 3 dimensiones, la visión estéreo y la calibración de la propia cámara.

Posteriormente, la función carga las imágenes en escala de grises en su memoria interna para que sea más fácil su detección y posterior comparación. Además, crea tanto vectores que almacenen los puntos clave del objeto en cuestión como variables encargadas de almacenar los elementos descriptores de la imagen.

Como ya se ha definido previamente los puntos clave serán los encargados de definir la forma y los caracteres del elemento mientras que, los descriptores, se encargarán de definir la situación de la misma en el momento de la captación. Definiendo así la escala del objeto, la orientación de la imagen, la luminosidad y el tamaño del elemento en cuestión.

Seguidamente, la función "features2d" comienza a realizar la detección y posterior extracción de puntos clave para introducirlos en el vector previamente creado. Dicha función ofrece la posibilidad de trabajar con multitud de los algoritmos explicados en el capítulo dos, pero en nuestro caso usará SIFT. Este realizará la escala de la imagen, posteriormente hallará y hará un refinado de los puntos clave de la misma y tras esto, especificará la orientación de dichos puntos para por último particularizar la posición de los mismos dentro del objeto.

Para todo ello, los parámetros que necesita que se le pasen son los siguientes:

- "nfeautes" número de puntos característicos que retener, estos se organizan de mayor a menor importancia según sus propiedades, primando en los mismos aquellos con un mayor contraste.
- "noctavelayers" número de capas usadas en la resolución de la imagen, su valor habitual es 3, aunque actualmente el algoritmo determina su valor de manera automática al percibir la misma.
- "contrastThreshold" se usa con el fin de filtrar aquellos puntos clave con bajo contraste y con características semi uniformes en el plano de la imagen, mientras mayor sea su valor menor será el número de puntos característicos detectados.
- "EdgeThreshold" se usa para filtrar los puntos clave que se encuentren en los límites o bordes del objeto. Su funcionamiento es contrario al anterior ya que mientras mayor sea su valor, mayor será el número de puntos característicos extraídos.
- "sigma" valor que determina el funcionamiento de la diferencia gaussiana, filtro empleado en la detección de bordes, su valor debe ser menor mientras menor sea la calidad de la imagen tomada ya que la detección de bordes y los radios de desenfoque se realizarán de manera más abrupta.
- "Descriptortype" valor que determina el tipo de descriptores que van a utilizarse, solo puede darse el valor CV_32F, que corresponde con el algoritmo SIFT que va a utilizarse.

Tras la configuración de todos estos parámetros, la función es capaz de extraer tanto los puntos clave del objeto como los descriptores del mismo. Seguidamente, los almacena en los vectores descritos previamente y con esto termina la parte de aprendizaje de la función en el contexto de la detección de imágenes. En el apartado de reconocimiento se continuará desarrollando el algoritmo de funcionamiento.

5.3 Límites y características de la detección de señales

Llegados a este punto es necesario conocer la limitación que puede llegar a imponernos nuestro sistema. El hecho es que la detección de señales se realizará mediante una cámara Orbbec Astra RGBD que cuenta con características de sobra para llevar a cabo esta función. Por tanto, lo que se expone a crítica no es más que el sistema empleado para la detección de objetos, en nuestro caso el "find_object_2d".

Como se ha contado previamente este utiliza una serie de puntos de referencia para la detección de imágenes y su posterior comparación con las percibidas en tiempo real. Para la experiencia, se han utilizado numerosas señales fácilmente apreciables en el día a día habitual en la carretera. Como por ejemplo, stops, cede al paso, giros obligatorios y limitaciones de velocidad.

El objetivo de la experiencia era saber a qué distancia comenzaba a reconocer cada señal, con que ángulo, con que iluminación y con qué inclinación el sistema detectaba el elemento mostrado. Además, se pone a prueba si el sistema detecta con mayor facilidad flechas, números o letras dentro de las señales.

Tras varios ensayos puede aclararse que el sistema funciona de manera muy eficiente en la mayoría de todos los aspectos. Cabe decir que las señales alfanuméricas, aquellas que cuentan con números y letras, son mucho más reconocibles por el dispositivo. Desde una distancia de más de 1.5m es capaz de detectar este tipo de señales. En adición, con un ángulo de visión menor de unos 30º también es capaz de notar su presencia, la luminosidad se considera que no supone algo determinante, siempre y cuando haya una iluminación mínima para actuar con normalidad, aunque muchas veces supone un problema en el enfoque.

Por otro lado, aquellas señales que contienen flechas se resisten algo más a la hora de ser detectadas. Sobre todo, cuando el robot se encuentra en movimiento. A pesar de esto, el funcionamiento se considera bastante correcto, es destacable el hecho de que en este tipo de señales sí que importa algo más la inclinación de la cámara o la imagen. Además, se recomienda utilizar flechas que describan una curva (figura 5-5) y no sean rectas (figura 6-6). Ya que estas suponen una mejoría en la detección de puntos de referencia que ayudan en el posterior reconocimiento.

Figura 5-5 Señal de giro a la derecha usado (Fuente: [14])

Figura 5-6 Señal giro a la derecha no recomendada

Es importante resaltar que la detección de la imagen no es estable. Es decir, al ser un video recibido en tiempo real, el sistema puede detectar en un primer momento la señal y que posteriormente siga avanzando. Lo que puede llegar a suponer que al procesar el reconocimiento, la señal se pierda de vista y por tanto a pesar de estar en casi la misma ubicación no reconocer el objeto que hacía un instante había avistado.

En líneas generales la detección de objetos se considera bastante correcta, para los objetivos de este proyecto ni mucho menos supone una limitación si no que cumple su función de sobra. Dentro de lo exigible, lo más reprochable podría ser el hecho de la detección de señales en movimiento ya que en algunas ocasiones supone una complicación si la velocidad es algo más elevada de lo esperado.

5.4 Detección de objetos

Aunque dentro del propio apartado de aprendizaje ya se ha hablado algo de la detección y de como funciona, para el caso en el que lo que queramos no sea ver cómo se desarrolla de primera mano el reconocimiento en sí, y solo queramos actuar en consecuencia del mismo, utilizamos este apartado. En este caso, variaremos algunos parámetros del mismo nodo empleado para el aprendizaje.

En primer lugar, el parámetro "gui" es aquel que regula la aparición de la ventana representativa de la aplicación "find_object_2d" por tanto en este caso como no vamos a añadir ningún nuevo objeto no será necesaria su apertura. Es por ello por lo que se le dará un valor "false".

Por otro lado, será necesario configurar el parámetro "objects_path" que no es más que la dirección del fichero que almacena todas las imágenes capturadas previamente. Esta se guardó en la ventana de "save objects" del apartado de aprendizaje y será necesario recuperarla. Nuestra dirección de almacenaje se muestra en la imagen 5-7

Cabe destacar que la configuración de estos parámetros puede realizarse de manera automática y así se ha hecho en el archivo launch. Sea como fuere, una vez realizada ya puede comenzar a trabajarse mediante la detección. El sistema arranca de nuevo el mismo nodo que en ocasiones anteriores pero esta vez no aparece la ventana de la herramienta "find_object_2d". Lo que si aparece es una serie de mensajes en la terminal.

Estos son mensajes del tipo "Float32multiarray", básicamente el nodo se dedica a publicar en el tópico "/objects" este tipo de elementos donde los datos que encontramos son el ID del objeto, su tamaño y su orientación. El elemento más importante que muestra es el ID del objeto, este se considera un identificador del mismo. Si nos fijamos en la carpeta donde hemos guardado el objeto previamente, podemos observar cómo se ha guardado con un número particular, este es su identificador, y es el que posteriormente va a mostrar el sistema cuando sea detectado. A modo de ejemplo, utilizamos la señal de stop que se muestra a continuación, y como puede comprobarse en nuestra carpeta cuenta con el identificador número 5 que le diferencia de los demás objetos.

Figura 5-7 ejemplo fichero almacenamiento de imágenes a comparar

Al ejecutar la orden que nos muestra lo que está publicando el tópico "/objects", en un primer momento antes de presentar ningún objeto no escribe nada por pantalla salvo el mensaje previamente descrito en blanco, sin embargo, al aparecer la señal de STOP comienza a devolver su identificador además de otros datos que carecen de interés para el proyecto. Como ejemplifica la imagen 5-8

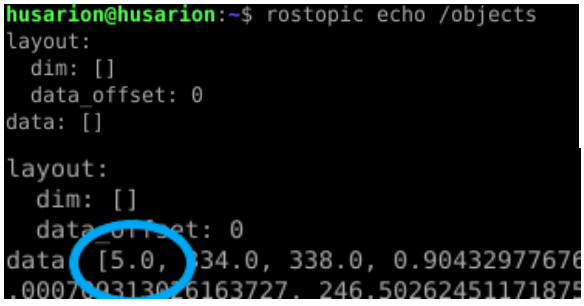


Figura 5-8 ejemplo detección de imagen

Como dato de interés, destaca la rapidez con la que detecta la imagen y su inminente capacidad de respuesta tras el reconocimiento de la misma. Que en un futuro va a permitirnos diseñar un programa que actúe en consecuencia de manera correcta y a tiempo.

5.1.1 Funcionamiento del nodo “Find_Object”

Dentro de la aplicación find_object_2d utilizada por ROS nos encontramos que el reconocimiento de la imagen se realiza a través de la librería FLANN que se explica a continuación. Al inicio de la visión artificial, las coincidencias en los puntos de una imagen almacenada y otra propuesta para comparar siempre fueron analizadas mediante la distancia euclidiana conocida como el módulo del vector que une los dos puntos estudiados.

Pero, en esta ocasión debido al avance que supone la detección de descriptores a través de SIFT con su orientación gradual basada en los puntos vecinos, se emplea una alternativa basada en el histograma de la misma. Entendiendo histograma como aquel gráfico que aúna todas las escalas de grises de la imagen determinando cuantos píxeles cuentan con cada uno de los valores de la escala.

En esta alternativa, en lugar de usar la distancia euclidiana se usará la distancia de Hamming. esta se define como el número de bits que se deberían de cambiar para que dos variables informáticas coincidan. Para valorar la posible veracidad de las coincidencias se usa el filtro de Lowe. Este establece qué puede considerarse una coincidencia entre puntos cuando la distancia de Hamming se encuentra por debajo de un umbral determinado. El filtro se muestra en la figura 5-9.

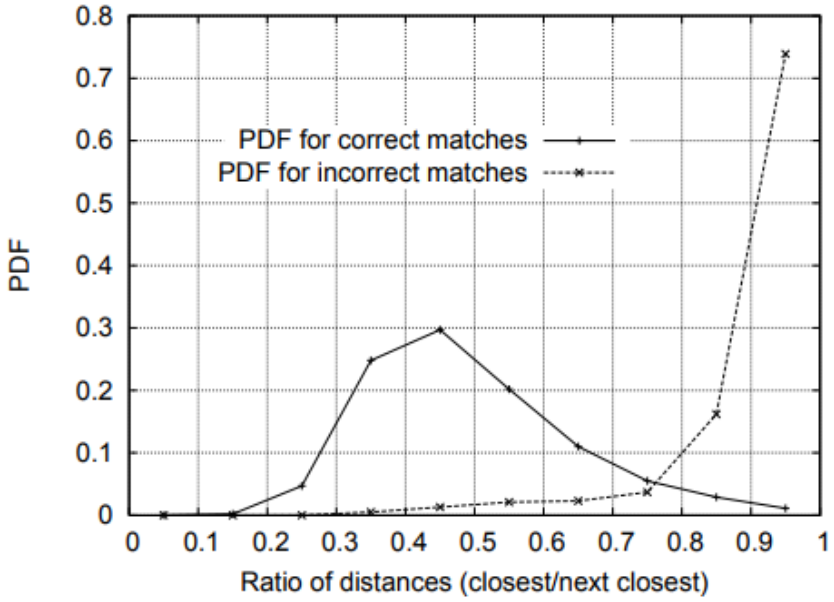


Figura 5-9 Ejemplo filtro de Lowe (Fuente: [13])

Por tanto, la función en el apartado de reconocimiento de imágenes recoge los descriptores extraídos previamente y los almacena en un vector propio, seguidamente crea otro vector en el que almacenara las coincidencias que se encuentren. Posteriormente, se configura la función “KnnMatch” encargada de la comparación, que necesita la definición de los siguientes parámetros:

- “QueryDescriptors” aquí se le pasa el conjunto de descriptores recogidos previamente que van a ser consultados para su comparación. La variable es un vector.
- “TrainDescriptors” aquí se le pasa el conjunto de descriptores que ya se tenían almacenados de la imagen a la que se quiere comparar.
- “matches” en esta ocasión se le pasa un vector donde serán almacenadas las coincidencias halladas.
- “mask” define el número máximo de posibles coincidencias entre los descriptores de la imagen almacenada y aquellos de la imagen que está siendo analizada.

Tras esto la función “KnnMatch” devuelve el vector de coincidencias con todas las que ha sido capaz de encontrar siguiendo el método explicado anteriormente, en una de las variables de cada componente del vector se especificará el grado de similitud que se ha hallado entre ambos puntos. Posteriormente se le va a aplicar el filtro de Lowe anteriormente descrito, para ello se realizarán los siguientes pasos:

- Definición del umbral de funcionamiento del filtro de Lowe “ratio_tresh”.
- Creación de vector de coincidencias correctas definitivo “Good_matches”.
- Recorrido del vector de coincidencias original.
- Comparación del grado de similitud de cada coincidencia con el exigido por el filtro de Lowe.
- Almacenamiento de la coincidencia en el vector de “Good_matches” si supera el filtro.

Una vez superado el filtro contamos con las coincidencias aceptables que pueden pasar a ser examinadas para ver si suponen una cantidad suficiente para hablar de reconocimiento de imagen o por el contrario no es así. Para ello la función termina con los siguientes pasos:

- Establecimiento del número de coincidencias mínimas para considerar que una imagen ha sido reconocida.
- Si el tamaño del vector de coincidencias correctas coincide con el parámetro mínimo recientemente establecido se determina el reconocimiento.
- Si por el contrario no se cumple la restricción, se declara que no hay coincidencias suficientes y se continúa examinando.

Una vez finalizado el proceso, si el resultado ha sido positivo, la función es capaz de rodear la imagen que está siendo sometida a reconocimiento con un color característico. Para ello, se detectan los bordes de la imagen usando de nuevo la detección gaussiana explicada con anterioridad y se cambian los valores de los elementos de estos límites al color deseado para resaltar la detección.

5.5 Actuación en consecuencia de la detección de objetos

Para hacer al dispositivo capaz de reaccionar antes los estímulos creados al detectar objetos debemos crear un nuevo paquete, y dentro del mismo un nuevo nodo, que básicamente se suscriba al topico “/objects” donde se publica la detección de señales. Por otro lado, este nuevo nodo debe publicar en el tópico “cmd_vel” que es el que controla la velocidad del dispositivo.

Como se ha explicado previamente, el nodo no es más que un archivo en c++ (.cpp). En este caso, funcionaremos como siempre, declarando al inicio las bibliotecas necesarias, a continuación, definiremos mediante el ID de cada imagen almacenada en nuestro directorio una señal determinada, del tipo “Giro derecha” o “recto”.

Lo siguiente será diseñar la función de reacción frente a objetos, esta será una switch/case donde dependiendo del identificador reconocido se impondrá una condición u otra. Cabe destacar que se juega básicamente con la velocidad lineal y la angular. Por ello fue necesario probar mediante ensayo error cual era la velocidad adecuada tanto en rectas como en curvas para que con un simple avistamiento de la figura el dispositivo fuera capaz de realizar un giro o avanzar hasta la siguiente indicación. En el caso de no detectar ninguna señal, en este caso se presupone que el vehículo va en un circuito y se le mueve a una velocidad de crucero.

La totalidad de imágenes utilizadas para el desarrollo de la función y del apartado en general se detallan a continuación con su identificador particular. Por último, se desarrolla el código del propio nodo donde se comprueba el estado del sistema general, se suscribe al tópico “/objects” y se publica en el tópico “/cmd_vel”.

Cabe recordar que como siempre tras crear un paquete, nodo o archivo launch es necesario actualizar la “CMakeList” en la que como se ha explicado previamente, declararemos como ejecutable el nodo, aclararemos las bibliotecas indispensables para el desarrollo del mismo y dirigiremos el camino hacia el directorio del nuevo nodo. Además, no puede olvidarse ejecutar la orden “catkin_make” para actualizar el sistema.

En el apartado de resultados se adjuntan una serie de videos donde se muestra el funcionamiento del dispositivo en la vida real utilizando este paquete y en el anexo B se incluye el código de este nodo [2]

5.6 Señales utilizadas para su detección

A continuación, se adjuntan todas las señales que han sido utilizadas para el diseño del proyecto, seguidas del identificador asignado por el sistema y su consecuencia tras la detección.

- Señal de continúe recto: cuenta con el identificador número 3, cuando esta señal es detectada el sistema mantiene su dirección a la velocidad que ya circulaba previamente.

Figura 5-10 Señal Recto-utilizada (Fuente: [14])

- Señal giro derecha: este objeto cuenta con el identificador número 2, como se ha explicado con anterioridad primero se tomó una flecha con orientación a la derecha normal y posteriormente está en la que el giro es más fácilmente reconocible. Cuando se detecta esta imagen la velocidad angular se actualiza con un valor positivo que hace girar al dispositivo hacia la derecha.

Figura 5-11 Señal giro a la derecha utilizada (Fuente: [14])

- Señal giro izquierda: esta imagen cuenta con el identificador número 1, sigue la misma dinámica explicada previamente para el giro a la derecha, cuando se detecta esta imagen la velocidad angular se establece con valor negativo haciendo que el robot gire hacia el sentido deseado.

Figura 5-12 Señal giro izquierda utilizada (Fuente: [14])

- Señal de STOP: esta imagen cuenta con el identificador número 5 en el sistema, su tamaño de letra y el hecho de que existan caracteres dentro de la señal lo supone como una de las más sencillas de detectar, cuando el sistema la detecta tanto la velocidad lineal como la angular se tornan nulas.

Figura 5-13 Señal STOP utilizada (Fuente: [14])

- Señal velocidad 120: esta señal cuenta con el identificador número 4 dentro del sistema. Debido a la cantidad de números con los que cuenta su detección se considera bastante asequible. Cuando el sistema detecta esta señal, la velocidad lineal se torna al máximo de su capacidad, es peligroso ya que muchas veces el exceso de velocidad produce errores en la detección de otras señales. En el sistema se la nombra como "max_vel"

Figura 5-14 Señal 120 utilizada (Fuente: [14])

- Señal de velocidad 50: esta señal cuenta con el identificador número 8 en el sistema tiene las mismas características que la señal de velocidad máxima debido a su similitud en cuanto a características. En el sistema se reconoce como "min_vel" cuando es detectada la velocidad lineal del sistema se torna al mínimo de su capacidad.

Figura 5-15 Señal 50 utilizada (Fuente: [14])

- Señal de velocidad 90: este objeto cuenta con el identificador número 6 en el sistema, sus características son similares a las dos anteriormente descritas, en el dispositivo es conocida como “med_vel”, cuando es detectada el sistema actualiza la velocidad lineal del robot a su capacidad media.

Figura 5-16 Señal 90 utilizada (Fuente: [14])

6 NAVEGACIÓN SLAM

6.1 Introducción

La navegación SLAM (simultaneous localization and mapping), consiste en un método capaz de crear el mapeado de un entorno al mismo tiempo que se determina la posición del robot dentro del mismo. la idea principal es que cada vez que el dispositivo se desplaza, se actualiza su posición con respecto a la justo anterior.

Es por ello por lo que se considera indispensable conocer la localización inicial del robot en el mapa. En nuestro caso no contamos con sistemas de localización global (GPS), por lo que sería recomendable colocar siempre al dispositivo en la misma posición al comenzar. Cosa que con el GPS no sería necesaria.

Si bien la finalidad de realizar un mapeado en nuestro proyecto era la de situar los posibles objetos a detectar con las técnicas anteriormente expuestas, el desarrollo y explicación del mismo finalmente, se ha ejecutado como fundamento de la planificación de trayectorias. Que se ha convertido en uno de los objetivos finalistas de nuestro trabajo. Además, se considera indispensable que, si se quiere lograr el objetivo final en trabajos futuros, se conozca de primera mano el funcionamiento de dicho proceso y los problemas que puede entrañar su desarrollo.

El mapeado de entornos se realizará mediante cuadriculas en las que se dividirá el espacio como aparece en la imagen 6-1 que se irán rellenando a medida que sean recorridas. Para saber la transformación que se realiza de una cuadrícula a la siguiente se utiliza el tópico "tf" del paquete "tf2" de ROS, este nos muestra tanto la traslación como la rotación realizada para el paso de una casilla a la siguiente.

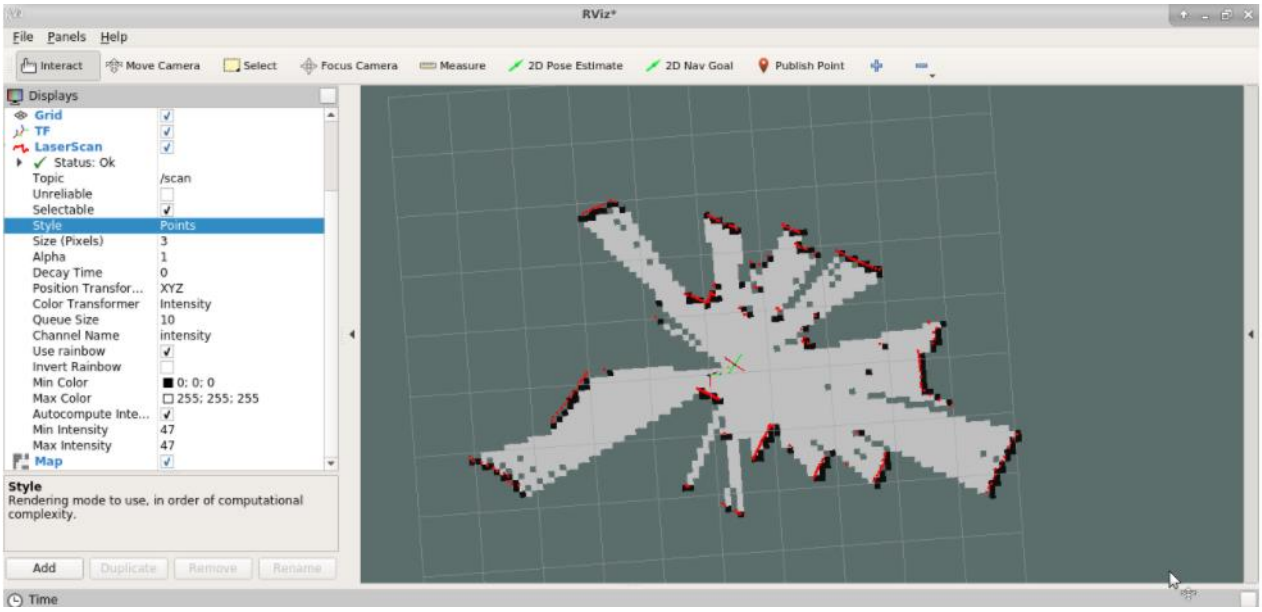


Figura 6-1 Ejemplo mapeado SLAM en Rviz

Para conseguir llevar a cabo la navegación deseada, crearemos un nodo que se suscribirá al tópico "/pose" publicando un mensaje del tipo "/posestamped" donde irá incluida la información de la transformación de los distintos objetos que nos interesan. El nodo creado para controlar todo esto se llamará "drive_controller" y su código se incluye comentado en el anexoB del proyecto [**3**].

Cabe destacar, que en este trabajo utilizamos un sistema de tiempo real que varía su posición en cada momento. Ya que existe también la posibilidad de publicar información de manera estática. Es decir, transformar posiciones que no cambian en el tiempo. Esta por ejemplo es interesante para la posición del láser del robot con respecto a su base. Porque a pesar de que el conjunto se mueve, la diferencia entre uno y otro nunca varía.

Como será habitual casi siempre, la información obtenida podrá ser observada a través de la herramienta “Rviz”. Dónde añadiremos una nueva pantalla o “display” con el tópico “tf”. Además, podremos añadir “pose” lo que nos permitirá observar el desplazamiento del mismo. La configuración del display se muestra en la imagen 6-2.

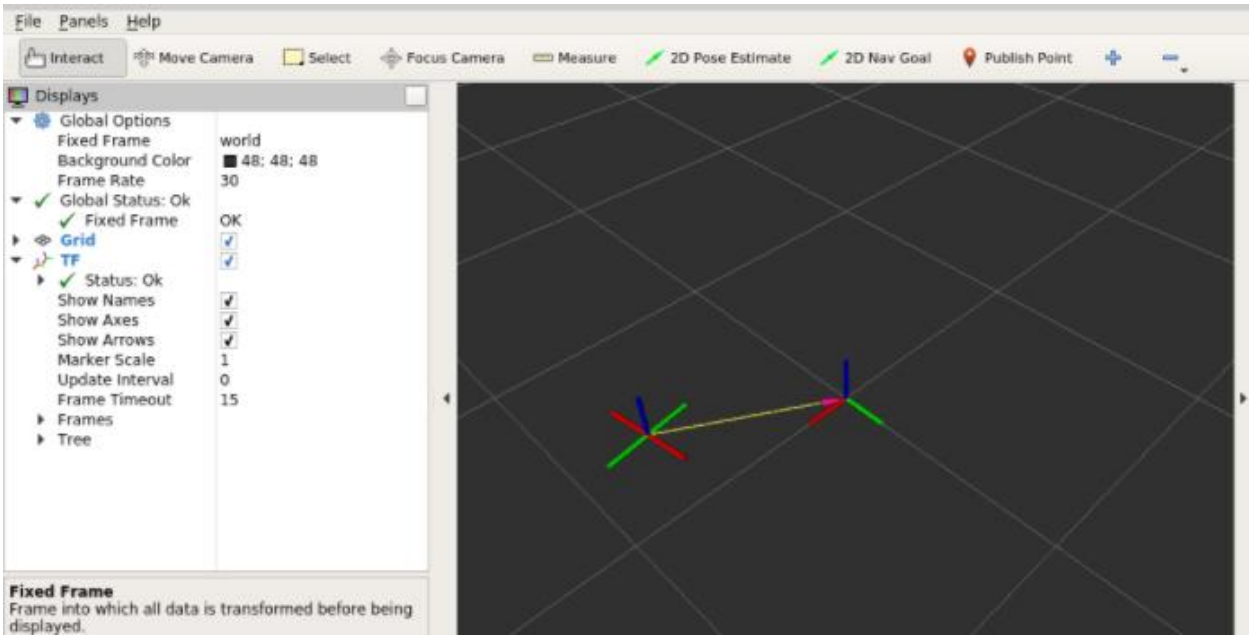


Figura 6-2 Ejemplo Display TF en RViz

6.2 Control remoto del Robot

En este apartado se resumirá el sistema que rige el movimiento de nuestro dispositivo. Este se coordina mayormente mediante comandos del tipo "/Twist". Comandos que son almacenados por el controlador del motor para posteriormente ser ejecutados, en estos comandos básicamente se describen dos campos:

1. "Vector3_Linear" en este campo se especifica la velocidad lineal en [m/s] del dispositivo, por así decirlo describe el movimiento hacia adelante o atrás del robot.

2. "Vector3_angular" en este campo se especifica la velocidad radial del robot. Esta se mide en [rad/s] un valor positivo del mismo produce un giro hacia la derecha, mientras que un valor negativo produciría un desplazamiento a la izquierda.

Una vez comprendido cómo funciona el movimiento de nuestro dispositivo nos disponemos a describir la manera de controlarlo remotamente. Esto se conseguirá mediante una serie de sencillos pasos:

- Conectar de manera remota un portátil o ordenador al dispositivo como se ha explicado previamente. Como es obvio, el control del robot puede hacerse de manera remota mediante un ordenador externo o de manera directa con un teclado conectado al propio robot, ambos son válidos.

- Lanzar el nodo "rosbot_ekf" encargado de utilizar el filtro de Kalman extendido, utilizado para calcular de manera mucho más aproximada la posición y orientación del sistema.

- Ejecutar el nodo "Teleop_twist_keyboard" encargado de coordinar el movimiento del robot mediante el teclado del dispositivo ejecutor del mismo. Este se encarga de crear los mensajes al pulsar cada tecla y enviarlo al motor correspondiente.

- Ejecutar el nodo "Teleop_twist_keyboard.py" que convierte los mensajes creados por el nodo anterior a aquellos más funcionales que pueden interpretar los controladores de cada motor.

Al cumplir todos estos pasos podremos al fin controlar el sistema de manera remota mediante el teclado de nuestro dispositivo conectado al robot. Los comandos que coordinan el movimiento se describen a continuación:

- "i" para desplazar al dispositivo hacia adelante.

- "," para desplazar el robot hacia detrás.

- "j" para girar el sistema hacia la izquierda.

- "l" para girar el dispositivo hacia la derecha.

- "k" para parar el robot.

- "q" para aumentar la velocidad del sistema.

- "z" para disminuir la velocidad del dispositivo.

Para terminar con el apartado se aclara que es posible configurar diferentes dispositivos de control remoto para trabajar con el robot, esto queda fuera del alcance del proyecto y por tanto no se desarrollará, pero queda constancia de la posibilidad de hacerlo.

6.3 Implementación en ROS de la navegación SLAM

En nuestro caso la manera más sencilla de implementar este tipo de navegación en ROS a través de nuestro robot es mediante el uso del escáner laser con el que cuenta el dispositivo. Es por ello por lo que utilizaremos el sensor LIDAR integrado dentro de nuestro sistema que junto con los sistemas de odometría con los que cuenta nos permitirán realizar un mapeado del entorno bastante correcto.

Debido a esto, nuestro robot utilizará el "RPlidarNode" encargado de comunicar el sistema con el sensor para permitirle publicar lo escaneado en el tópico "/scan" con un mensaje del tipo "/LaserScan". Esto será lo que recogerán los sistemas de mapeado para rellenar casilla a casilla el entorno del dispositivo y esbozar un mapa con los posibles obstáculos de alrededor.

Para comenzar a realizar el propio mapeado deberemos iniciar una serie de nodos particulares que se describen a continuación:

1. El "RPlidarNode" indispensable para que el sensor comience a escanear el entorno y publique los datos con el objetivo de que sean transformados en el posterior mapa.

2. El "drive_controller" creado por nosotros mismos anteriormente, encargado de publicar las transformaciones (traslación y rotación) realizadas por el dispositivo para avanzar de una casilla a la siguiente del mapa cuadriculado. Este nodo en particular se explica en el anexo posterior de manera detallada [**3**].

3. El "teleop_twist_keyboard" que nos va a permitir controlar el dispositivo de manera remota mediante el teclado, como se ha explicado anteriormente.

4. El "Rviz" como herramienta de visualización de todos los datos recopilados por el sensor LIDAR.

5. El "gmapping" paquete de ros que trabaja mediante el sensor laser incluido dentro del LIDAR basado en la navegación SLAM. Tal y como nosotros buscamos donde la localización y el mapeado se realizan de manera simultánea.

Cuando nos aparezca la interfaz de la herramienta Rviz debemos configurarla correctamente, para ello añadiremos nuevos objetos o pantallas, seleccionaremos "add object" y en la nueva ventana escogeremos "by_topic" donde escogeremos "/scan" y dentro del mismo los distintos marcos de encasillado del mapa los cambiaremos a "laser_frame" esto nos devolverá lo que detecta el láser a través del LIDAR del entorno que rodea al dispositivo.

Si queremos comenzar a explorar lo que se encuentra alrededor del robot, cambiaremos los "frames" del mapa de "laser_frame" a "odom", añadiremos el objeto "TF" que nos indica la posición del robot en cada momento, y comenzaremos a desplazarnos mediante el teclado para detectar los obstáculos que nos rodean.

Para ser capaces de versar toda esta información en un mapa usaremos el nodo "slam_gmapping". Este se suscribe al tópico "/tf" que consigue la posición del robot en cada momento. También se suscribe al tópico "/scan" para recabar la información vertida por el LIDAR. El nodo publica todo esto en el tópico "/map" donde se guarda la información relativa al mapeado.

En Rviz debemos añadir el objeto "/map" de la carpeta de tópicos, en un principio este será un mapa por defecto provisto por el sistema, pero a medida que avance nuestro dispositivo será capaz de modelar un entorno cercano al que de verdad lo rodea.

A continuación, se dejan una serie de imágenes como ejemplo de cómo el dispositivo ha mapeado una casa. Estas son la figura 6-3 y 6-4.

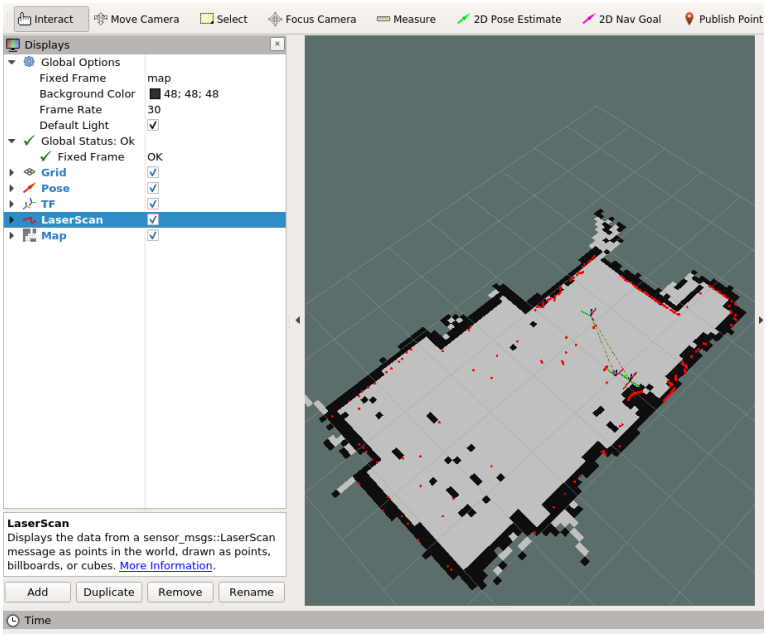


Figura 6-3 ejemplo mapeado salón

En esta primera figura se observa el mapeado realizado al salón de mi casa. En ella puede observarse como el robot es capaz de detectar los sofás como obstáculos que suponen límites infranqueables, además de las patas de los muebles en altura, ya sean sillas o mesas o elementos por el estilo que suponen las marcas negras de la derecha de la imagen. En rojo podemos observar los límites que provee el escáner láser de los obstáculos que detecta en ese momento.

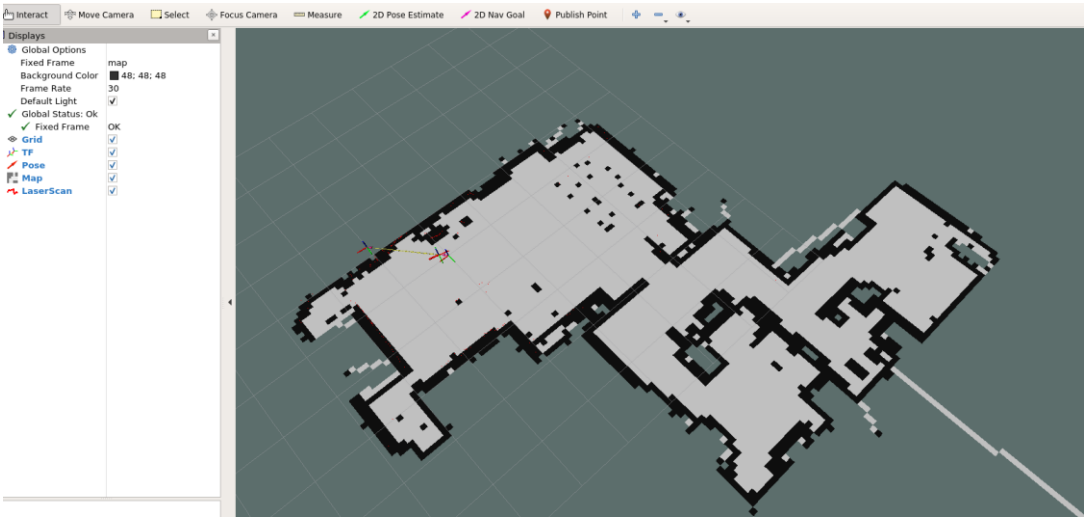


Figura 6-4 ejemplo mapeado casa

En esta segunda figura podemos apreciar un entorno algo más amplio de mi hogar donde queda recogido el salón en la parte superior, la cocina, un aseo y una habitación en la parte inferior. En este caso es destacable el hecho de la capacidad de mapeado de grandes superficies que lleva a cabo el dispositivo con buenos resultados.

Detecta con un mayor acierto los obstáculos que suponen las patas de las sillas, y de la mesa. Detecta correctamente también las puertas de las habitaciones abiertas, y los obstáculos referidos a muebles de cada estancia. Comete ciertos errores como se aprecia en la línea que se pierde en el infinito, pero pueden considerarse aislados, ya que en líneas generales el resultado es muy correcto.

Cuando consigamos el resultado deseado en nuestro mapeado será indispensable guardar los resultados, con la finalidad de poder ser reutilizados en el futuro sin necesidad de volver a mapear el entorno al completo. Para ello nos serviremos del nodo "map_saver", cabe destacar que sin él cuando el nodo deje de funcionar perderemos toda la información obtenida y deberemos empezar de nuevo.

Lo primero será crear un nuevo directorio en nuestro paquete que llamaremos "maps" y cuando en Rviz el mapa sea el deseado nos cambiaremos a dicha carpeta para que los avances se guarden ahí. Entonces ejecutaremos el nodo de la siguiente forma "rosrun map_server map_saver -f nombre mapa" y el mapa en cuestión quedará guardado.

En ese momento, se crearán dos archivos, uno del tipo "yaml" y otro del tipo "pgm" el de tipo yaml contiene toda la información del mapa, su resolución, la posición de origen, la zona ocupada y la libre. El de tipo pgm es una imagen del mapa recogido en escala de grises.

En nuestro caso el archivo pgm generado, luego de ser abierto por una aplicación particular llamada GIMP que gestiona este tipo de imágenes, se muestra como aparece en la figura 6-5. Puede considerarse el resultado muy parecido al original previamente mostrado.

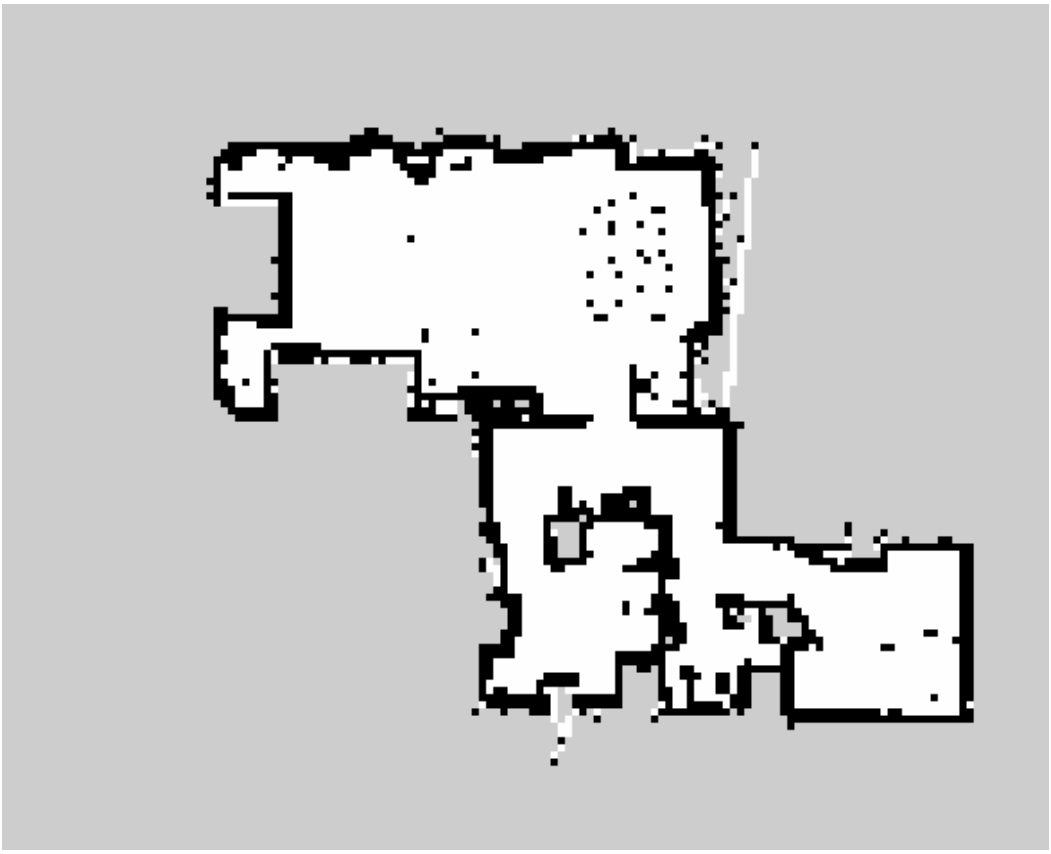


Figura 6-5 Ejemplo mapa.pgm

En el mapa se aprecia de nuevo en la parte superior el salón de la casa, seguido de un pasillo del hall de entrada, la cocina a la izquierda y un aseo junto con una habitación a la derecha.

6.4 Planificación de trayectorias

La planificación de trayectorias consiste en la determinación de la secuencia de movimientos necesaria para trasladar al dispositivo de una posición inicial a otra final o destino evitando la colisión con los obstáculos encontrados en el camino.

El método de planificación utilizado es una mezcla entre el método gráfico y el de celdas ocupadas. El método gráfico sirve para definir los límites de la estancia, además de determinar las posibles puertas abiertas, que conectan habitaciones o pasillos. En adición, cada límite puede tener asignada su anchura para determinar la dificultad de atravesarlo en un futuro. Encontrar la trayectoria se fundamenta en determinar el camino más eficiente posible de conexión entre las posiciones inicial y final.

Por otro lado, el método de celdas empleado divide el área en casillas a modo de pixeles. A estas se les asigna la categoría de ocupadas o libres. Además, una de las celdas es marcada como el origen y otra como el destino final. Por tanto, en este caso encontrar la trayectoria se fundamenta en trazar la línea más corta posible que conecte las dos celdas de origen y destino sin cruzar por ninguna que esté ocupada.

Es indispensable para la realización de este apartado la configuración del nodo "move_base" este es el encargado de crear el llamado "costmap" que determina un valor a cada celda del mapa dependiendo de lo cerca o lejos que se encuentre del obstáculo. Mediante el mismo somos capaces de crear la trayectoria con menor "coste" es decir aquella más corta y con menos posibilidades de chocar con un obstáculo.

6.4.1 Configuración previa del nodo "Path_Planning"

Para determinar el "costmap" descrito previamente debemos configurar una serie de parámetros que se describen a continuación:

- Sensor_frame: coordina el marco de acción relacionado con el sensor.
- Data_type: determina el tipo de mensaje enviado por el sensor.
- Marking: se activa cuando una celda se marca como ocupada.
- Clearing: se activa cuando una celda se marca como libre.
- Obstacle_range: determina el rango en el que se comienzan a incluir obstáculos en el costmap gracias al sensor.

De la misma manera que se ha configurado el "costmap" deberíamos ajustar también el "local_costmap" y el "global_costmap" con parámetros muy similares a los descritos anteriormente. En el apartado de la configuración de la propia planificación de trayectorias cabe destacar una serie de parámetros como la velocidad máxima o mínima de desplazamiento tanto lineal como radial, la diferencia entre su posición y la posición de destino deseada que es aceptable o la aceleración máxima del conjunto.

Todo esto se ha automatizado mediante un archivo launch que se incluye en el anexo donde todos los parámetros necesarios para el correcto desarrollo del sistema se ajustan instantáneamente al ejecutar el archivo en cuestión.

6.4.2 Lanzamiento del nodo “Path_Planning”

Una vez configurado el sistema, podremos ser capaces de ejecutarlo y comprobar su funcionamiento. Para ello deben seguirse los siguientes pasos, donde se determinan los nodos a lanzar:

1. Ejecutar el nodo “move_base” descrito previamente.
2. Lanzar el nodo “teleop_twist_keyboard” para el control mediante el teclado del dispositivo.
3. Activar el nodo que desarrolla el filtro de Kalman extendido “rosbot_ekf”.
4. Ejecutar el nodo “tf” encargado de transformar la posición real del robot en la que puede ser leída por el sistema.
5. Lanzar el nodo “slam_gmapping” encargado de generar un mapa del entorno que rodea al robot.
6. Activar el programa Rviz que nos permite visualizar todo lo explicado de manera más gráfica.

Una vez abierto Rviz como se ha explicado antes deberemos configurar las distintas “displays” o pantallas que nos serán útiles para la visualización del proyecto, los elementos son los mismos empleados en ocasiones anteriores salvo que, se añadirán las pantallas “path” del fichero “move_base/TrajectoryPlanerROS” tanto a nivel global como a nivel local.

Además, será necesario añadir la pantalla “Polygon” del “global_costmap” y los propios “costmaps” global y local definidos previamente. Tras un ajuste del color de visualización de ambos podremos apreciar nuestro entorno con los obstáculos que nos rodean de una manera llamativa como se muestra en la figura 6-6.

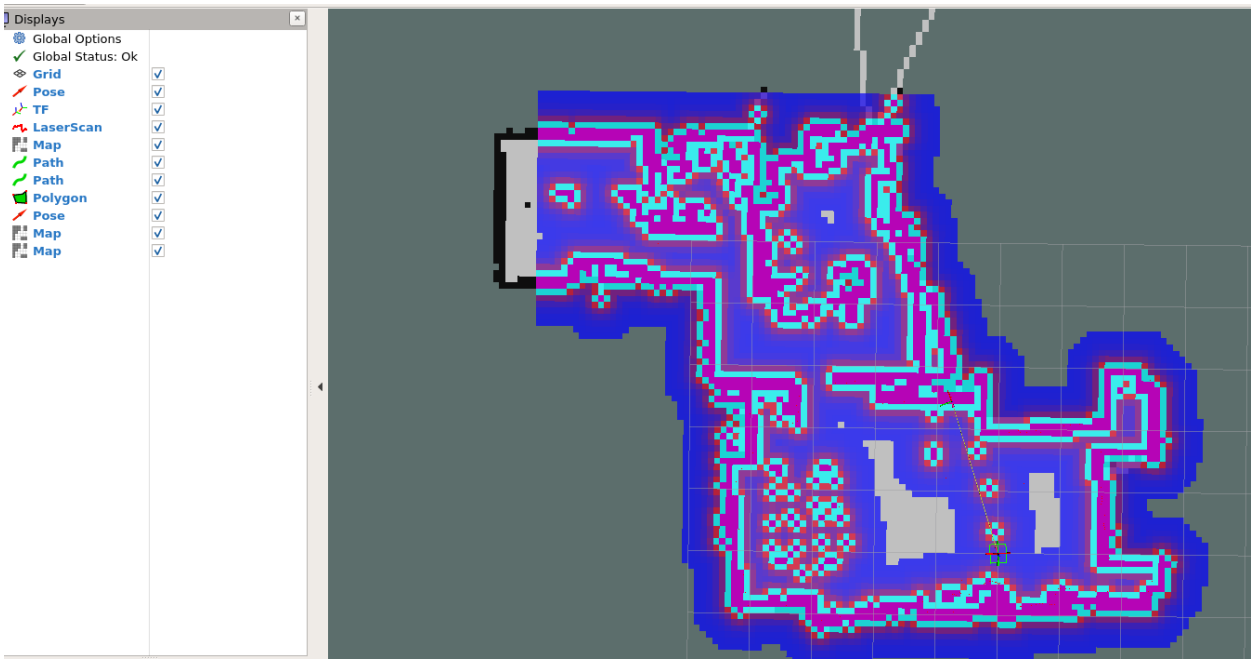


Figura 6-6 ejemplo mapeado planificación de trayectorias

Como puede observarse el entorno es el mismo que el previamente descrito en la navegación SLAM. Con la diferencia de que en este apartado con la finalidad de la planificación de trayectorias, el sistema es capaz de detectar de manera más exacta los obstáculos que lo rodean con el fin de no chocar con ellos en la trayectoria.

6.4.3 Funcionamiento de la planificación de trayectorias

Posteriormente, mediante una de las posibilidades que nos brinda el programa Rviz seremos capaces por fin de determinar un punto de destino final y que el sistema programe una trayectoria siguiendo los criterios previamente descritos, mínimo coste posible, mínima distancia y mayor evitación de obstáculos.

En la figura 6-7 puede apreciarse en la parte superior la opción de navegación "2D Nav Goal" que nos da la posibilidad de seleccionar un punto de destino determinado dentro del mapa previamente elaborado. Una vez hecho esto el robot automáticamente detecta la trayectoria más correcta y comienza a ejecutarla.

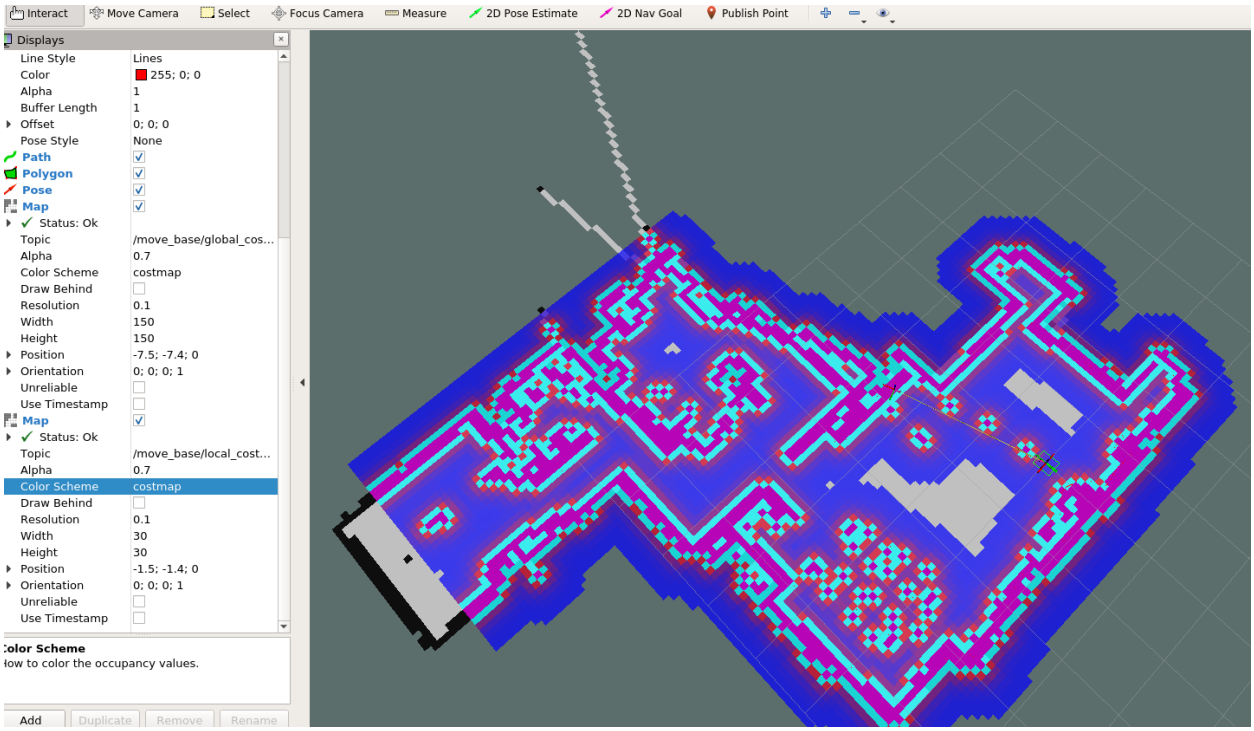


Figura 6-7 ejemplo mapeado planificación de trayectorias

La trayectoria quedará definida mediante una serie de líneas discontinuas de color amarillo. Cabe destacar que es muy importante la posición inicial y actual del robot en cada momento. Ya que las trayectorias son calculadas con respecto a ellas y si difieren de la realidad es posible que colisione contra obstáculos debido a la falta de exactitud. Llegados a este punto es mejor comenzar de nuevo y colocar al robot en la posición de partida original.

6.4.4 Fundamentos de la planificación de trayectorias en ROS

Particularmente en nuestro entorno de ROS podemos explicar de manera más detallada el funcionamiento de cada nodo que interviene en el desarrollo de la planificación de trayectorias. Como se ha explicado previamente, el nodo principal es el "move_base" que proviene del paquete "navigation" este es bastante sencillo conceptualmente, ya que toma información de la odometría y los sensores que la emiten, para ajustar los comandos de velocidad y movimiento enviados al robot.

En particular el nodo "move_base" de este paquete busca como se ha descrito previamente que, dada una posición final de destino, esta sea alcanzada por la base móvil. Su funcionamiento se aclara en la figura 6-8 incluida a continuación, en ella puede observarse como la información que llega desde la odometría y los distintos sensores, son transformados mediante el nodo "tf" descrito con anterioridad, o el nodo "odom", que transforma la información odométrica en otra utilizable por el dispositivo.

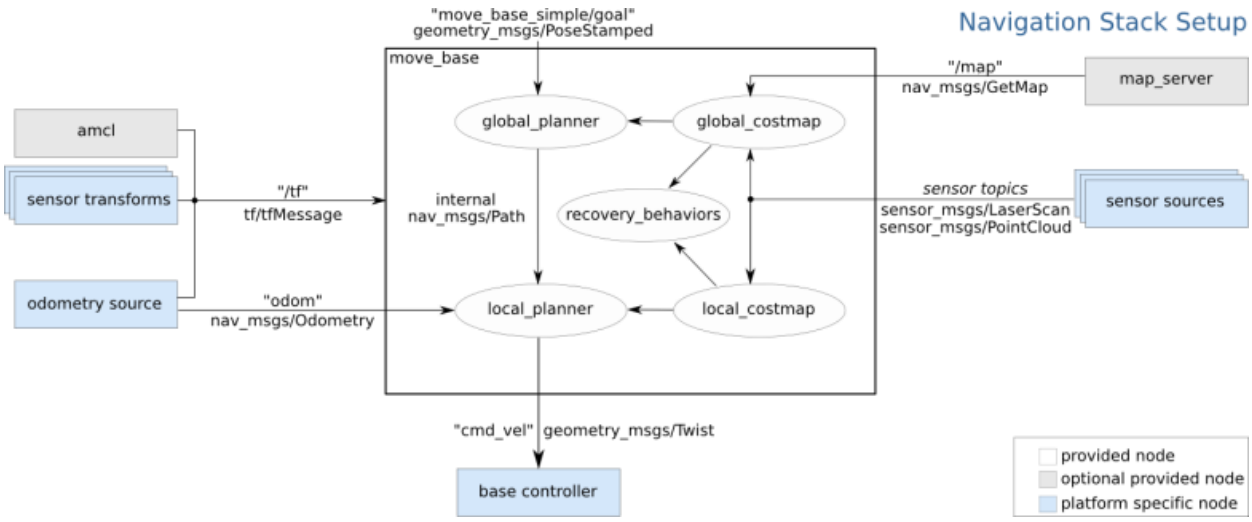


Figura 6-8 esquema funcionamiento nodo “move_base” (Fuente: [11])

Por otro lado, el nodo encargado de la creación del mapa y otra serie de sensores, elabora los distintos mapas de coste explicados antes y que posteriormente serán analizados con mayor profundidad. Con toda esta información el nodo “Path_planning” envía la información al nodo “nav_core” que utiliza un planificador global en el que se almacena la posición de destino deseada, y la trayectoria que debe seguirse. Este planificador utiliza el algoritmo del tipo “navfn” que realiza una rápida interpolación de la función de navegación y crea la trayectoria a seguir por la base.

Sin embargo, antes de transmitírsela al controlador de la base entra en juego un planificador local basado en el algoritmo “Base_Local_Planner”. Este, mediante un contraste de información con el mapa de coste local que le provee la posición de los obstáculos que se encuentran más cerca, es capaz de tomar una decisión sobre el siguiente movimiento que debe realizarse. Para ello toma información del planificador global y la compara con la situación de los obstáculos que le rodean para seguir encaminando el móvil a la posición de destino deseada, enviando los comandos de velocidad que debe seguir la base.

Su funcionamiento se basa en la conexión de la trayectoria definida por el planificador global con la base móvil del robot. Se considera un controlador capaz de elaborar una función de coste con los obstáculos que rodean a la base, para seguidamente escoger el trayecto que menor coste suponga atravesar.

En la práctica el funcionamiento que se aprecia de nuestro dispositivo se repite de manera secuencial cada vez que debe tomar una decisión. Primeramente, todos los obstáculos que se encuentren fuera de una región específica de interés serán eliminados. Posteriormente, el dispositivo realizará una rotación completa sobre sí mismo con el fin de examinar el espacio que se encuentra a su alrededor.

Si tras estos dos procesos, el robot continúa estancado y no obtiene una salida, se realizará una limpieza mucho más agresiva del entorno donde eliminará todos los obstáculos que no sean aquellos que se encuentran en el espacio que abarca realizando un giro sobre sí mismo. Este proceso se realizará por dos veces, una vez hecho si no encuentra ninguna salida, el objetivo se considera inalcanzable.

En caso contrario, el dispositivo irá avanzando, esquivando obstáculos y será capaz de alcanzar la posición deseada con la trayectoria más eficiente posible.

Tras esto, procedemos a la definición de manera específica del nodo “costmap_2d” que es el encargado de crear los mapas de obstáculos a nivel local y global como se ha definido previamente. En resumen, este nodo se encarga de tomar información de los sensores, en nuestro caso el LIDAR para luego volcarlo en un mapa en 2 dimensiones donde rellena celdas según se detecte un obstáculo o no.

El nodo “costmap_2d” también cuenta con la opción de inicializar un mapa de coste (obstáculos) ya creado previamente y almacenado en un servidor local. La utilidad de este mapa que se crea o se inicializa es proveer una estructura configurable que mantiene la información de hacia dónde puede navegar el robot. Este nodo en particular trabaja en un entorno considerado plano. Por tanto, dos obstáculos en el mismo punto, pero a diferente altura se apreciarán como uno solo ubicado en el mismo lugar.

En cuanto al funcionamiento que sigue el nodo para la configuración del mapa, este utiliza un sistema mediante capas, en la práctica nos encontramos una primera capa que es el mapa estático donde se almacenan los límites del entorno. En una segunda capa, esta en color azul se identificará la zona transitable por el dispositivo. Para un correcto funcionamiento el robot nunca debe abandonar esta zona, ya que si lo hace estará navegando en un entorno desconocido.

Esta segunda capa ya descrita, como el resto, se va actualizando a medida que el robot explora, y avanza por el entorno. En una tercera capa, esta roja, se encuentran los propios obstáculos a esquivar, por tanto, estas zonas nunca deben ser transitadas por el dispositivo si se desea evitar una colisión. Cada una de estas tres capas funcionan de manera individual, compilan por separado y se actualizan individualmente.

El nodo “costmap” se suscribe automáticamente a aquellos tópicos de ROS encargados de proveer la información dada por los sensores, ya sea el LIDAR, o la cámara. Encargándose de marcar u ocupar cada una de las celdas de nuestro mapa, para ello recorre una matriz donde actualiza el valor de cada elemento según la información detectada.

Si el sensor muestra la detección de un obstáculo, entonces se actualiza el coste del elemento de la matriz que representa la celda estudiada y posteriormente en la práctica se colorea como obstáculo en el mapa visible. De igual manera, también es capaz de liberar un elemento de la matriz si allí donde se apreciaba un obstáculo ahora no se encuentra nada mediante el mismo proceso.

Cada celda de nuestro mapa cuenta con un valor en la matriz que va desde 0 a 255 asignado según el método de inflación. Dónde un valor por encima de 252 supone una colisión definitiva, otro entre 127 y 252 describe una posible colisión (aún salvable) dependiendo de la orientación, y por último un coste entre 0 y 127 define que no hay colisión posible además de establecer el rango en el que la mayor parte de preferencias deben ser expuestas. La gráfica que describe la asignación de valores a cada elemento se muestra a continuación en la figura 6-9. En ella se relaciona la distancia entre el obstáculo y el robot con el valor asignado a cada celda del mapa.

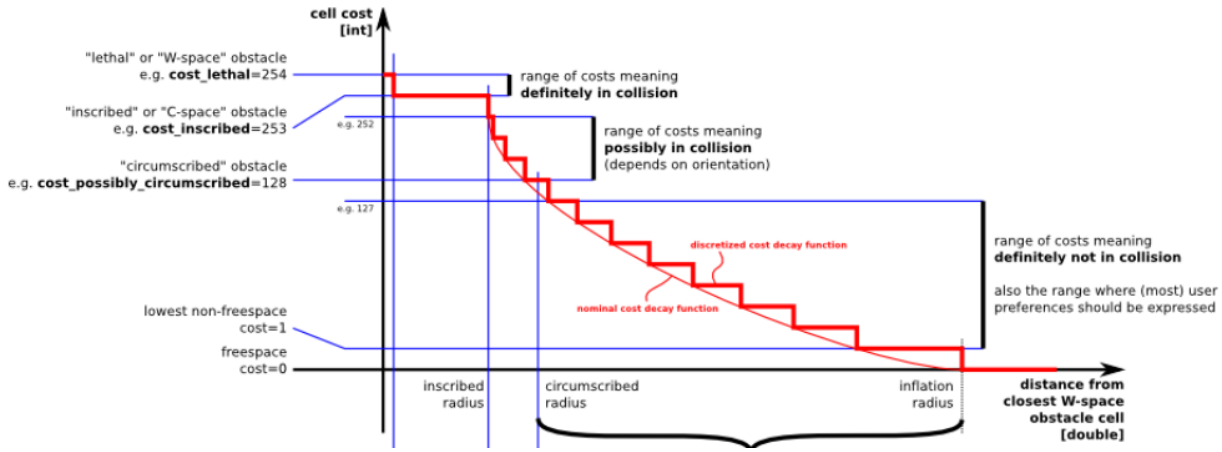


Figura 6-9 gráfica de asignación de valores para cada elemento de la matriz (Fuente: [11])

En la práctica, para el mapeado mediante capas, estos valores pueden resumirse en solo 3, que se irán asignando a cada celda del mapa para que este se vaya completando, estos son el valor ocupado, libre o desconocido. Y se rellenará cada celda mediante el color rojo (ocupado), azul (libre) o gris (desconocido).

La actualización de nuestro mapa se lleva a cabo mediante ciclos que se realizan con una frecuencia definida como parámetro por nosotros mismos mediante la variable "update_frequency". Cada vez que se realiza un ciclo, se efectúa la operación de marcado o limpieza de celdas para posteriormente plasmar esta información en el mapa de interés.

Para el correcto funcionamiento de nuestro nodo, se considera indispensable el uso del nodo "tf", este como se ha descrito previamente, hará las transformaciones necesarias para mostrarnos la posición actualizada del dispositivo dentro del entorno. Si deja de funcionar el sistema, se considera estancado y termina por parar.

Previamente se ha descrito mediante la gráfica de la figura 6-9 el procedimiento que asigna valores a cada celda de la matriz mediante la inflación, este proceso tiene como objetivo relacionar los valores de coste de cada celda con la distancia que las separa del robot, catalogando cada uno en 4 intervalos diferentes:

- "Lethal" este valor muestra que hay un obstáculo dentro de la celda, por lo tanto, si el robot se encuentra en esta celda habrá habido una colisión.
- "Inscribed" este valor indica que el obstáculo se encuentra a una distancia inferior que el radio de funcionamiento del robot, por tanto, dependiendo de la orientación se podría considerar una colisión o no.
- "Freespace" el coste en este caso es asumible como cero. Y significa que no hay ningún obstáculo del que preocuparse en la propia celda estudiada.
- "Unknown" este valor significa que no hay información sobre la celda estudiada. Deberá explorarse pertinentemente para ser rellenado.

Tras esto el nodo que configura y controla la planificación de trayectorias queda completamente definido, en el apartado de resultados se mostrarán una serie de videos con su funcionamiento en la vida real.

6.5 Mapeo de entornos con cámara

En este apartado definiremos el funcionamiento de nuestro sistema de mapeado en la herramienta de Rviz con un nuevo sensor de mapeo, la cámara. El modelo de nuestra herramienta es la Orbbec Astra RGBD camera, que como se ha definido previamente cuenta con un rango de visión de entre 0.6 y 8m, una frecuencia de 30 fps además de incluir 2 micrófonos.

El funcionamiento de la herramienta Rviz será muy similar al empleado con anterioridad. De hecho, los programas que utilizaremos serán los descritos previamente, deberemos lanzar el archivo "maping.launch" junto con el "visión.launch" en modo reconocimiento. Esto nos permitirá iniciar el programa Rviz con la configuración deseada para el mapeado. Además, nos iniciará los nodos relativos a la cámara que nos hacen falta para su inclusión en el posterior mapeado.

En este momento al intentar añadir nuevas "displays" a nuestra herramienta contaremos con nuevas opciones en el apartado de "by_topic". Ahora nos aparecerán una serie de tópicos relacionados con la cámara, hay que tener claro que nuestra cámara está publicando en el tópico "image_raw" del paquete "rgb_camera" por tanto deberemos buscar esa carpeta dentro de la ventana y seleccionar la pantalla "camera". En este momento dentro del programa contaremos en la esquina inferior izquierda con una ventana en la que se apreciará la vista de la cámara en cada momento.

Además, en la parte inferior se añadirá un nuevo apartado en el que se muestra la distancia al obstáculo más cercano detectado por la cámara. En segundo lugar, se añadirá otro "display" en el apartado de "by_topic" donde se seleccionará "Depth_Cloud". Este elemento nos permitirá apreciar el entorno apercibido por la cámara en forma de nube de puntos, ofreciendo así un mapeado en 3 dimensiones del entorno que rodea al dispositivo.

Su funcionamiento es muy simple, toma una nube de puntos generada por la cámara al percibir su entorno y es capaz de elaborar un modelo en 3 dimensiones de la imagen. Esta nube de puntos no es más que un mensaje considerado del tipo "PointCloud.msg" que guarda dentro una serie de puntos con 3 coordenadas más alguna información adicional opcional como es la orientación o la ubicación dentro de la figura en general.

El mensaje descrito con anterioridad donde se almacena la nube de puntos es creado y gestionado por el paquete "pcl" o "point_cloud_library" que contiene numerosos algoritmos destinados a la filtración, estimación de tamaño, reconstrucción de superficies y modelado de entornos. Básicamente es una librería abierta que trata el procesamiento de puntos en 3 dimensiones.

Para la generación del mapa que cuenta con obstáculos en 3 dimensiones, se emplea el siguiente procedimiento descrito en el paquete "pcl" del que se ha hablado previamente.

- En primer lugar, se realiza una filtración de la imagen en tiempo real. Debido a fallos en la medición hay numerosos "shadow points" dentro de nuestros datos, estos complican mucho la estimación local de las características de cada punto dentro de las 3 dimensiones. Para su filtración, se realiza un análisis de los puntos vecinos al que se estudia y se realiza un cribado eliminando aquellos que no cumplen ciertas características. Esta comprobación se basa en una media de la distancia de los puntos vecinos al estudiado, media que si no se cumple se cataloga como erróneo y conduce a su eliminación.

- En segundo lugar, se procede con la caracterización estimada de la nube de puntos. Esta se fundamenta en la determinación de los patrones geométricos de un punto basándose en la información disponible alrededor de cada uno de ellos. Uno de los más importantes es la determinación de la curvatura de los puntos, esta se fundamenta en el análisis del vector característico de los puntos que se encuentran en las capas más cercanas al que se estudia para realizar una estimación de la curvatura que este podría tener.

- En tercer lugar, se realiza la detección de puntos clave, proceso descrito con anterioridad en el apartado de visión y que en este caso no se va a analizar por funcionar de manera muy similar a lo ya explicado.

Una vez realizado este proceso, como se ha descrito anteriormente, el "display" "Depth_Cloud" empleando toda la información creada por el paquete "pcl" realiza un modelo de los obstáculos en 3 dimensiones. Además, es capaz de ubicarlos dentro del mapa de obstáculos y como se ha explicado previamente determinar la distancia al más cercano.

A continuación, se añade la figura 6-10 donde se percibe lo visto por la cámara en tiempo real y su representación posterior en el mapa que se muestra en la figura 6-11.

Figura 6-10 visión de la cámara en un momento preciso

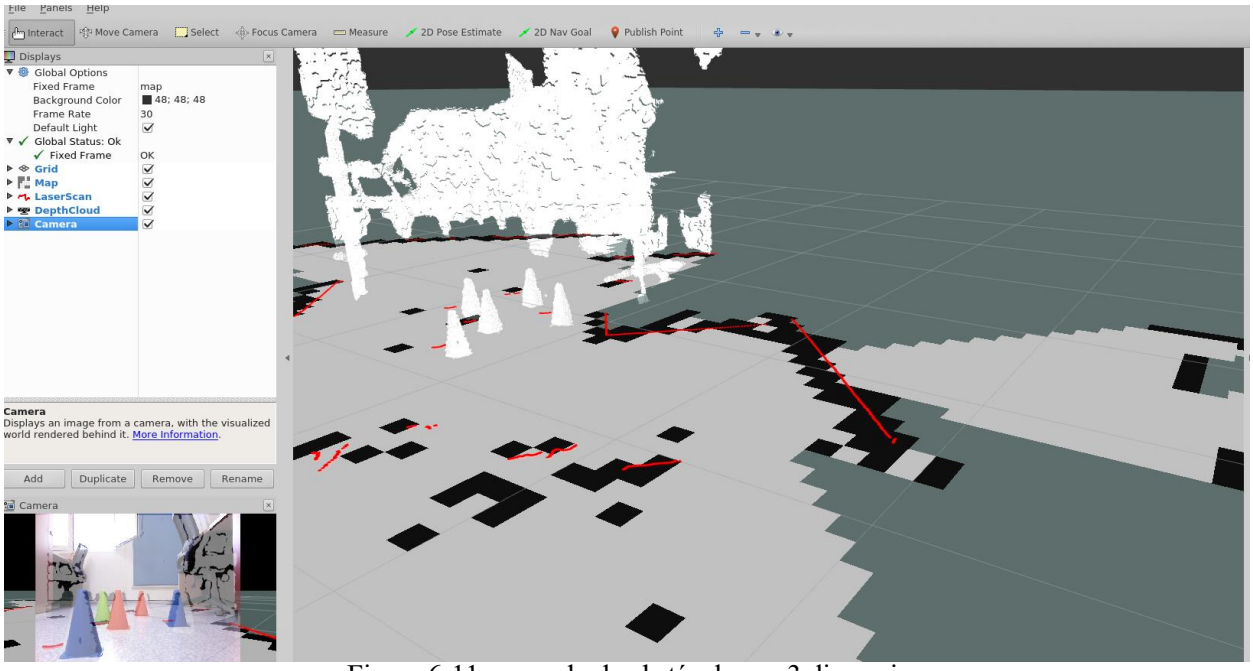


Figura 6-11 mapeado de obstáculos en 3 dimensiones.

6.6 Mapeo de un entorno desconocido

Dentro de la navegación SLAM se ofrece la posibilidad de realizar una navegación de manera autónoma del entorno. Nuestro dispositivo cuenta con un nodo ya preinstalado que es capaz de detectar los obstáculos antes de chocar con ellos y de esta manera ser capaz de esquivarlos y continuar con el mapeo de la superficie.

Este se basa en una distribución por celdas del espacio que rodea al robot, aquellas de las que no se tiene información se catalogan como desconocidas, mientras que las que se van detectando mediante el LIDAR comienzan a rellenarse con la información entregada.

La exploración del entorno se repite de manera indefinida rellenando celdas desconocidas con la información recibida hasta que no queda ninguna. El nodo encargado de la realización de esta acción es el "explore_lite" del paquete "explore" este se dedica a escribir la información detectada sobre el mapa propuesto por el nodo "slam_mapping" y publica la posición de destino deseada en el tópico "/move_base/goal". Su funcionamiento en Rviz se realiza de manera similar a lo que aparece en la figura 6-12.

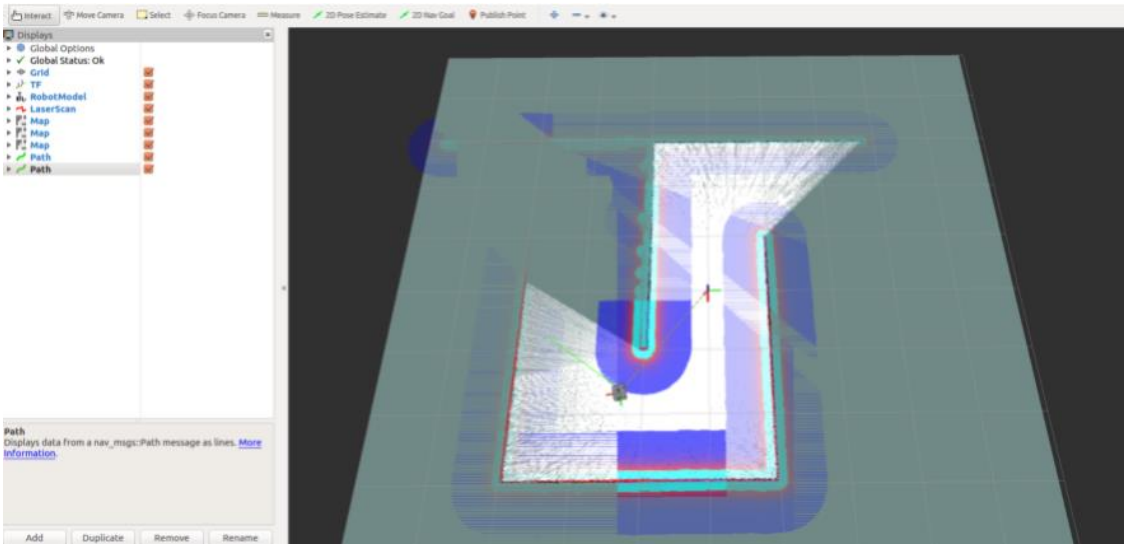


Figura 6-12 ejemplo mapeado entorno desconocido (Fuente: [11])

La configuración previa de esta forma de exploración consiste en suscribir al robot al tópico anteriormente explicado y publicar en el mismo mensajes donde se establezca la posición de destino deseada. Por otro lado, deberá publicar en el tópico "/map" la información sobre las celdas ocupadas que detecte.

A continuación, se enumeran una serie de parámetros con los que el robot trabaja y que previamente deben ser establecidos para un correcto funcionamiento del nodo en cuestión:

- "robot_base_frame" esta se establecerá de igual manera que la base de nuestro robot está establecida, con el fin de que sea la base del movimiento que se estudia.
- "costamap_topic" tópico específico en el que publicar los mensajes de celda ocupada dentro del mapa.
- "visualize" parámetro encargado de determinar si se muestran o no las fronteras detectadas.
- "planner_frequenzy" frecuencia con la que las nuevas fronteras son actualizadas y la posición de destino es corregida.
- "progress_timeout" tiempo específico en el que el robot considerará que el reconocimiento ha terminado si no es capaz de avanzar en su proceso de detección.

- "potential_scale" este parámetro determina la distancia a partir de la cual el dispositivo es capaz de detectar una frontera.

Todos estos parámetros serán configurados en un archivo yaml que se adjunta en el anexo del final del trabajo donde se explica de nuevo línea a línea. Una vez realizada la configuración previa ya podemos seguir los siguientes pasos para comenzar a trabajar con el nodo de exploración de espacios desconocidos.

1. Lanzar el nodo "rosbot_ekf" ya explicado previamente.
2. Ejecutar el nodo "rpLidar" también explicado con anterioridad.
3. Activar el transformador de la posición estático del paquete "tf" anteriormente descrito.
4. Lanzar el nodo "slam_mapping" para crear un mapa vacío en el que poder comenzar a introducir la información.
5. Ejecutar el nodo "explore_lite" que se encargará del movimiento dentro del entorno desconocido y versará la información sobre fronteras detectadas en el mapa creado antes.
6. Activar la herramienta "Rviz" como visualizadora de la información recogida.

Debido a la enorme cantidad de acciones a realizar se incluye todo lo anteriormente descrito en un archivo launch llamado "exploration" que se añade al final del trabajo dentro del anexo donde se automatiza todo este proceso.

Para el correcto desarrollo del paquete, su funcionamiento se basa en el uso del nodo "costmap" descrito con anterioridad, que se utiliza para rellenar el mapa con los obstáculos encontrados, y en el uso del nodo "navfn" que provee una rápida interpolación de una función de navegación que planifica trayectorias para la base del robot. Este nodo opera suponiendo un robot circular, que a efectos prácticos no nos supone un gran problema para nuestro dispositivo, además, se sirve del mapa de obstáculos para diseñar una trayectoria con el mínimo coste posible desde un punto inicial hasta otro final.

En general el nodo que coordina el funcionamiento de todo lo anteriormente explicado, es el "explore_lite" que trata de definir las fronteras de un entorno determinado basado en la exploración del mismo. Este nodo se encargará de enviar comandos a la base para determinar sus movimientos. Además, se suscribirá al tópico que envía la información relativa a la ocupación de celdas del paquete de navegación SLAM. Otra opción es que se suscriba al tópico que crea los "costmap" donde en lugar de servirse de un mapeado estándar como en el caso de la navegación SLAM, utilizará un mapa de obstáculos más completo.

La elección de uno u otro depende del entorno en el que vaya a explorarse, de esta forma, cobrará una mayor importancia la determinación de una serie de parámetros relacionados con la configuración del funcionamiento como son la mínima altura que supone una frontera, o la distancia a la que puede acercarse de la misma. Para su configuración debe lanzarse el nodo "move_base" para posteriormente ser capaz de controlarlo desde Rviz.

Tras esto, para cerciorarse de una correcta configuración del paquete, debemos emplear el nodo "navfn" para mover el robot hasta una posición deseada. Además, será necesario tomar información del mapa de obstáculos activando el parámetro "track_unknown_space". Una vez hecho todo esto puede lanzarse el nodo "explore_lite" que nada más ejecutarse comenzará instantáneamente a explorar su entorno.

A medida que vaya detectando fronteras publicará en el tópico “frontiers” la ubicación de las mismas para evitar colisiones y determinar un mapeado correcto del entorno. En la figura 6-13 se muestra el recorrido que realiza el dispositivo mediante flechas en color azul, y como va aumentando el mapa a través de la ubicación de fronteras.

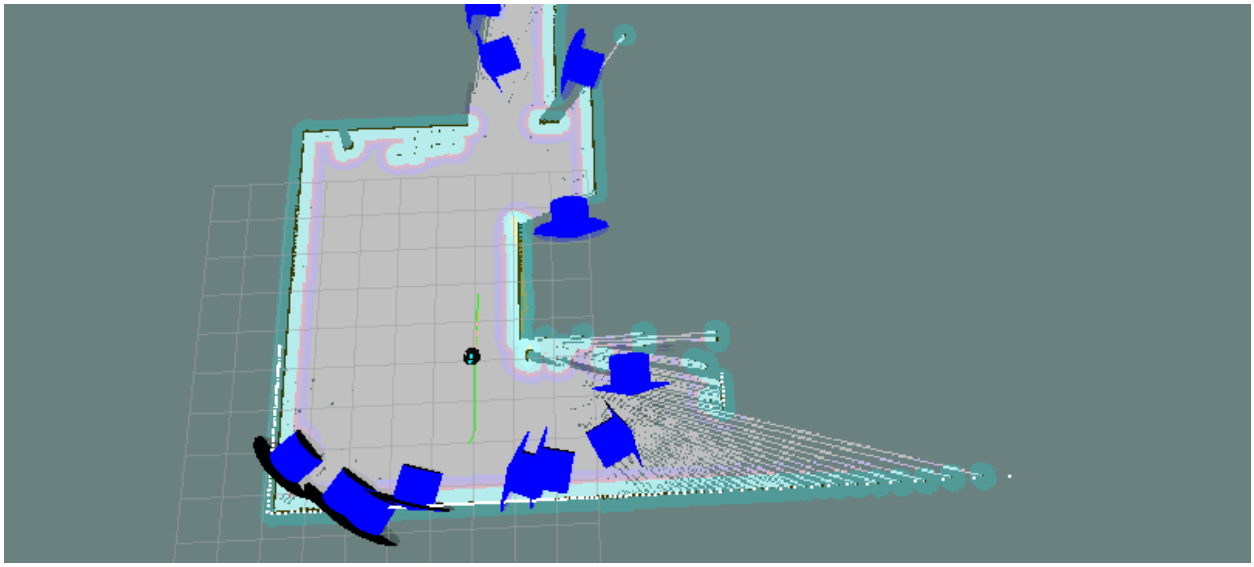


Figura 6-13 ejemplo de funcionamiento del nodo “explore_lite” (Fuente: [11])

7 RESULTADOS

En este apartado el objetivo será mostrar que todos los métodos anteriormente descritos funcionan y que por tanto lo que se ha ido exponiendo en este trabajo previamente tiene un resultado positivo y favorable. La muestra de resultados contará con aquellos referidos al apartado de visión, más concretamente al reconocimiento de señales y actuación en consecuencia. Además de aquellos del mapeado de entornos y planificación de trayectorias.

7.1 Resultados de detección de objetos

En cuanto a los resultados de visión se subdividirán en dos apartados diferentes. El primero de ellos, consistirá en el aprendizaje y posterior reconocimiento de imágenes. Dónde se mostrará la eficiencia del proceso descrito. En el otro se mostrarán las reacciones del dispositivo frente a la detección de las propias señales.

7.1.1 Resultados del reconocimiento de señales.

Como se ha aclarado con anterioridad en el apartado de las características de los componentes, la cámara es capaz de percibir cualquier objeto con un rango de entre 0.6 y 8m de distancia al objetivo. Por tanto, en las pruebas realizadas se ha tratado de evaluar ese rango colocando señales a distintas distancias. Como se ha aclarado en el apartado de limitaciones, el dispositivo no reacciona igual ante todas las señales y por tanto los resultados con algunas han sido más favorables y con otras no tanto.

Los resultados en cuanto al aprendizaje de imágenes mediante la detección de sus puntos clave han sido los siguientes:

Figura 7-1 Puntos clave de la señal de 120

Figura 7-2 Puntos clave señal de STOP

Figura 7-3 Puntos clave señal giro izquierda

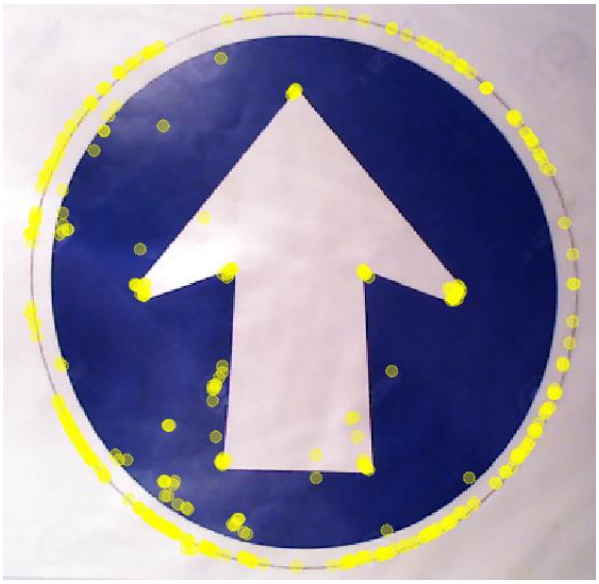


Figura 7-4 Puntos clave señal recto

Como puede apreciarse en las imágenes y como se explicó en el apartado de limitaciones, las señales con flechas tienen muchos menos puntos clave que aquellas con letras o números. Cabe destacar el hecho de que entre las dos son las señales con números las que tienen más puntos clave. Por tanto, son más fáciles de detectar.
A continuación, se muestran los resultados obtenidos en cuanto a la detección de imágenes. En el propio Programa "find object 2d" y a través del tópico objects como se explicó anteriormente.

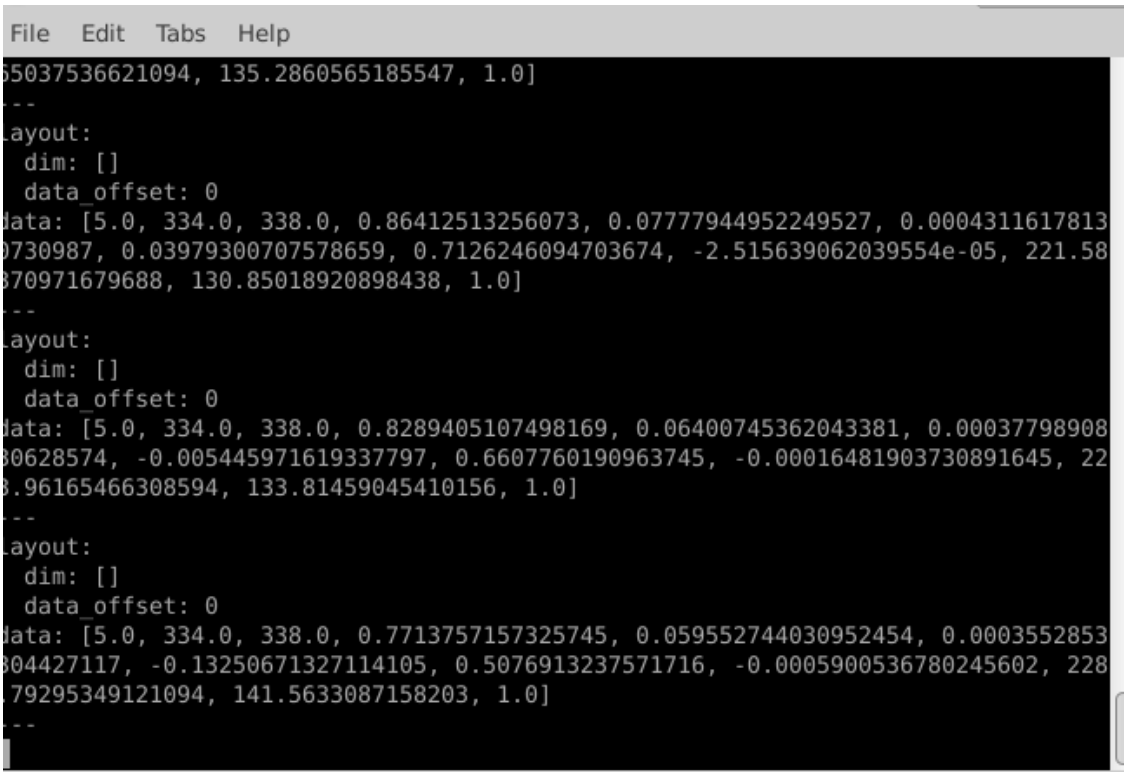


Figura 7-5 Reconocimiento de objetos mediante el tópico Objects

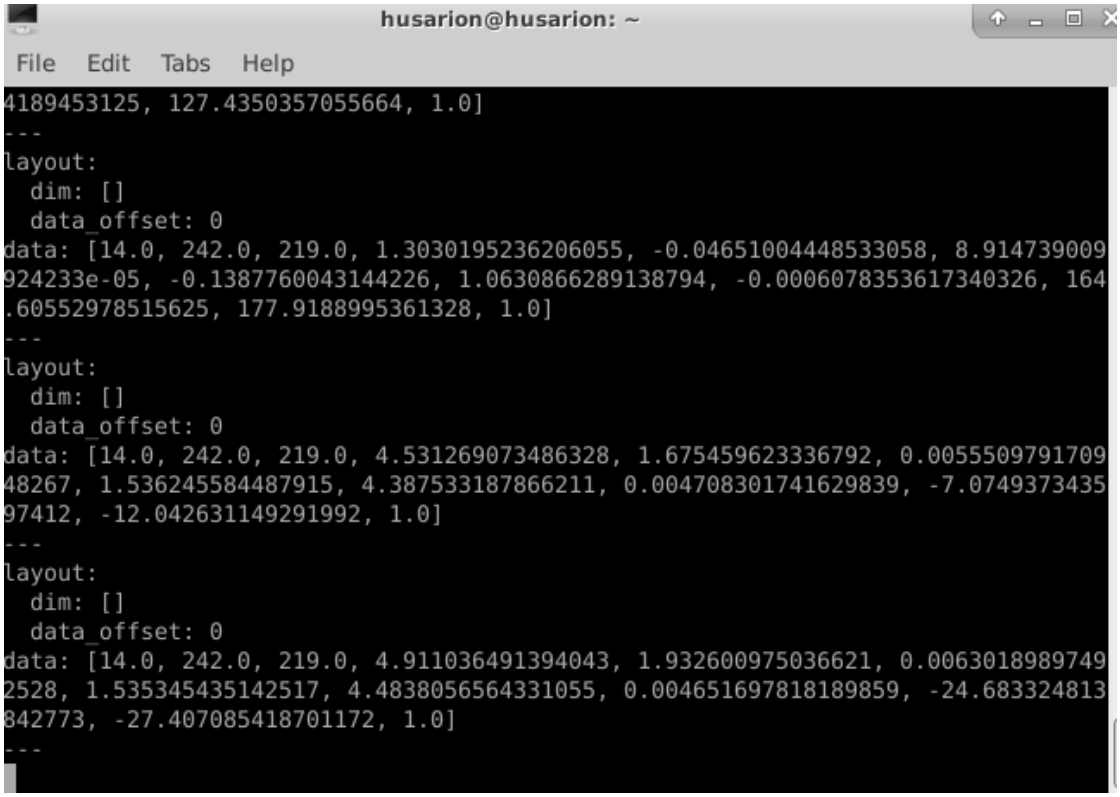


Figura 7-6 Reconocimiento de objetos mediante el tópico Objects

Como ya se explicó con anterioridad, el tópico objects devuelve el identificador de cada imagen además de su Distancia y orientación con respecto a la cámara.

Para determinar el rango de detección de nuestra cámara se han tratado de detectar objetos a 3 distancias distintas, La primera a 80cm la segunda a 120cm y la tercera a 220cm los resultados han sido los siguientes:

Figura 7-7 Primera distancia de reconocimiento

Figura 7-8 Reconocimiento en el programa find object 2d

Figura 7-9 Segunda distancia de reconocimiento

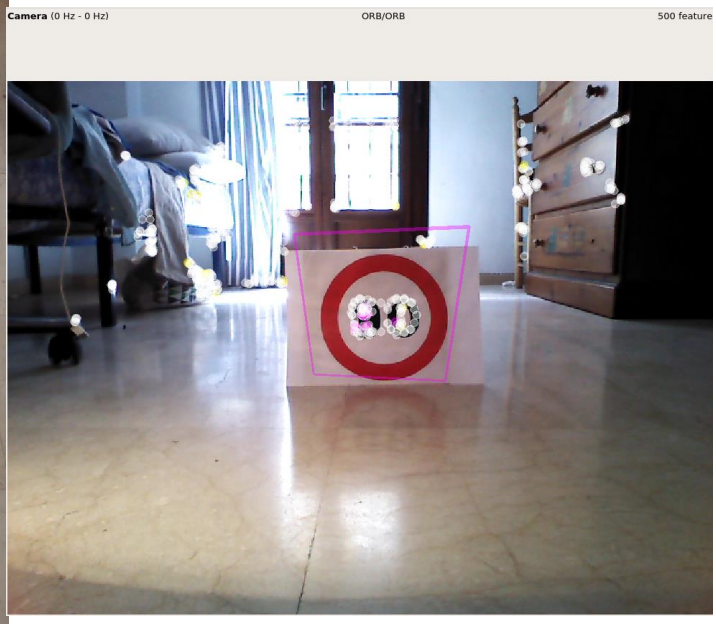


Figura 7-10 Reconocimiento en el programa find object

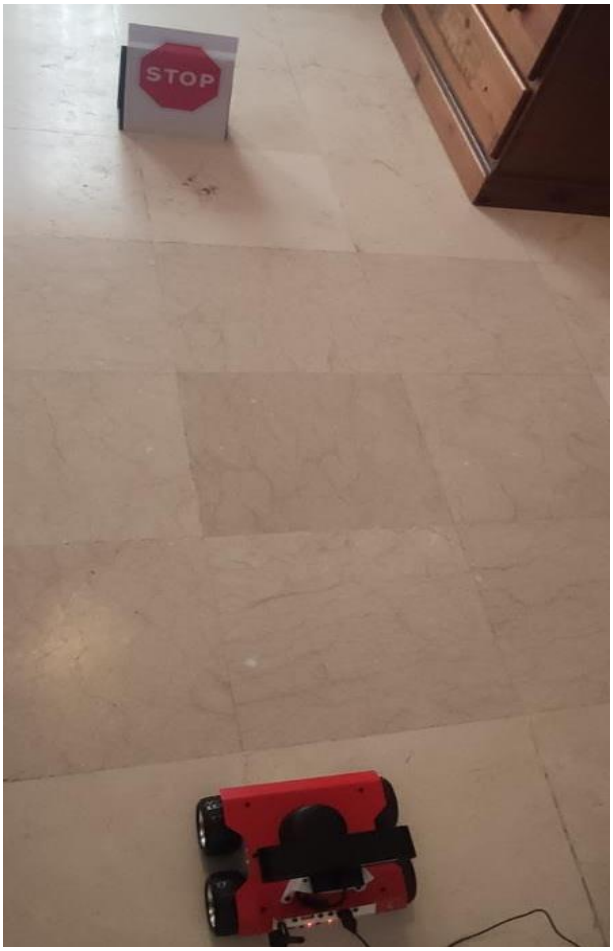


Figura 7-11 tercera distancia de reconocimiento

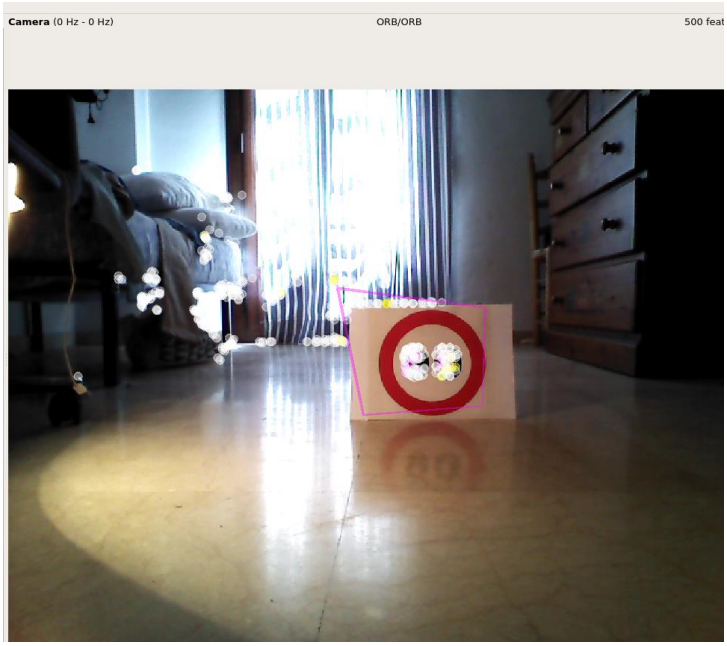


Figura 7-12 Reconocimiento en el programa Find object

Como puede apreciarse en las imágenes se ha usado una señal distinta a la del modelo ya que proveía mejores resultados como se ha explicado previamente. Aun así, la tercera distancia de reconocimiento no llegó a superarla ya que el sistema no reconocía la señal con tanta separación. Debe aclararse que la luminosidad se ha regularizado para mejorar los resultados, ya que un cambio en la misma puede suponer un desfase en la detección de puntos y la comparación con los ya existentes.

Por último, cabe decir que se ha empleado el programa únicamente para la detección de señales, pero podría utilizarse para la detección de otros muchos objetos o incluso caras. Ya que, la detección de puntos clave en objetos con numerosas características le resulta mucho más sencilla.

7.1.2 Resultados de la reacción tras detección

En este apartado se mostrarán los resultados obtenidos en la reacción del dispositivo tras la detección de algunas de las señales.

En primer lugar, se adjuntan una serie de videos en los que se muestra la reacción del dispositivo tras detectar la señal de giro a la derecha o la señal de giro a la izquierda. En estos puede apreciarse como actúa correctamente y vira en mayor o menor medida dependiendo del tiempo de detección de la imagen.

Video 7-1 Reacción de robot tras la visión

Video 7-2 Reacción del robot tras la visión

Tras esto, y con el fin de evaluar la mayor parte de las señales que se introdujeron en el dispositivo, se creó un circuito con sillas de las que pendían las señales que debían ser detectadas. En esta ocasión al estar fijas dichas señales, se consigue que se estabilice el tiempo de detección de las mismas y que el resultado sea más exacto.

El circuito consiste en una primera señal de limitación de velocidad a 80Km/h que aumenta la velocidad del dispositivo, posteriormente una señal de 50km/h hace que el robot reduzca su velocidad, a continuación, una señal de giro a la derecha obliga a virar a la base. Seguidamente, gira hacia la izquierda y por último, para al detectar la señal de STOP. Se adjunta un video con el resultado obtenido.

Video 7-3 Reacción del robot tras la detección

Cabe decir que esta prueba es bastante complicada, como ya hemos visto la detección de señales depende en gran medida de la luminosidad de cada uno, de la comparación de las mismas con el momento en que fueron detectadas, de su orientación en cada situación y de varios factores más.

En el apartado de los giros, estas señales son algo más difíciles de ser detectadas y por tanto el tiempo de reacción del robot varía. Todo ello podría configurarse modificando la velocidad radial de giro con la que se mueve cada vez que detecta la imagen.

Por otro lado, las señales de limitaciones de velocidad, así como la de STOP funcionan bastante bien, son detectadas desde casi cualquier ángulo y la reacción no se hace esperar, además no tiene casi retraso y la adaptación tras dejar de verlas es casi inmediata.

Por último, cabe decir que las posibilidades que ofrece este programa son muchísimas, el circuito podría variar de muchas maneras, el tamaño de las señales podría ser mayor para ser detectadas desde mucha más distancia, o el empleo de ciertos elementos podría hacerse de modos distintos.

7.2 Resultados de mapeado y planificación de trayectorias

En este apartado se mostrarán los resultados obtenidos del mapeado de entornos en un primer subcapítulo seguido de la planificación de trayectorias. Además del mapeado de un circuito limitado por conos y la inclusión de los sistemas de visión dentro del mapeado.

7.2.1 Resultados del mapeado de entornos

En este primer apartado van a mostrarse los resultados del mapeado de un entorno determinado. En este caso los lugares escogidos para que la experiencia sea realizada son el salón de mi casa y mi propia casa al completo. Se adjuntarán una serie de imágenes para comparar el trabajo realizado por el dispositivo con la realidad en la que explora.

Para la realización de este apartado se ha controlado el dispositivo de manera manual para que recorra la totalidad del entorno que nos interesaba que mapease. Cabe destacar que el mapa creado se muestra desde el punto de vista de la planta mientras que las imágenes son tomadas desde una perspectiva diferente. Aun así, se pretenderá aclarar cada elemento para que todo concuerde.

Las imágenes de mapeados que van a mostrarse como se ha explicado con anterioridad son tomadas de la herramienta Rviz con la que se ha trabajado en este proyecto para el desarrollo de este apartado.

Los resultados obtenidos han sido los siguientes:

Figura 7-13 Aspecto del entorno a mapear

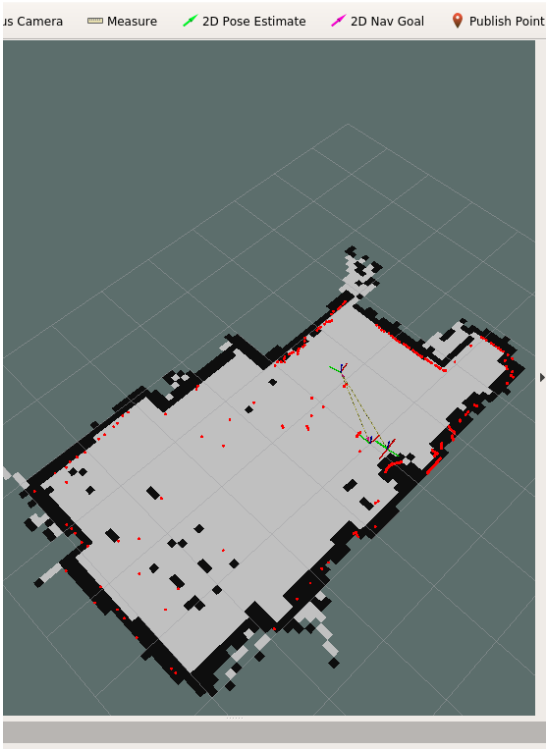


Figura 7-14 Resultado del mapeo en Rviz

Como puede apreciarse los resultados son bastante buenos, el sistema muestra los sofás como fronteras ya que el sensor no detecta que cuentan con espacio por debajo, esto podría solucionarse empleando los sensores de infrarrojos o la cámara, pero se escapa a los objetivos de nuestro trabajo. De igual manera, los puntos negros de la parte izquierda de la imagen se corresponden con las patas de las sillas y la mesa. Se pueden apreciar también como fronteras el resto de los muebles que ocupan la habitación.

Figura 7-15 resultado del mapeo de una casa en Rviz

En esta imagen podemos apreciar el resultado de mapear no solo el salón que se encuentra en la parte inferior de la figura, si no también unas cuantas más de habitaciones del hogar. En la parte superior, nos encontramos la cocina además del aseo y una de las habitaciones. Cabe destacar que en el salón se han detectado con más exactitud los huecos que se encuentran entre los sofás, así como la totalidad de las sillas y las mesas que allí se encuentran.

Como se ha dicho previamente, los resultados obtenidos son muy buenos y se ajustan a la perfección a la realidad. Esta parte del proyecto es indispensable ya que posteriormente la planificación de trayectorias se basará en lo bien o mal que se haya hecho esta sección. Cabe decir que este mapa no es una captura directamente tomada del programa si no que ha sido extraído en un archivo particular.

Como aplicación de este proyecto se plantea la adaptación del mapeado a la detección de una serie de obstáculos, en este caso conos, que delimiten el recorrido de un circuito en particular, la idea es que, mediante una vuelta de reconocimiento, el dispositivo sea capaz de detectar el circuito y mapearlo de manera simultánea.

Es por ello por lo que se realizó la compra de una serie de conos, se buscó un espacio abierto en el edificio de laboratorios y se elaboró un circuito para que fuese mapeado. Este cuenta con dos conos que delimitan por donde debe cruzar el robot para avanzar.

En una primera figura, se mostrará el circuito elaborado visto desde la altura, asemejándose así a lo que el programa plasma posteriormente en su herramienta, después, se adjuntará el mapeado que se crea en el sistema.

Los resultados obtenidos son los siguientes:

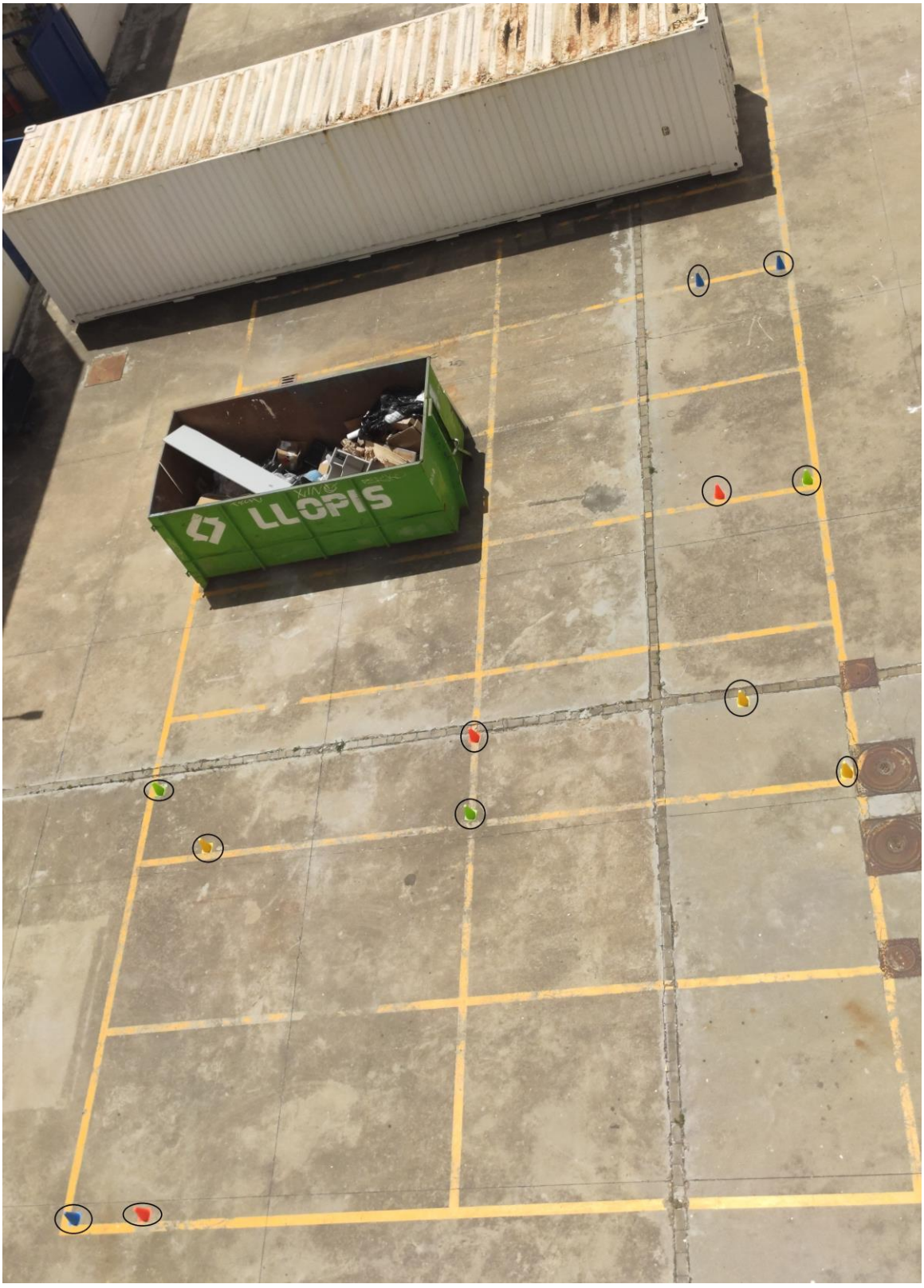


Figura 7-16 circuito elaborado en los laboratorios

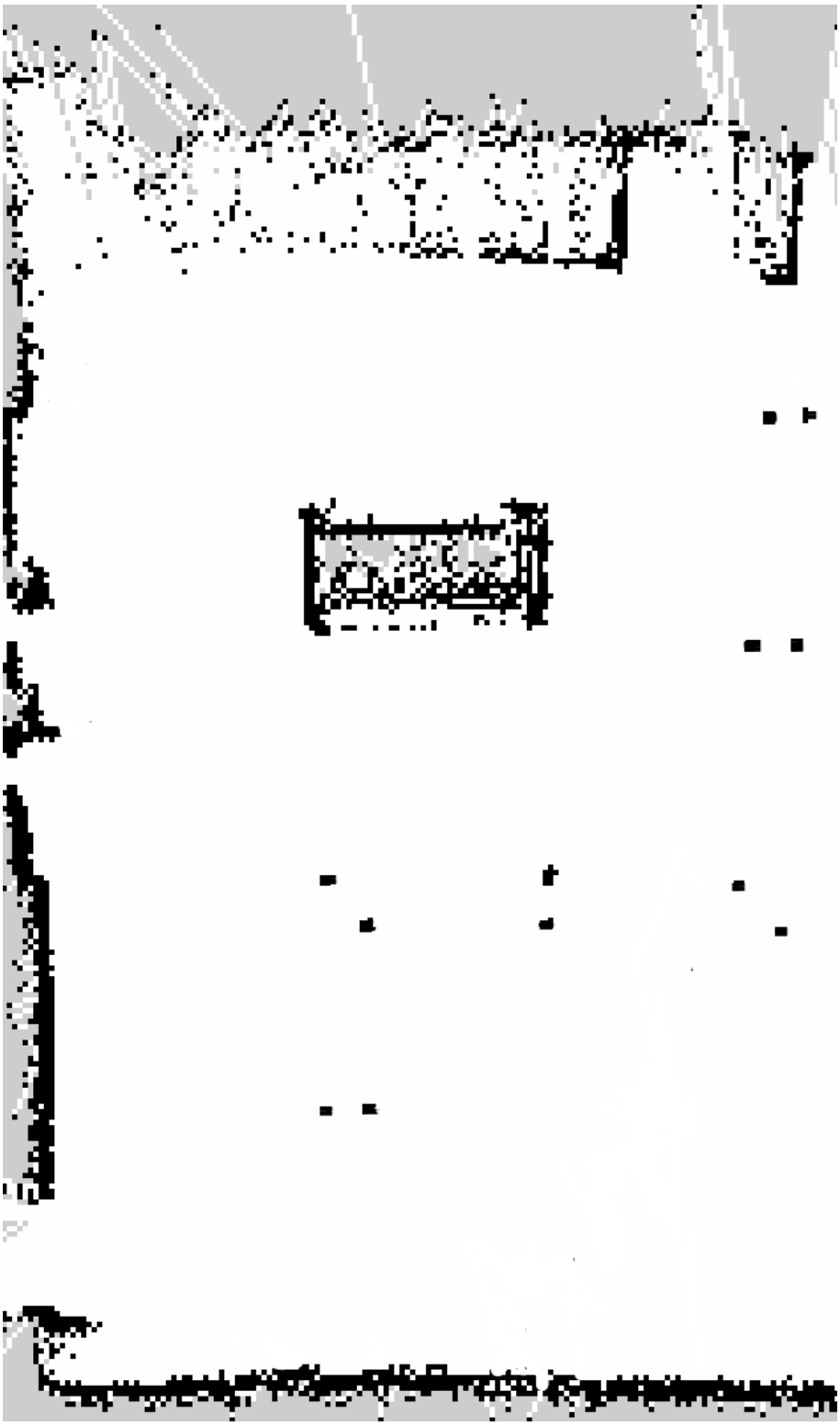


Figura 7-17 Mapeo del circuito tras vuelta de reconocimiento.

Como puede apreciarse los puntos negros que se ven en la figura corresponden con los conos que se colocaron para delimitar el circuito, queda claro como el dispositivo es capaz de mapear en un entorno abierto cualquier obstáculo y elaborar así un mapa del circuito para recorrerlo posteriormente.

Tras todo esto, se planteó la inclusión de los sistemas de visión dentro del mapeado de entornos, como se ha explicado previamente, nuestro robot cuenta con una cámara incorporada que puede funcionar de manera síncrona al LIDAR. La idea es emplear una serie de "displays" en Rviz que implementen un esquema en 3 dimensiones de los obstáculos que se encuentren.

Siguiendo con la idea de mapear un circuito, se consideró que el mejor modo para incluir la visión en los sistemas de mapeado era utilizando como obstáculos los propios conos empleados en el apartado anterior. Para ello se buscó un entorno en el que se colocaron en forma de zigzag y se examinaron mediante el uso de la cámara. El resultado que se mostrará a continuación no es más que la nube de puntos que se ha formado a partir de todo lo percibido en la imagen.

Los resultados obtenidos son los que se muestran a continuación, en un primer momento se presenta el entorno en la realidad que se va a mapear, para posteriormente adjuntar el resultado desde varios ángulos distintos.

Figura 7-18 Entorno para mapear mediante visión

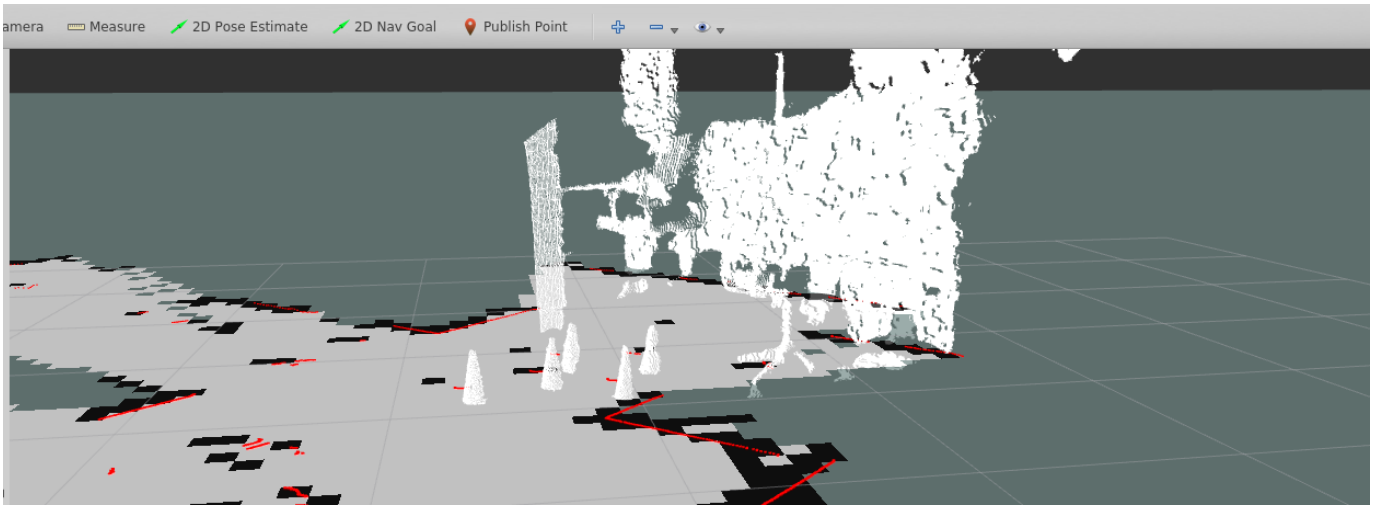


Figura 7-19 resultado del mapeado con visión en Rviz

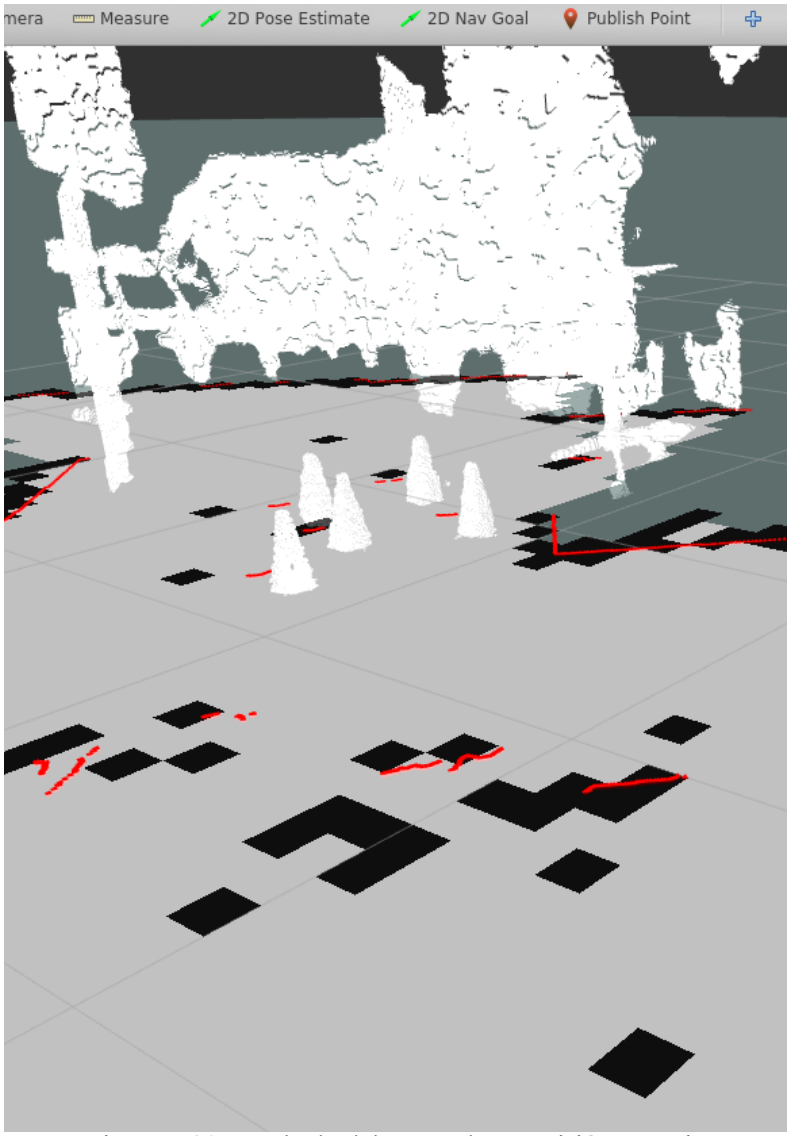


Figura 7-20 Resultado del mapeado con visión en Rviz

Como se aprecia en la captura tomada como resultado, los 5 conos colocados en zigzag son fácilmente reconocibles por la nube de puntos que los conforman. Por otro lado, se puede apreciar también la base con ruedas de una silla que también está en el plano, o el armario de la izquierda, así como la pared del fondo.

Por último, cabe decir que las posibilidades que ofrece la inclusión de la visión dentro del mapeado de entornos son muchas. Con esto, comenzamos a ser capaces de apreciar los distintos obstáculos en 3 dimensiones, pudiendo elaborar trayectorias más exactas en las que queden definidos un mayor número de parámetros.

La inclusión de la nube de puntos y la visión en el mapeado, se consideran un elemento del que se podrá desarrollar mucho más en un futuro y que en la situación actual, por ser otro el tema principal del proyecto no podrá aportar más que el valor lectivo que supone aprender a usarlo.

7.2.2 Resultados de la planificación de trayectorias

En este apartado presentaremos los resultados obtenidos en el apartado de planificación de trayectorias explicado con anterioridad. En primer lugar, se presentará el entorno en el que se realizarán las pruebas, es el mismo descrito antes. Aun así, se aporta de nuevo una foto del mismo comparado con el mapa de obstáculos o "costmap" con el que se trabajará.

En este primer apartado, los resultados son los esperados, ya que el método de funcionamiento es bastante parecido a lo que ya se ha realizado con anterioridad en el mapeado de entornos, con la excepción de que se añaden unos cuantos "displays" más a Rviz que se encargarán de elaborar los mapas de obstáculos locales y globales.

Los resultados obtenidos se presentan a continuación

Figura 7-21 entorno para planificación de trayectorias

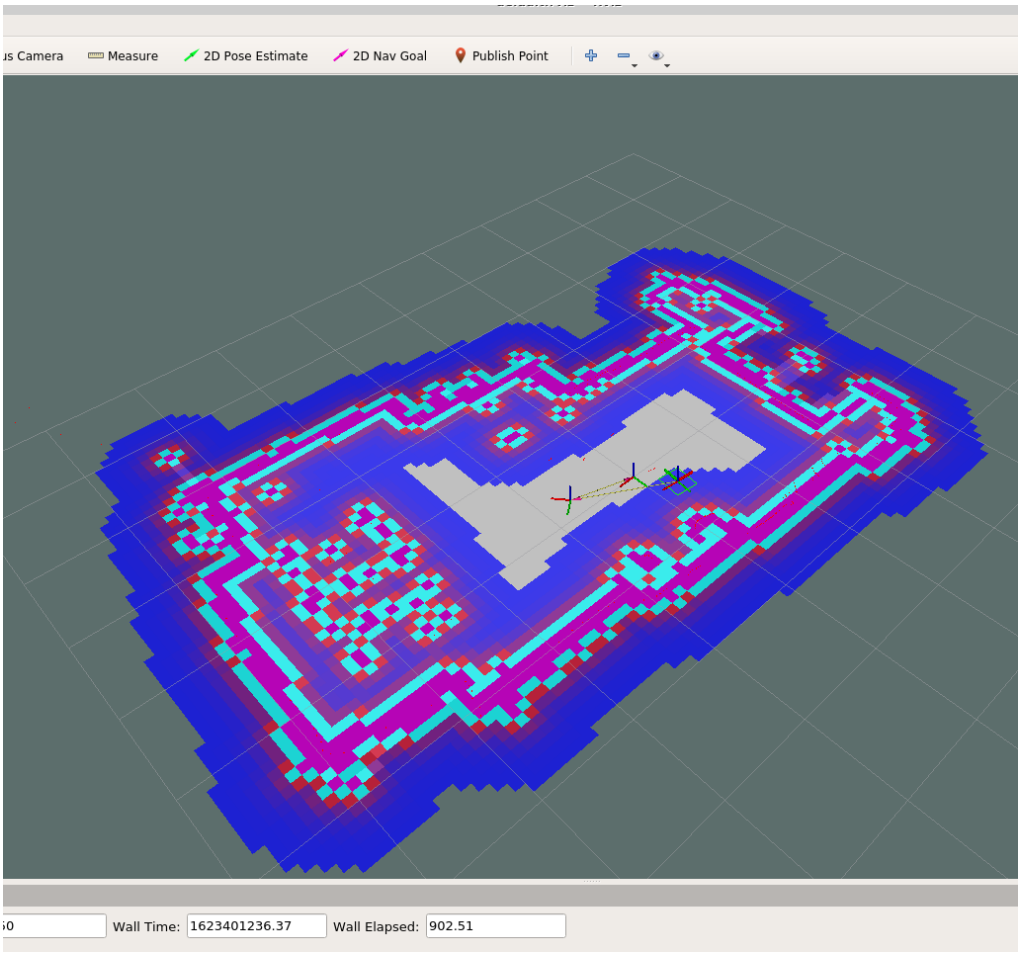


Figura 7-22 Resultado del mapa de coste en rviz

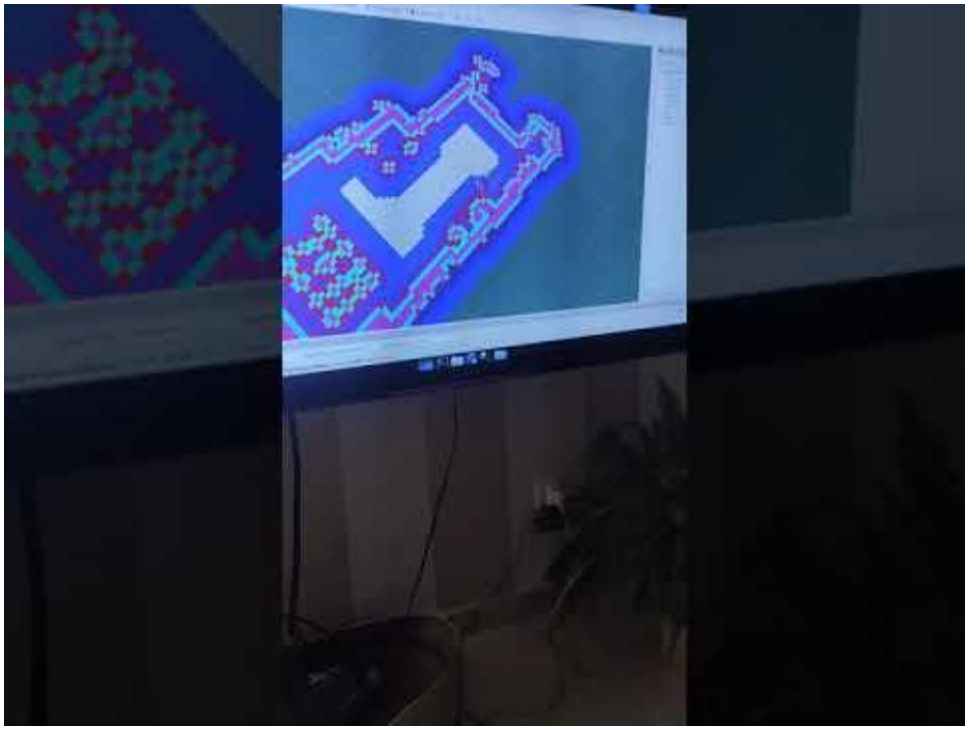
En un segundo apartado se presentan ahora si los resultados de la planificación de trayectorias, tras haber realizado el mapeado del entorno y haber conseguido elaborar con éxito un mapa de coste o "costmap" que reúna todos los obstáculos del entorno, se pone en marcha la planificación.

En la ventana de Rviz se muestra la opción de "2D Nav Goal" tras seleccionarla y elegir un punto del propio mapa, nuestro dispositivo elaborará la ruta óptima para alcanzar la posición final desde su emplazamiento al comenzar.

Los resultados que se presentan a continuación son un par de videos en los que esto se pone en marcha. En el primero de ellos, se aprecia como se elige un punto antes de que el robot se desplace y lo alcance, se realiza el proceso por dos veces en lugares cercanos al monitor ya que se encontraba conectado al mismo.

El segundo de los videos nos muestra el mismo proceso de inicio, la diferencia es que en esta ocasión, un compañero desconecta la imagen del monitor para que el dispositivo quede libre y pueda alcanzar la posición deseada.

Los resultados se muestran a continuación.



Video 7-4 ejemplo de planificación de trayectorias

Video 7-5 Ejemplo de planificación de trayectorias.

Como puede apreciarse en los videos existen ciertas limitaciones relacionadas con la obligación de estar conectado a un monitor para designar el punto de destino al que dirigirse. Aun así, el funcionamiento es bueno y el dispositivo es capaz de alcanzar los objetivos marcados sin problema.

Como conclusión, cabe decir que, en el apartado de planificación de trayectorias, como culmen al mapeado de entornos y la detección de obstáculos, el correcto funcionamiento de la planificación plasmado en resultados prácticos supone una gran cantidad de posibilidades de desarrollo para un futuro.

Al no contar con limitaciones espaciales, el dispositivo sería capaz de explorar entornos mucho más grandes que el actual. Podría moverse por dichas zonas de manera autónoma y el resultado puede ser el de tener información de un punto en particular. Sin importar el tamaño del entorno que se estudie y sin tener que desplazar una persona física hasta esa posición.

8 CONCLUSIONES

Una vez presentados los resultados en el apartado anterior, con este capítulo, se pretende valorar de manera genérica el trabajo realizado. Además de hablar de los posibles avances que realizar en el futuro y de comentar las limitaciones que en el desarrollo del proyecto se han encontrado.

8.1 Conclusiones

Para comenzar, es importante decir que al inicio de este proyecto nunca había trabajado si quiera con un sistema operativo Ubuntu en profundidad. La programación en terminales, los comandos útiles y el trabajo con dicho entorno, para mi han supuesto el primer reto y también el primer incentivo. Además, he sido capaz de conocer la magnitud de la comunidad de ROS que es enorme, ofrece muchas posibilidades diferentes, y ha puesto al alcance de mi mano un gran número de conocimientos.

Es por ello por lo que el poder moverme en su entorno y programar en el mismo ha sido uno de los alicientes más importantes en la realización del proyecto. Cabe decir que no soy ningún experto, aún queda mucho por aprender, pero la realización del trabajo me ha supuesto un gran avance en cuanto a entendimiento con el programa se refiere. Es también importante destacar, la importancia de la comunidad de la empresa Husarion. En su foro, siempre he hallado respuestas, además de muchísima información extra.

En cuanto a los objetivos de este proyecto en el apartado de visión, puedo decir que son extremadamente satisfactorios. Cabe destacar que todo lo realizado ha sido en la realidad, en ningún momento se ha empleado simulador alguno y como se ha aclarado en momentos anteriores, los inconvenientes son muchos.

Estos aparecen cuando varían ciertos aspectos como son:

- La luminosidad del objeto al tomar la imagen y a la hora de compararla.
- La orientación o rotación del mismo en la situación en la que se encuentre.
- El tiempo que transcurre detectando la señal en sí.

Además de un largo etcétera que hace que cada intento sea distinto al anterior. Normalizar todo esto es muy complicado, sobre todo cuando hay una serie de factores que no dependen de uno mismo. Sin embargo, la reacción del dispositivo ante la detección de señales es un éxito y funciona correctamente.

Es importante señalar que los algoritmos empleados en este apartado, así como los programas utilizados, se fundamentan en su mayoría en la biblioteca de OpenCV de la que me he nutrido a la hora de fundamentar los distintos códigos empleados en estos procesos.

Por otro lado, en lo relativo al mapeado de entornos y la planificación de trayectorias los resultados obtenidos también me congratulan. En el apartado del mapeado de entornos, he sido gratamente sorprendido por la eficiencia y exactitud con la que el sistema trabaja, los distintos resultados realizados en los ensayos se adaptan a la realidad con un grado de acierto elevado.

En un futuro, me gustaría hacer más hincapié en la importancia de situar dentro de un mapa objetos específicos que posteriormente puedan ser detectados mediante los sistemas de visión, con el fin de entrelazar de manera más palpable la detección de objetos y el mapeado de entornos.

Hay que ser consciente de la limitación con la que trabajamos, ya que, en un principio, el mapeado de entornos se realiza solo en 2 dimensiones, lo que supone un problema a la hora de detectar obstáculos que se encuentren por encima o por debajo del sensor LIDAR. Sin embargo, con la inclusión de los sistemas de visión en el mapeado se produce un gran avance, siendo capaz de introducir la altura y por tanto una tercera dimensión.

Para trabajos futuros, creo que es de suma importancia que este apartado se desarrolle con mayor profundidad, debido a las enormes posibilidades que ofrece en comparación con el trabajo a emplear para su inclusión. En cuanto a la planificación de trayectorias, tras haber conocido el funcionamiento del algoritmo y configurarlo para su desarrollo posterior, creo que arroja resultados muy buenos.

Es importante conocer que tener que estar conectados a un monitor para poder seleccionar los puntos de destino deseados limitaba mucho el radio de funcionamiento del dispositivo. Este hecho se produce ya que la conexión de manera remota al robot se ha podido realizar en numerosas ocasiones, pero cuenta con ciertas limitaciones en mi caso. Como el hecho de no poder abrir ventanas en Rviz. En el futuro debe trabajarse de manera más profunda en el control del dispositivo de manera telemática para evitar estos problemas. Aun así, el funcionamiento de la herramienta es correcto y ha mostrado buenos resultados en la puesta en escena.

En resumen, de todo lo anteriormente descrito, caben destacar las siguientes conclusiones de carácter general:

- En la detección de objetos encontramos una limitación alcanzada la distancia de los 150cm.
- El reconocimiento de señales funciona correctamente limitado solo por las condiciones del entorno y la restricción a figuras en 2 dimensiones.
- La reacción tras la detección de objetos es correcta, pero mejoraría si se estandarizase el tiempo que el dispositivo permanece reconociendo la señal.
- El mapeado de entornos se considera excelente, aunque tiene amplias posibilidades de mejora incorporando el uso de infrarrojos y uno más desarrollado de la cámara. Para así añadir una tercera dimensión
- La planificación de trayectorias es la esperada. A pesar de ello, se debe profundizar en el control de manera telemática del dispositivo para aumentar el rango de funcionamiento.

En el apartado de errores del proyecto se pueden destacar algunos en visión, que como se ha dicho depende de numerosas variables cambiantes y que particularizan cada ensayo. Quizás el momento en el que se capturen las imágenes que luego van a ser reconocidas, debería tener las mismas situaciones de luminosidad, orientación altura, inclinación que el momento en el que se detectan, ya que, aunque soporte cierto grado de variación, los resultados mejorarían mucho.

En la detección de obstáculos empleando la visión hay un error en el encuadramiento de los conos creados como nubes de puntos por la cámara, y la detección como obstáculos por parte del LIDAR, esto se muestra en la figura 8-1 que se adjunta a continuación.

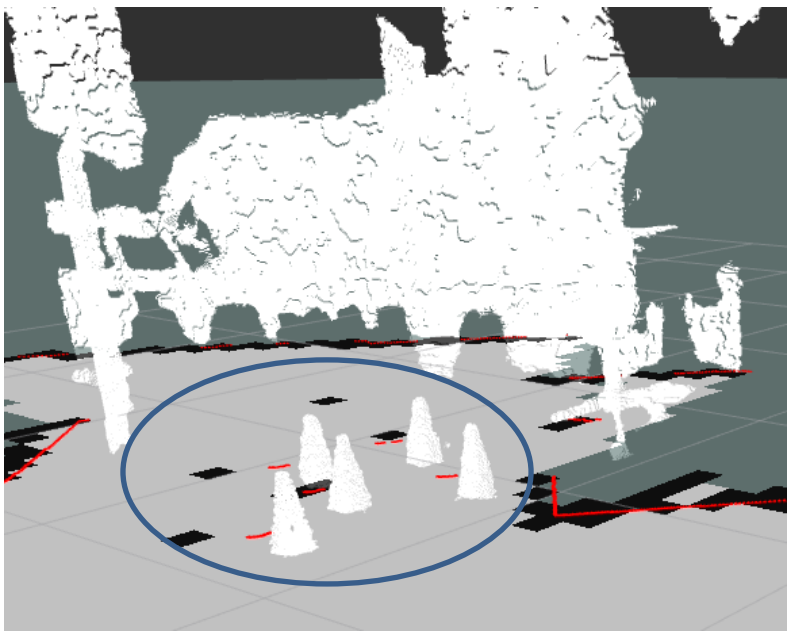


Figura 8-1 Error en el mapeado

Como se ha explicado anteriormente, la inclusión de la cámara supone un gran avance en el mapeado. Aun así, su funcionamiento en concordancia con el resto de los sensores puede ser algo a tratar en un futuro. Es obvio que en nuestro caso no funciona con exactitud, ya que hay una diferencia considerable entre la nube de puntos y su posición según el LIDAR.

8.2 Limitaciones y posibles mejoras

Las limitaciones que se han encontrado en este proyecto se dividen en 3 apartados, el primero de ellos es el relativo al hardware del dispositivo, su sistema operativo, es limitado, no cuenta con una gran cantidad de programas para el desarrollo de código y además tan solo cuenta con 2GB de memoria RAM, esto en un momento determinado supone que al lanzar numerosos nodos a la vez y tratar de hacer funcionar la cámara junto con el LIDAR, y alguna otra herramienta el sistema colapsa y da fallos, generalmente termina cerrando todos los programas y debemos volver a empezar.

Como posible mejora no hay mucho que proponer ya que el departamento cuenta con el modelo PRO de la empresa que dobla la memoria RAM y ofrece una serie de posibilidades más amplias, si en un futuro se profundiza en este objetivo, sería recomendable trabajar con dicho modelo desde un primer momento.

El segundo apartado de las limitaciones es el referido a la visión, ya se ha comentado mucho la limitación que suponen los factores externos, pero en este caso me gustaría resaltar el hecho de que nuestro programa de detección de características trabaja únicamente en 2 dimensiones, por tanto, es difícil realizar la detección de elementos u objetos en 3 dimensiones. Como por ejemplo los conos empleados en la realización de este proyecto.

Como posible mejora, en un futuro creo que podría implementarse un sistema que detecte obstáculos en 3 dimensiones ya que facilitaría mucho la labor de planificación de trayectorias por ejemplo en un circuito limitado por conos. Que es la idea de la que se partía en el trabajo.

En el tercer apartado de nuestras limitaciones se comentará todo lo relativo a la planificación de trayectorias. Cabe destacar que la mayor parte de las mismas ya se han explicado con anterioridad, pero a modo de resumen se dirá que el principal impedimento es la necesidad de estar conectado a un monitor de manera constante, además de tener que contar con un ratón que se encargue de la elección del punto de destino.

Como posible mejora, ya se ha aclarado antes que se debería extender el uso de la pantalla portátil que puede desplazarse y seguir el dispositivo, además esta es táctil por lo que también eliminaría la necesidad de usar un ratón conectado al dispositivo. Objetivamente pienso que cada robot debe contar con una para alcanzar el máximo de posibilidades que ofrece.

Como análisis general, las limitaciones encontradas son las siguientes:

- Escasez de memoria RAM (2Gb) que limita el funcionamiento de varios procesos en paralelo.
- Factores externos cambiantes (luminosidad, tamaño, orientación) en la detección de señales que limita el tiempo de reconocimiento de cada una de ellas.
- Limitación de detección de objetos en 2 dimensiones por parte de nuestro algoritmo.
- Conexión constante a monitor en la planificación de trayectorias que reduce el rango de funcionamiento.

ANEXO A GUÍA DE USO

A. 1. Instalación de Ubuntu 18.04 en windows 10

A continuación, se explicará como se procede a la instalación del sistema operativo Ubuntu en su versión 18.04 en el sistema operativo Windows 10, esta guía será lo más breve posible y se advierte que puede variar en función del sistema operativo usado o el ordenador desde el que se parta. Los datos de partida en mi caso son los siguientes:

- Portátil: Acer Aspire F5
- Sistema operativo instalado: Windows 10
- Sistema operativo que instalar: Ubuntu 18.04 LTS
- Procesador: Intel Core i5 10th gen
- Ram: 12Gb

Seguidamente, se enumeran los pasos a seguir para realizar el proceso deseado:

1. Realizar una partición del disco duro donde se instalará el sistema operativo, esta se describe en el siguiente enlace: https://www.profesionalreview.com/2018/11/02/particionar-disco-duro-windows-10/

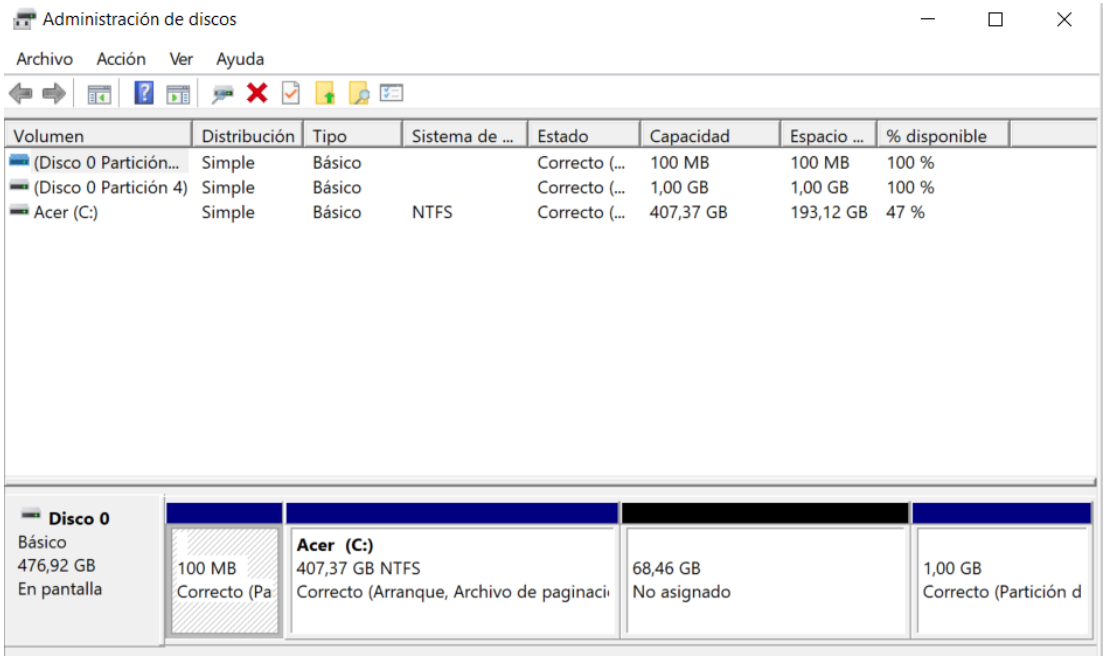


Figura A-1 Administrador de discos del PC

2. Descargar la imagen del sistema operativo en formato ISO, puede hacerse desde la propia página oficial ya que es gratuito: https://releases.ubuntu.com/18.04/
3. Flashear una memoria USB de la mano del programa RUFUS cuya instalación puede realizarse mediante el siguiente enlace: https://rufus.ie/es/
4. Modificar la Bios del ordenador configurando las prioridades de arranque del sistema para que lo haga desde el propio usb y no desde el sistema operativo preinstalado.
5. Iniciar el sistema finalmente desde la memoria USB y escoger la instalación del sistema operativo, no la prueba del mismo.

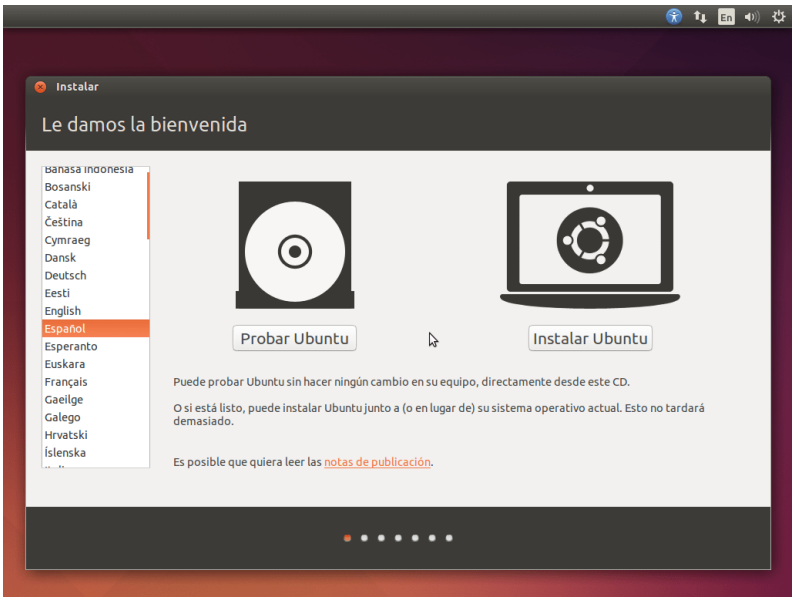


Figura A-2 Captura instalación ubuntu

6. Elegir la instalación normal, y escoger correctamente el disco creado anteriormente para realizar la instalación

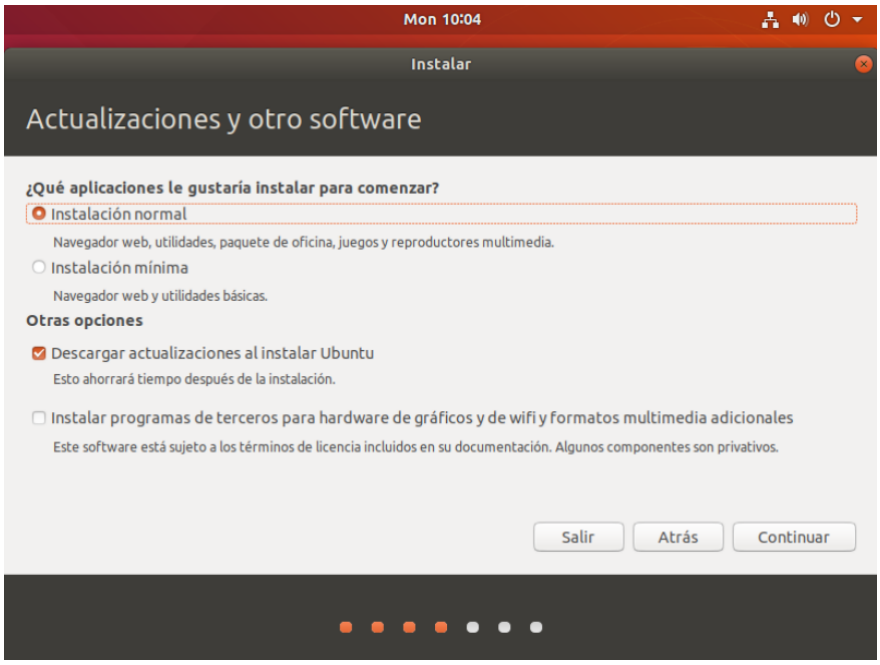


Figura A-3 Proceso instalación ubuntu

7. Hecha por fin la instalación, una vez iniciado el sistema debemos pulsar la tecla F2 repetidas veces hasta que nos muestre las distintas opciones de arranque, donde se encontrará nuestro sistema operativo original y también el propio sistema Ubuntu recién instalado, puede escogerse la que se prefiera.

A. 2. Instalación alternativa. Uso de Máquina virtual

Para aquellos que no les agrade el realizar una partición de discos e instalar un nuevo sistema operativo en su dispositivo se presenta una alternativa también viable que nos provee una solución a la necesidad del uso de Ubuntu.

Esta consiste en el uso de una máquina virtual que nos simplifica el trabajo y no necesita tantos pasos como en el caso anterior. En este manual mostraremos la manera más sencilla de utilizar Ubuntu en tu dispositivo, aunque cabe decir que el hecho de que sea una máquina virtual y no una partición del disco nos limita el trabajo en ciertas funciones específicas.

Los datos de partida son iguales al caso anterior, los pasos a seguir en este caso son los siguientes:

1. Descarga de la máquina virtual VMware Workstation PRO cuya instalación puede hacerse desde la web oficial del sistema en este enlace: https://www.vmware.com/products/workstation-pro/workstation-pro-evaluation.html

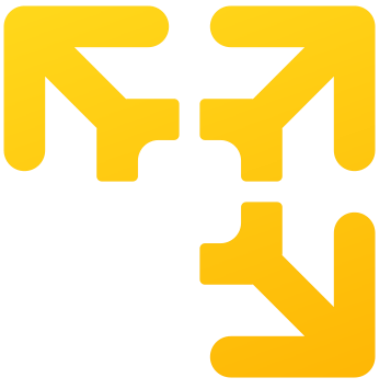


Figura A-4 Logo VMware (Fuente:[])

2. Descarga de la imagen del sistema operativo en formato ISO en este caso Ubuntu 18.04 LTS de su web oficial ya mencionada en el apartado anterior. https://releases.ubuntu.com/18.04/
3. Creación de una nueva máquina virtual dentro de VM Ware esto es fácilmente realizable desde su pantalla de inicio. No supone ningún problema.
4. En ella le ponemos el nombre que queramos como por ejemplo “Ubuntu” escogemos el tipo Linux, y la versión Ubuntu de 64 bit (en mi caso)
5. Elegimos el tamaño de memoria deseado para esta máquina y seleccionamos el archivo del sistema operativo de entre nuestros archivos.
6. Nos aparecerá una nueva máquina dentro de las ya existentes y podremos trabajar con ella con facilidad.

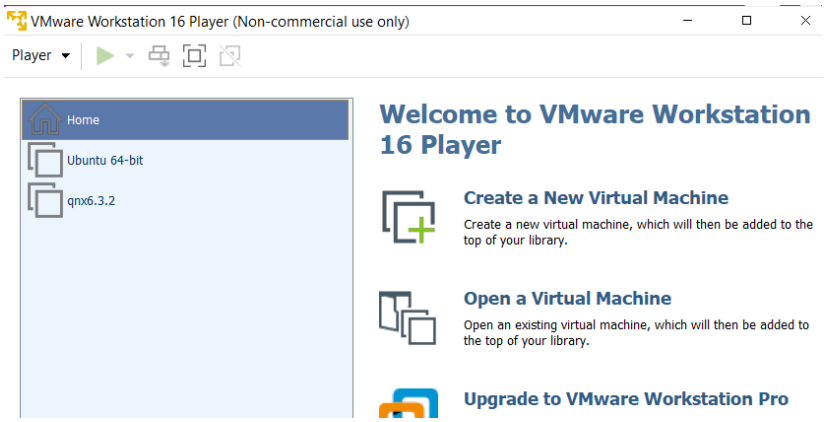


Figura A-5 Ejemplo ventana VMware

A. 3. Instalación de ROS Melodic Morenia

Como es lógico, ROS es indispensable dentro de nuestro proyecto ya que en él se fundamenta casi todo el mismo, su instalación una vez contemos con Ubuntu en el dispositivo es bastante sencilla. En el robot tenemos la suerte de ya viene instalado de fábrica y por tanto podemos trabajar directamente con el mismo.

Sin embargo, en el PC seremos nosotros los que instalemos el programa. Recordemos que la instalación se hace en el sistema operativo Ubuntu 18.04 LTS, y la versión de ROS es Melodic Morenia. Para ello seguiremos los siguientes pasos:

1. Configurar los repositorios de Ubuntu para permitir la instalación de cualquier fuente. Esto se explica en la siguiente guía: https://help.ubuntu.com/community/Repositories/Ubuntu
2. Abrir una ventana de terminal dentro de Ubuntu
3. Configuración del dispositivo para que acepte el software de packages.ros.org mediante el siguiente comando en la terminal:

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc
/apt/sources.list.d/ros-latest.list'
```

4. Se configuran e introducen las claves de acceso para permitir la instalación en el sistema mediante el siguiente comando:

```
sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-key C1
CF6E31E6BADE8868B172B4F42ED6FBAB17C654
```

5. Se actualiza el paquete debian mediante el siguiente comando:

```
sudo apt update
```

Es posible que cuando hagamos esto nos demande la clave de acceso predefinida para acceder al sistema operativo.

6. Instalación de ROS, existen varios paquetes distintos, algunos más livianos con menos programas y otros completos que cuentan con todas las herramientas, nosotros instalaremos el paquete completo, mediante la siguiente orden:

```
sudo apt install ros-melodic-desktop-full
```

Una vez hecho esto ya contamos con ROS en nuestro dispositivo de manera física.

7. Configuración del entorno añadimos las nuevas variables automáticamente a nuestra sesión de bash con el siguiente comando.

```
echo "source /opt/ros/melodic/setup.bash" >> ~/.bashrc
```

```
source ~/.bashrc
```

y con esto hemos terminado con la instalación e ROS, podemos probarlo mediante el uso de la orden “roscore” que iniciará la tarea maestra.

A. 4. Comandos útiles en ROS

A continuación, se adjunta una lista de una serie de comandos útiles de ROS que pueden ayudarnos a ejecutar cualquier tipo de tarea con una mayor facilidad.

- **Rosnode list:** con este comando obtendremos la lista de nodos que está en funcionamiento en cada momento.
- **Rosnode info nombre_nodo:** muestra por pantalla la información referente al nodo indicado, donde publica, a qué topico está suscrito y muchos más datos
- **Rosnode kill nombre_nodo:** para de ejecutar instantáneamente el nodo seleccionado.
- **Rostopic list:** muestra la lista de tópicos activos en cada momento
- **Rostopic info nombre_tópico:** imprime por pantalla la información relativa al tópico deseado, que nodo está suscrito, cual publica, su id y muchos más datos
- **Rostopic echo nombre_tópico:** enseña al usuario los mensajes que están siendo publicados en cada momento dentro del tópico.
- **Rosrun nombre_nodo nombre_nodo:=nuevo_nombre_nodo:** permite cambiar el nombre del nodo a cualquier otro que nos interese.
- **Rosrun nombre_nodo nombre_antiguo_tópico:=nombre_nuevo_tópico:** permite cambiar el tópico al que está suscrito un nodo por defecto
- **Param name=”nombre_parametro” value=”valor”/:** este comando nos permite cambiar el valor de cualquier parámetro al deseado
- **Roscore:** comando de vital importancia, necesario para inicializar la tarea maestra a la vez que el sistema de archivos con el que contemos, sin él es imposible comenzar a trabajar.
- **Cd ~/dirección_deseada:** orden utilizada para cambiar de carpeta en la que nos encontramos en cada momento.
- **Mkdir ~/dirección_deseada:** comando que nos permite crear una nueva carpeta en el repositorio que elijamos mediante su ruta.
- **Remap from=”nombre_antiguo” to=”nuevo_nombre”:** elemento que nos permite remapear un tópico de un nodo a otro.
- **Rosrun nombre_paquete nombre_nodo:** orden encargada de ejecutar un nodo
- **Roslaunch nombre_paquete nombre_launch:** comando que ejecuta el archivo launch que deseemos.

A. 5. Creación de un espacio de trabajo

Como se ha explicado previamente el espacio de trabajo o "ros_workspace" será el repositorio donde guardemos todos nuestros paquetes, nodos y archivos launchs o ejecutables que nos servirán para desarrollar nuestro proyecto, cabe destacar que es indispensable contar con el mismo para el correcto funcionamiento de nuestro trabajo.

Los pasos a seguir para la creación del mismo son los siguientes:

1. Creación de una nueva carpeta que será la ubicación de nuestro repositorio mediante el siguiente comando en la terminal de Ubuntu

   ```
   mkdir -p ~/ros_workspace/src
   ```

   Esto creará una nueva carpeta llamada "ros_workspace" y dentro de la misma otra nueva llamada "src"

2. Inicializar el espacio de trabajo mediante el siguiente comando ejecutado en la nueva carpeta "src" a la que deberemos movernos para hacerlo.

   ```
   cd ~/ros_workspace/src
   ```

   ```
   catkin_init_workspace
   ```

3. Por último, nos movemos de nuevo a nuestro directorio principal

   ```
   cd ~/ros_workspace
   ```

4. Y lo compilamos mediante la siguiente orden.

   ```
   catkin_make
   ```

5. Si todo ha salido correctamente nos aparecerá el siguiente mensaje en la ventana de la terminal:

   ```
   ####
   #### Running command: "make -j4 -l4" in "/home/pi/ros_workspace/build"
   ####
   ```

6. Creación de paquetes: ya sabemos de la importancia de los paquetes dentro de nuestro sistema y mediante esta orden podremos crear todos aquellos que nos interesen

   ```
   cd ~/ros_workspace/src
   ```

   ```
   catkin_create_pkg (nombre_paquete) (paquetes previos necesarios)
   ```

7. Creación de nodos: los ejecutables son los archivos en los que programamos los comandos que nos interesan que realice nuestro sistema y mediante esta orden podrás crearlos directamente en tu repositorio.

   ```
   touch ~/ros_workspace/src/(nombre_paquete/src/(nombre_nodo).cpp
   ```

8. Ejecutar el nodo: para comenzar debemos movernos a la carpeta de nuestro directorio principal y desde ahí lanzar el comando que configurará el entorno para luego poder ejecutar el nodo con el 2º comando.

   ```
   source ~/ros_workspace/devel/setup.sh
   ```

   ```
   rosrun nombre_paquete nombre_nodo
   ```

9. Ejecutar un archivo launch, para ello será necesario crear una carpeta dentro de "src" con su propio nombre para posteriormente guardar ahí el archivo y poder ejecutarlo.

   ```
   mkdir ~/ros_workspace/src/nombre_paquete/launch
   ```

   ```
   roslaunch nombre_paquete nombre_archivo_launch.launch
   ```

ANEXO B ARCHIVOS PROGRAMADOS

B. 1. Código del aprendizaje y detección de imágenes: *vision.launch*

```
<launch>
//necesario en cada archivo launch al empezar y acabar

<arg name="teach" default="true"/>  // definimos el caso por defecto como el de enseñanza

<arg name="recognize" default="false"/>  //damos también la opción del reconocimiento.

<arg if="$(arg teach)" name="chosen_world" value="rosbot_world_teaching"/>

<arg if="$(arg recognize)" name="chosen_world" value="rosbot_world_recognition"/>

<!-- ROSbot 2.0 -->

    <include file="$(find rosbot_ekf)/launch/all.launch">
    // se incluye el nodo del filtro de kalman extendido

    <include file="$(find astra_launch)/launch/astra.launch"/>
    // se incluye el nodo de funcionamiento de la camara

</group>

<node pkg="tf" type="static_transform_publisher" name="camera_publisher" args="-0.03 0 0.18 0 0 0 base_link camera_link 100" />
    // se lanza el nodo tf que nos indica la posición del robot en cada momento

<node name="teleop_twist_keyboard" pkg="teleop_twist_keyboard" type="teleop_twist_keyboard.py" output="screen"/>
    // se ejecuta el nodo que nos permite controlar el dispositivo mediante el teclado

<node pkg="find_object_2d" type="find_object_2d" name="find_object_2d">
// se abre el programa con el que trabajamos

<remap from="image" to="/camera/rgb/image_raw"/> // se redirige la imagen de la cámara de un tópico a otro

<param name="gui" value="$(arg teach)"/> //gui indica si el robot va a estar estático o no, por ello
                                        //su valor se asocia a la veracidad del parámetro teach

<param if="$(arg recognize)" name="objects_path" value="$(find action_controller)/images_vision/"/>
// si entramos en modo reconocimiento hay que especificar donde se encuentran las imágenes enseñadas previamente

</node>

  <node pkg="action_controller" type="action_controller_node" name="action_controller" output="screen"/>
  // se define el nodo que vamos a emplear para actuar en consecuencia de la detcción de imagenes

</launch>  // necesario en cada archivo launch al principio y final.

```

Como puede apreciarse en este archivo, la ejecución del mismo va seguida de la definición de una serie de parámetros cómo son “teach” o “recognize” los posibles comandos serían los siguientes:

Roslaunch tutorial_pkg visión.launch teach:=true recognize:=false // con esto ejecutamos en modo aprendizaje

Roslaunch tutorial_pkg visión.launch teach:=false recognize:=true // con esto ejecutamos en modo detección

B. 2. Nodo de actuación tras reconocimiento de señal: *action_controller_node.cpp*

```
1  #include <ros/ros.h>  // inclusión de bibliotecas
2  #include <std_msgs/Float32MultiArray.h>  // inclusión de bibliotecas
3  #include <geometry_msgs/Twist.h>  // inclusión de bibliotecas
4
5  #define GIRO_IZQDA 1  // Definición de constantes respecto a cada imagen
6  #define GIRO_DCHA 2   // Definición de constantes respecto a cada imagen
7  #define RECTO 3   // Definición de constantes respecto a cada imagen
8  #define MAX_VEL 4   // Definición de constantes respecto a cada imagen
9  #define STOP 5   // Definición de constantes respecto a cada imagen
10 #define MED_VEL 6   // Definición de constantes respecto a cada imagen
11 #define MIN_VEL 13   // Definición de constantes respecto a cada imagen
12 #define vel 14   // Definición de constantes respecto a cada imagen
13
14 int id = 0;
15 ros::Publisher action_pub;
16 geometry_msgs::Twist set_vel; //establecimiento de que los mensajes
17                               //varían la velocidad del dispositivo
18
19 void objectCallback(const std_msgs::Float32MultiArrayPtr &object)
20 //función que recibe el mensaje de la detección en cada momento
21 {
22   if (object->data.size() > 0) //se entra en el if si se detecta algo
23                           //esto se controla con el tamaño del objeto
24   {
25     id = object->data[0];// se guarda el id del objeto en una avriable
26
27     switch (id) // se inicia un SWITCH/CASE
28     {
29     case GIRO_IZQDA: // en caso de detectar giro izqda.
30       set_vel.linear.x = 0; //la vel. lineal se para
31       set_vel.angular.z = 5; //la vel. angular se vuelve positiva
32       break;
33     case RECTO: // en caso de detectar recto
34       set_vel.linear.x = 0.5 //la vel. lineal se vuelve positiva
35       set_vel.angular.z = 0; //la vel angular se vuelve cero
36       break;
37     case GIRO_DCHA: // en caso de detectar giro dcha.
38       set_vel.linear.x = 0; //la vel. lineal se para
39       set_vel.angular.z = -5;  //la vel. angular se vuelve negativa
40       break;
```

```
41          case STOP: // en caso de detectar stop
42              set_vel.linear.x = 0; //la vel. lineal se para
43              set_vel.angular.z = 0; //la vel angular se vuelve cero
44              break;
45          case MAX_VEL: // en caso de detectar max. vel.
46              set_vel.linear.x = 0.75; //la vel. lineal se vuelve máxima
47              set_vel.angular.z = 0; //la vel angular se vuelve cero
48              break;
49          case MED_VEL: // en caso de detectar med vel
50              set_vel.linear.x = 0.5;//la vel. lineal toma un valor intermedio
51              set_vel.angular.z = 0; //la vel angular se vuelve cero
52              break;
53          case MIN_VEL: // en caso de detectar min vel
54              set_vel.linear.x = 0.25; //la vel. lineal toma un valor mínimo
55              set_vel.angular.z = 0; //la vel angular se vuelve cero
56              break;
57          default: // en caso de encontrar otro objeto
58              set_vel.linear.x = 0.25; //discurre a una velocidad baja avanzando
59              set_vel.angular.z = 0; //la vel angular se vuelve cero
60          }
61          action_pub.publish(set_vel); // se establece la velocidad determinada
62      }
63      else
64      {
65          // si no detecta nada
66          set_vel.linear.x = 0.25; //discurre a una velocidad mínima
67          set_vel.angular.z = 0; //no gira
68          action_pub.publish(set_vel); //establece esta velocidad
69      }
70  }
71
72  int main(int argc, char **argv)
73  {
74
75      ros::init(argc, argv, "action_controller"); //inicia el nodo action_controller
76      ros::NodeHandle n("~");
77      ros::Rate loop_rate(50); // se espera unos cuantos ciclos
78      ros::Subscriber sub = n.subscribe("/objects", 1, objectCallback);
79      //suscribe el nodo al tópico donde se publican las detecciones
80      action_pub = n.advertise<geometry_msgs::Twist>("/cmd_vel", 1);
81      //lo pone a publicar en el tópico que modifica la velocidad
82      set_vel.linear.x = 0; //hace cero todas las velocidades
83      set_vel.linear.y = 0; //hace cero todas las velocidades
84      set_vel.linear.z = 0; //hace cero todas las velocidades
85      set_vel.angular.x = 0; //hace cero todas las velocidades
86      set_vel.angular.y = 0; //hace cero todas las velocidades
87      set_vel.angular.z = 0; //hace cero todas las velocidades
88      while (ros::ok()) //comprueba que la tarea maestra sigue funcionando
89      {
90          ros::spinOnce();
91          loop_rate.sleep(); //espera hasta vovler al inicio.
92      }
93  }
```

B. 3. Nodo encargado de definir la posición del robot: *drive_controller_node.cpp*

```cpp
#include <ros/ros.h> //bibliotechas necesarias
#include <geometry_msgs/PoseStamped.h> //bibliotechas necesarias
#include <tf/transform_broadcaster.h> //bibliotechas necesarias

tf::Transform transform; //definición de mensaje que almacena transformaciones
tf::Quaternion q; //definición del mensaje que almacena rotaciones

void pose_callback(const geometry_msgs::PoseStampedPtr &pose)
{
    q.setX(pose->pose.orientation.x); //inclusión de posición en eje x del robot
    q.setY(pose->pose.orientation.y); //inclusión de posición en eje y del robot
    q.setZ(pose->pose.orientation.z); //inclusión de posición en eje z del robot
    q.setW(pose->pose.orientation.w); //inclusión de orientación tras rotación

    transform.setOrigin(tf::Vector3(pose->pose.position.x, pose->pose.position.y, 0.0));
    //definición completa del mensaje de posición lineal en tf

    transform.setRotation(q); //definición del mensaje de rotación en q

    br.sendTransform(tf::StampedTransform(transform, ros::Time::now(), "odom", "base_link")
    //envío de ambos mensajes al sistema
}

int main(int argc, char **argv)
{
   ros::init(argc, argv, "drive_controller"); //inicialización del nodo
   ros::NodeHandle n("~");
   ros::Subscriber pose_sub = n.subscribe("/pose", 1, pose_callback);
   // suscripción del nodo al tópico de donde recibe la posición

   ros::Rate loop_rate(100); //espera de varios ciclos
   while (ros::ok()) //comprobación del correcto funcionamiento
   {
      ros::spinOnce();
      loop_rate.sleep(); //vuelta al inicio
   }
}
```

B. 4. Archivo para iniciar el mapeado de entornos: *Maping.launch*

```
<launch>
//necesario en cada archivo launch al empezar y acabar

 <!-- ROSbot 2.0 -->

 <include file="$(find rosbot_ekf)/launch/all.launch">
 //se inicia el nodo del filtro de kalmann extendido

  <include file="$(find rplidar_ros)/launch/rplidar.launch" />
  // se inicia el nodo que hace funcionar al lidar del dispositivo

  </group>

  <node pkg="tf" type="static_transform_publisher" name="laser_broadcaster"
  args="0 0 0 3.14 0 0 base_link laser 100" />
  // se lanza el nodo tf que nos indica la posición del robot en cada momento

  <node pkg="teleop_twist_keyboard" type="teleop_twist_keyboard.py"
  name="teleop_twist_keyboard" output="screen"/>
  // se ejecuta el nodo que nos permite controlar el dispositivo mediante el teclado

  <node pkg="rviz" type="rviz" name="rviz"/>
  // se abre el programa rviz

  <node pkg="gmapping" type="slam_gmapping" name="gmapping">
  //se inicia el nodo gmapping encargado de plasmar en un mapa lo captado por el Lidar

        <param name="base_frame" value="base_link"/> //se define la base
        <param name="odom_frame" value="odom" /> //se define la odometría
        <param name="delta" value="0.1" /> // se define el incremento en la posición
   </node>

</launch>
//necesario en cada archivo launch al empezar y acabar
```

B. 5. Congiuración de parámetros comunes para el mapeado de obstáculos: *costmap_common_params.yaml*

```
obstacle_range: 6.0 //rango de definición de obstáculos

raytrace_range: 8.5 //rango para considerar libre un area

footprint: [[0.12, 0.14], [0.12, -0.14], [-0.12, -0.14], [-0.12, 0.14]]
//distancia minima para detectar una colisión

map_topic: /map //lugar en el que plasmar lo explorado

subscribe_to_updates: true //define la actualización automática
                           //del estado del mapa

observation_sources: laser_scan_sensor //tópico del que se toma información

laser_scan_sensor: {sensor_frame: laser_frame, data_type: LaserScan,
topic: scan, marking: true, clearing: true}
//define las propiedades con las que trabaj el sensor

global_frame: map //definición de los marcos del mapa

robot_base_frame: base_link //definición del marco del robot

always_send_full_costmap: true //define la publicación de
                               //toda la información del mapa
```

B. 6. Congiuración de parámetros locales para el mapeado de obstáculos: *costmap_local_params.yaml*

```
local_costmap: //define que los siguientes parametros son para el mapeado local

update_frequency: 5 //define la frecuencia de cálculo de obstaculos

publish_frequency: 5 //define la frecuencia con la que se publica en el tópico

transform_tolerance: 0.25 //define el tiempo máximo en segundos válido para aceptar
                          una transformación

  static_map: false //define si el mapa cambia con el tiempo

  rolling_window: true //define si el mapa sigue la osición del robot

  width: 3 //define el tamaño del mapa
  height: 3 //define el tamaño del mapa

  origin_x: -1.5 //define la posición de la esquina inferior izqda. del mapa
  origin_y: -1.5 //define la posición de la esquina inferior izqda. del mapa

  resolution: 0.1 //define el tamaño de las celdas
  inflation_radius: 0.6 //define la distancia radial a partir de la cual
                        se considera cercano un obstaculo
```

B. 7. Congiuración de parámetros globales para el mapeado de obstáculos: *costmap_global_params.yaml*

```
global_costmap: //define que los siguientes parametros son para el mapeado global

  update_frequency: 2.5 //define la frecuencia de cálculo de obstaculos

  publish_frequency: 2.5 //define la frecuencia con la que se publica en el tópico

  transform_tolerance: 0.5 //define el tiempo máximo en segundos válido para aceptar
                           una transformación

  width: 15 //define el tamaño del mapa
  height: 15 //define el tamaño del mapa

  origin_x: -7.5 //define la posición de la esquina inferior izqda. del mapa
  origin_y: -7.5 //define la posición de la esquina inferior izqda. del mapa

  static_map: true //define si el mapa cambia con el tiempo

  rolling_window: true //define si el mapa sigue la osición del robot

  inflation_radius: 2.5 //define la distancia radial a partir de la cual
                        se considera cercano un obstaculo
  resolution: 0.1 //define el tamaño de las celdas
```

B. 8. Parámetros para la planificación de trayectorias: *trajectory_planner.yaml*

```
TrajectoryPlannerROS: //expresa que los parametros definidos a continuación
                        se refieren al planificador de trayectorias de ROS

  max_vel_x: 0.2 //define la máxima velocidad de desplazamiento
  min_vel_x: 0.1 // define la mínima velocidad de desplazamiento
  max_vel_theta: 0.35 //define la máxima velocidad radial positiva
  min_vel_theta: -0.35 //define la maxima velocidad radial negativa
  min_in_place_vel_theta: 0.25 //define la minima velocidad radial
                                 llevada a cabo sobre si mismo

  acc_lim_theta: 0.25 //define los valores máximos de aceleración
  acc_lim_x: 2.5 //define los valores máximos de aceleración
  acc_lim_Y: 2.5 //define los valores máximos de aceleración

  holonomic_robot: false //define si el robot es holonómico

  meter_scoring: true //define si los argumentos de la función de
                        obstáculos se definen en metros

xy_goal_tolerance: 0.15 //define los límites para considerar alcanzado el destino
yaw_goal_tolerance: 0.25 //define los límites para considerar alcanzado el destino
```

B. 9. Archivo que ejecuta la planificación de trayectorias: *Path_planning.launch*

```
<launch> //necesario en todos los archivos launch

  <!-- ROSbot 2.0 -->

    <include file="$(find rosbot_ekf)/launch/all.launch">
     //inicia el nodo del filtro de kalmann extendido

    <include file="$(find rplidar_ros)/launch/rplidar.launch" />
    //inicia el nodo del Lidar en el robot

  </group>

  <node pkg="tf" type="static_transform_publisher" name="laser_broadcaster"
        args="0 0 0 3.14 0 0 base_link laser 100" />
 //inicia el nodo tf encargado de determinar la posición del robot en cada moemnto

  <node pkg="teleop_twist_keyboard" type="teleop_twist_keyboard.py"
        name="teleop_twist_keyboard" output="screen"/>
 //inciia el nodo del control del robot mediante el teclado

  <node pkg="rviz" type="rviz" name="rviz"/>
 //incia el programa rviz

  <node pkg="gmapping" type="slam_gmapping" name="gmapping">
    <param name="base_frame" value="base_link"/>
    <param name="odom_frame" value="odom" />
    <param name="delta" value="0.1" />
  </node>
 //incia y configura el nodo de slam_mapping

  <node pkg="move_base" type="move_base" name="move_base" output="screen">
    <param name="controller_frequency" value="10.0"/>
    <rosparam file="$(find tutorial_pkg)/config/costmap_common_params.yaml"
              command="load" ns="global_costmap" />
    <rosparam file="$(find tutorial_pkg)/config/costmap_common_params.yaml"
              command="load" ns="local_costmap" />
    <rosparam file="$(find tutorial_pkg)/config/local_costmap_params.yaml"
              command="load" />
    <rosparam file="$(find tutorial_pkg)/config/global_costmap_params.yaml"
              command="load" />
    <rosparam file="$(find tutorial_pkg)/config/trajectory_planner.yaml"
              command="load" />
  </node>
 //incia y configura el nodo de movimiento automático de la base. Además,
 //carga las configuraciones explicadas previamente

</launch> //necesario en todos los archivos launch
```

B. 10. Parámetros para la exploración autónoma: *exploration.yaml*

```
robot_base_frame: base_link
// determina la posición del robot a traves de su base

costmap_topic: map
//especifica el tópico del que se coge información
para rellenar el mapa

costmap_updates_topic: map_updates
//necesario cuando el mapa no publica habitualmente
actualizaciones

visualize: true
//especifica si poner o no fronteras visibles

planner_frequency: 0.33
//frecuencia de planificación de actuaciones
tras la detección de nuevas fornteras

progress_timeout: 30.0
//tiempo que pasa hasta que el robot abandona su labor

potential_scale: 3.0
//se utiliza para poderar fronteras

orientation_scale: 0.0
//se utiliza para ponderar fronteras

gain_scale: 1.0
//se utiliza para ponderar fronteras

transform_tolerance: 0.3
//tolerancia de transformación al situar al robot

min_frontier_size: 0.75
//minimo valor para considerar un objeto como una frontera.
```

B. 11. archivo que ejecuta la exploración autónoma: *exploration.launch*

```
<launch>  //necesario en todos los archivos launch


  <!-- ROSbot 2.0 -->

    <include file="$(find rosbot_ekf)/launch/all.launch">
          //inicia el nodo del filtro de kalmann extendido

    <include file="$(find rplidar_ros)/launch/rplidar.launch" />
        //inicia el nodo del Lidar en el robot

  </group>

  <node pkg="tf" type="static_transform_publisher" name="laser_broadcaster"
        args="0 0 0 3.14 0 0 base_link laser 100" />
//inicia el nodo tf encargado de determinar la posición del robot en cada moemnto

  <node pkg="teleop_twist_keyboard" type="teleop_twist_keyboard.py"
        name="teleop_twist_keyboard" output="screen"/>
 //inciia el nodo del control del robot mediante el teclado

  <node pkg="gmapping" type="slam_gmapping" name="gmapping_node" output="log">
    <param name="base_frame" value="base_link" />
    <param name="odom_frame" value="odom" />
    <param name="delta" value="0.01" />
    <param name="xmin" value="-5" />
    <param name="ymin" value="-5" />
    <param name="xmax" value="5" />
    <param name="ymax" value="5" />
    <param name="maxUrange" value="5" />
    <param name="map_update_interval" value="1" />
    <param name="linearUpdate" value="0.05" />
    <param name="angularUpdate" value="0.05" />
    <param name="temporalUpdate" value="0.1" />
    <param name="particles" value="100" />
  </node>
//incia y configura el nodo de slam_mapping

  <node pkg="move_base" type="move_base" name="move_base" output="screen">
    <param name="controller_frequency" value="10.0"/>
    <rosparam file="$(find tutorial_pkg)/config/costmap_common_params.yaml"
              command="load" ns="global_costmap" />
    <rosparam file="$(find tutorial_pkg)/config/costmap_common_params.yaml"
              command="load" ns="local_costmap" />
    <rosparam file="$(find tutorial_pkg)/config/local_costmap_params.yaml"
              command="load" />
    <rosparam file="$(find tutorial_pkg)/config/global_costmap_params.yaml"
              command="load" />
```

```
    <rosparam file="$(find tutorial_pkg)/config/trajectory_planner.yaml"
              command="load" />
  </node>
 //incia y configura el nodo de movimiento automático de la base. Además,
 //carga las configuraciones explicadas previamente

  <node pkg="explore_lite" type="explore" respawn="true" name="explore" output="screen">
    <rosparam file="$(find exploration_pkg)/config/exploration.yaml" command="load" />
  </node>
//incia y configura el nodo de exploración autónoma. Además, carga
//la configuración explicada previamente.

  <node pkg="rviz" type="rviz" name="rviz"/>
 //incia el programa rviz

</launch>  //necesario en todos los archivos launch
```

B. 12. Ejemplo de *CMakeLists.txt*

```
cmake_minimum_required(VERSION 3.0.2)
//versión del sistema utilizada

project(drive_controller)
//nombre del proyecto al cual pertenece

add_compile_options(-std=c++11)
//configuración requerida siempre

find_package(catkin REQUIRED COMPONENTS
  roscpp
)
//paquetes necesarios para su desarrollo

catkin_package(
#  INCLUDE_DIRS include
#  LIBRARIES drive_controller
#  CATKIN_DEPENDS roscpp
#  DEPENDS system_lib
)
//definición de los elementos revisados al
actualizar el paquete

include_directories(
# include
  ${catkin_INCLUDE_DIRS}
)
//inclusión de directorios

add_executable(${PROJECT_NAME}_node src/drive_controller_node.cpp)
//declaración del nodo ejecutable

target_link_libraries(${PROJECT_NAME}_node
  ${catkin_LIBRARIES}
)
//inclusión de bibliotecas necesarias
```

REFERENCIAS

1 Husarion Manuals. (2021) Autonomous Mobile Robots [Online]. https://husarion.com/manuals/

2 Blog Medium. (2018) Detección de objetos en Tiempo Real con python y OpenCV [Online]. https://medium.com/@di3cruz/detecci%C3%B3n-de-objetos-en-tiempo-real-con-python-y-opencv-e760fdaad58e

3 Introlab. (2020) wiki de la función find_object. [Online] http://introlab.github.io/find-object/

4 Blog Pocket Link. (2020) Ejemplo funcionamiento LIDAR en iphone. [Online] https://www.pocket-lint.com/es-es/tabletas/noticias/apple/151476-que-es-lidar-ipad-por-que-arkit-medida

5 Slamtec. (2021) proveedor de sensores LIDAR del ROSbot2.0. [Online] http://www.slamtec.com/en/Lidar/A2

6 Wiki de ROS. (2017) información sobre el nodo "tf". [Online] http://wiki.ros.org/tf

7 Ubuntu (2018) información y descarga del sistema operativo 18.04 LTS. [Online]. https://ubuntu.com/#community

8 Wiki de ros melodic morenia (2018) información de descarga e instalación del software. [Online].

http://wiki.ros.org/melodic

9 Biblioteca de proyectos fing (2017) Integración y funcionamiento de ROS en un S.O. [Online].

https://www.fing.edu.uy/inco/proyectos/butia/mediawiki/index.php/Integracion_ros_butia

10 Husarion Community (2021) Blog de preguntas y respuestas sobre el funcionamiento de los dispositivos. [Online]. https://community.husarion.com/

11 Wiki de ROS (2021) biblioteca de nodos y tópicos. [Online]. http://wiki.ros.org/es

12 Husarion tutorials (2021) tutoriales de manejo y programación del ROSbot2.0 [Online]. https://husarion.com/tutorials/

13 Github (2021) foro de preguntas y respuestas a cerca de software. [Online]. https://github.com/

14 Marve (2015) Empresa de señalización y seguridad. [Online]. http://www.marve.es/

15 Visión artificial en Wikipedia (2021) Fundamentos de la visión artificial. [Online]. https://n9.cl/k43m1

16 Research gate. Image Matching using SIFT, SURF, BRIEF and ORB. [Online]. https://n9.cl/t9io5

17 Wiki de ROS nodo de RPLIDAR. (2017) información y desarrollo del nodo. [Online] http://wiki.ros.org/rplidar

18 Wiki de ROS nodo de Gmapping. (2017) información y desarrollo del nodo. [Online] http://wiki.ros.org/gmapping

19 TFG de D. Manuel Mora Nieto. (2018) Integración de sistema laser en la navegación de robot móvil. [Online] https://idus.us.es/bitstream/handle/11441/84489/TFG-1974-MORA.pdf?sequence=1&isAllowed=y

20 Blog OpenWebinars. (2017) ¿Qué es Ros? (Robot Operating System. [Online]. https://openwebinars.net/blog/que-es-ros/

21 Wikipedia Robot Operating System (2021). Información sobre funcionamiento y estructura. [Online].

https://es.wikipedia.org/wiki/Robot_Operating_System

22 Wiki de ROS Rviz. (2016). Funcionamiento de la herramienta [Online]. http://wiki.ros.org/rviz

23 Wiki de ROS RQT_Graph (2016). Funcionamiento de la herramienta [Online]. http://wiki.ros.org/rqt_graph

24 Wikipedia SLAM navigation (2021). Información sobre funcionamiento y precedentes. [Online]. https://es.wikipedia.org/wiki/Localizaci%C3%B3n_y_modelado_simult%C3%A1neos

25 Canal Uned Oscar Reinoso (2016). Nociones básicas de la navegación SLAM. [Online]. https://canal.uned.es/uploads/material/Video/49999/Presentaci__n_Oscar_Reinoso.pdf

26 Introducción a ORB (2014). Nociones básicas del algoritmo ORB. [Online] https://ichi.pro/es/introduccion-a-orb-orientado-fast-y-rotated-brief-72709114183887

27 Introducción a SURF (2017). Nociones básicas sobre el algoritmo SURF. [Online] https://es.wikipedia.org/wiki/SURF

28 Introducción a la DWA (2012). Nociones Básicas sobre la evitación de obstáculos. [Online] https://en.wikipedia.org/wiki/Dynamic_window_approach