ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADO EN INGENIERÍA INFORMÁTICA

INTEGRACIÓN Y ADAPTACIÓN DE UN NAVEGADOR REACTIVO 3D EN LA ARQUITECTURA ROBÓTICA ROS

INTEGRATION AND ADAPTATION OF A 3D REACTIVE NAVIGATOR IN THE ROS ROBOTIC ARCHITECTURE

Realizado por:

Mario Alejandro Rueda Castro

Tutorizado por:

Javier González Jiménez

Javier González Monroy

Departamento:

Ingeniería de Sistemas y Automática

UNIVERSIDAD DE MÁLAGA

MÁLAGA, FEBRERO DE 2019

Fecha defensa:

El secretario del tribunal

Resumen

Debido a sus altos costes, el difícil acceso a componentes, así como la dificultad de los problemas de ingeniería que surgen en ella, no resulta viable dedicarse al campo de la Robótica sin atajarla aunando esfuerzos e inversión entre muchos profesionales de la investigación alrededor del mundo. El *open source* es fundamental más que nunca como filosofía para la transmisión de los últimos avances en cuanto a software robótico. ROS ofrece la plataforma principal, más conocida, con más paquetes y documentación y la posibilidad de contactar con profesionales. Uno de los problemas principales de la Robótica es el problema de la navegación hacia una meta. Existen multitud de estrategias para llevarla a cabo, pero en general, se sigue un enfoque *divide y vencerás* en el que cada subproblema del mismo se resuelve por separado. El presente TFG tiene como objetivo integrar una aplicación de navegación reactiva, esto es, sensorial, sin memoria, con el *stack* tecnológico de ROS dedicado a la navegación. Además, este trabajo pretende ofrecer una revisión bibliográfica de los algoritmos de navegación reactiva más relevantes y una fase de experimentación con unos resultados. Finalmente, el planteamiento de cómo extender el motor reactivo para soportar 3 dimensiones estructuradas en bloques.

Palabras clave: Robótica móvil, ROS, Navegación reactiva, MRPT, PTG, evasión de colisones.

Abstract

Due to their high prices, hard to reach components and complexity in the engineering problems surrounding them, it is not possible to dedicate to Robotics without joining in the efforts and investment of millions of dedicated proffesionals around the globe. Open Source is a necessity now more than ever as a way to exchange the new directions surrounding robotic software. ROS offers the most well-known platform, community, loaded with documented code and experts. The navigation problem is one of the main ones in Robotics. There are a lot of algorithms trying to solve it but none of them offers a robust, general solution. Generally it is tackled using divide and conquers approach. This project has the goal of integrate a reactive navigation application, (i.e. sensible, memoryless) with the tech stack in ROS. This project tries to offer a reference review of the most relevant reactive navigation algorithms. It also tries to offer a comparative between the subject and the reviewed ones. In addition, the outlines of how to enable 3 dimensional navigation using the integrated software.

Keywords: Mobile Robotics, ROS, Reactive Navigation, MRPT, PTG, Collision evasion.

Índice general

Índice de figuras

Capítulo 1

Introducción

1.1 Motivación

El concepto de vehículo autónomo hace referencia a la capacidad de un vehículo no tripulado de planificar una ruta de navegación entre un punto origen y otro destino, y ejecutar dicho plan sin intervención humana. En la actualidad este concepto se puede aplicar a gran variedad de vehículos como es el caso de los automóviles y los tan prometedores prototipos de coches autónomos [4, 5], aviones, drones y en general los diferentes tipos de vehículos aéreos no tripulados [6], así como a la gran y heterogénea variedad de robots móviles [7, 8].

Entre los muchos retos aún existentes para conseguir vehículos completamente autónomos a nivel comercial, el problema de la navegación y la evitación de obstáculos (véase Fig. 1.1) es uno de los problemas fundamentales aún por resolver de forma robusta y generalizada. La gran complejidad de esta tarea radica en la necesidad de calcular y ejecutar una ruta de navegación basándose única y exclusivamente en un conocimiento previo del entorno (esto es un mapa), y de la información disponible de un conjunto de sensores a bordo del propio vehículo. Dada las condiciones dinámicas del entorno en los que los vehículos autónomos tienen que actuar, la planificación de los movimientos a ejecutar debe realizarse en tiempo real para poder adaptarse a los posibles cambios del entorno, característica que incrementa notablemente la complejidad de los algoritmos a desarrollar. En este contexto, cabe mencionar que en las últimas décadas se han conseguido importantes avances mediante el diseño de estrategias que aúnan eficiencia y robustez a la hora de resolver de forma independiente algunos de los problemas derivados de la navegación autónoma, entre ellos la percepción de obstáculos en el entorno [9, 10], la estimación del movimiento del robot [11, 12], la evitación de obstáculos durante la navegación [3], o la generación de planes de navegación atendiendo a diferentes sistemas sensoriales [13].

Los algoritmos de navegación actuales pueden ser clasificados en tres grandes grupos atendiendo a la información utilizada para completar la tarea en cuestión:

- Planificadores deliberativos: introducidos por R. Brooks [14] en los años 80, consisten en trazar una ruta estática y segura desde el lugar donde se encuentra el robot hasta el destino deseado. Están diseñados para aquellas aplicaciones en los que se dispone de un conocimiento completo del entorno (mapa estático de obstáculos) y en los que no hay componente dinámica. Naturalmente, estos planificadores no pueden adaptarse a cambios del entorno, por lo que sólo son utilizados en ambientes industriales muy controlados.

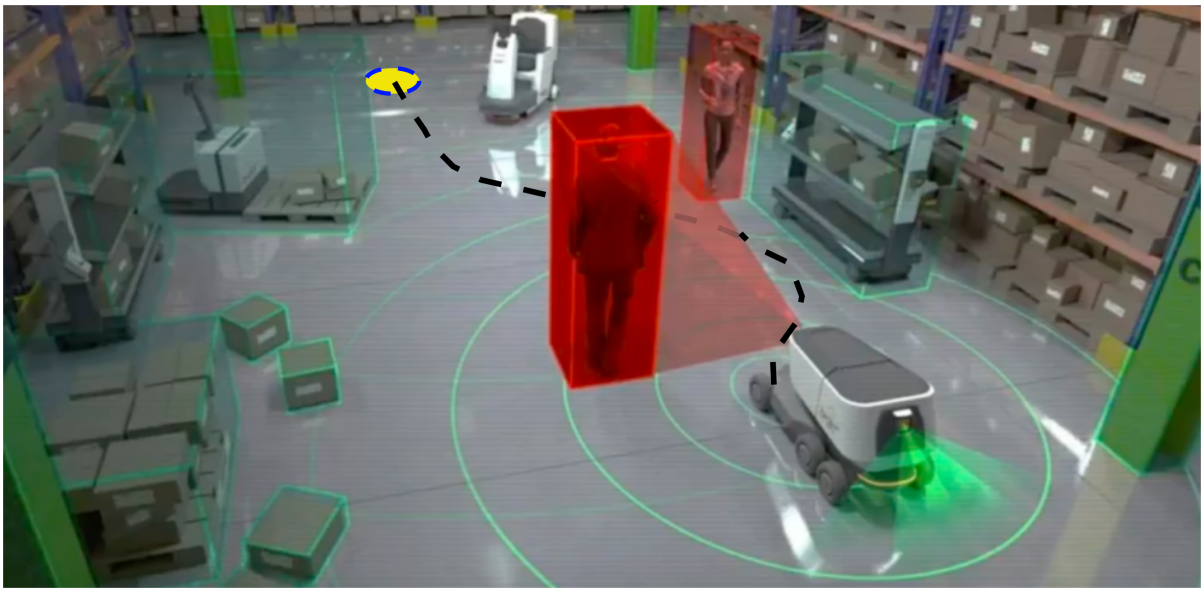


Figura 1.1: Ilustración de la navegación autónoma. El vehículo, en este caso un robot industrial, planea una ruta de navegación atendiendo al conocimiento previo del entorno, a su posición actual, a la meta que debe alcanzar y a los obstáculos en el entorno (percibidos mediante diferentes sistemas sensoriales).

- Planificadores reactivos: consisten en la generación de trayectorias de navegación en base únicamente a la información sensorial. Están especialmente diseñados para hacer frente a entornos altamente dinámicos donde una planificación deliberativa no tendría éxito. No obstante, dado que este tipo de planificadores no explota el conocimiento previo o parcial del entorno, su aplicación en entornos complejos está muy limitado. Reseñable es el caso de las tortugas de Grey Walter [15].

- Planificadores híbridos: presentan la posibilidad de explotar tanto una visión global del entorno, como un comportamiento reactivo atendiendo a las medidas sensoriales. El principal problema de esta estrategia reside en el diseño de la cooperación entre ambos planificadores (deliberativo y reactivo) para combinarlos en un nuevo planificador que permita resolver el problema en cuestión.

En este trabajo nos centramos en los planificadores híbridos, y más concretamente en la componente reactiva de dichos planificadores para el caso de sistemas robóticos basados en la ampliamente aceptada y distribuida arquitectura ROS [16] (de sus siglas en inglés: Robotic Operating System). ROS integra diferentes planificadores reactivos dentro de su paradigma de navegación híbrida, ofreciendo una interfaz común que permite el intercambio y/o desarrollo de nuevos planificadores reactivos. No obstante, todas las opciones disponibles tienen una limitación común, el uso de generadores de trayectorias que solo contemplan trayectorias circulares (resultado de aplicar comandos de velocidad constantes durante un intervalo de tiempo) para la generación de comandos de velocidad del robot. Aunque versátil, este conjunto de trayectorias no abarca la gran variedad de movimientos que un robot puede realizar, siendo, en ocasiones, el motivo por la que un vehículo autónomo no puede realizar satisfactoriamente la tarea de navegación requerida.

El objetivo principal perseguido en este trabajo es estudiar e integrar en la arquitectura robótica ROS, un modelo de planificador reactivo que permita mejorar las características y

capacidades de navegación de los robots móviles que emplean ROS. Para ello se propone la integración del planificador reactivo basado en el espacio parametrizado de trayectorias [17, 18] implementado en la librería robótica MRPT [19] (de sus siglas en inglés: Mobile Robot Programming Toolkit). Las ventajas fundamentales de este planificador reactivo frente a los existentes en ROS son dos: Por un lado, las trayectorias generadas no se limitan a movimientos circulares a trozos, sino que extiende el abanico a otros tipos de trayectorias más complejas y versátiles, siempre compatible con la cinemática del robot. Por otro lado, este navegador reactivo tiene la capacidad de considerar la diferente geometría del robot a distintas alturas, por lo que aprovecha la información proporcionada por sensores 3D como cámaras y escáner de distancia.

Finalmente, cabe mencionar que este proyecto extiende el trabajo fin de máster titulado: "*Implementación de un sistema de navegación reactivo 3D para robots en el entorno ROS*", presentado dentro del departamento de Ingeniería de Sistemas y Automática de la Universidad de Málaga en 2018.

1.2 Objetivos

El objetivo de este trabajo fin de grado es integrar dentro de la arquitectura robótica ROS el novedoso sistema de navegación reactiva implementado en la librería MRPT, permitiendo así extender la generación de trayectorias a otros espacios más allá de las trayectorias circulares. De forma más detallada, los objetivos específicos de este trabajo son comentados a continuación:

- **Estudio del meta-paquete ROS encargado de la navegación**: Dado que el objetivo principal es integrar un nuevo planificador reactivo en ROS, se hace necesario estudiar y analizar el meta-paquete $move\_base$ para analizar la interfaz que ofrece a los nuevos planificadores reactivos, así como para comprender en detalle como cooperan los planificadores deliberativos y reactivos en dicha arquitectura.

- **Estudio del planificador reactivo basado en el espacio parametrizado de trayectorias**: Para poder integrar este nuevo planificador en la arquitectura ROS, es imprescindible comprender su funcionamiento y determinar cuales son los datos de entrada/salida con los que trabaja.

- **Diseño e implementación del plugin de navegación local**: El paquete move_base ofrece la posibilidad de integrar nuevos planificadores reactivos mediante la implementación de plugins. En este trabajo diseñaremos e implementaremos un plugin para integrar el reactivo basado en el espacio parametrizado de trayectorias.

- **Análisis y comparativa de funcionamiento:** Una vez integrado el nuevo planificador reactivo, se llevará a cabo un estudio experimental, tanto en simulación como en casos reales, para analizar las diferencias de funcionamiento y rendimiento entre diferentes navegadores reactivos.

- **Soporte de información en tres dimensiones**: Una de las ventajas adicionales de considerar el planificador reactivo de la MRPT es su capacidad de soportar información 3D. Atendiendo a la geometría del robot, este planificador permite analizar diferentes conjuntos de trayectorias (atendiendo a la altura) y seleccionar siempre una trayectoria que resulte segura. Para ello, se considerarán en este punto cámaras RGB-D (que aporten información sensorial del entorno en 3D) y una descripción 3D del robot.

1.3 Estructura del Documento

En adelante documento se ha estructurado en capítulos de la siguiente manera:

- En el Capítulo 2, se exponen las dos tecnologías principales usadas en este proyecto. La finalidad última de este apartado es definir ciertos conceptos a los que se hacen referencia durante el escrito.
- En el Capítulo 3, se presenta una revisión del estado del arte en los campos de control de movimiento robótico en ROS, así como de los algoritmos de navegación reactiva más populares. Además, se introduce el fundamento del algoritmo objeto de la integración software de este proyecto.
- En el Capítulo 4, se describe la contribución de este trabajo, haciendo especial énfasis en la implementación del software necesario y describiéndose en profundidad los elementos que lo conforman así como las dificultades técnicas encontradas.
- En el Capítulo 5, se describe el proceso de validación experimental, tanto en simulación como con un robot móvil en un entorno de laboratorio, así como se discuten los resultados obtenidos.
- Finalmente, en el Capítulo 6, se resumen las conclusiones fruto de haber realizado este trabajo fin de grado y se establecen diferentes lineas de trabajo futuro para su continuación.

Capítulo 2

Tecnologías utilizadas

La finalidad de este capítulo es introducir las tecnologías y términos que en el presente trabajo se utilizan y a los que se harán referencia a lo largo del escrito.

2.1 ROS

2.1.1 Ecosistema

El Sistema Operativo Robótico, Robot Operating System, ROS a partir de ahora, es un conjunto de paquetes y librerías que forman un framework de programación y a la vez un entorno que sirve de vía de comunicación entre sistemas operativos y hardware robótico.

Fue creado por Open Robotics con la finalidad de diseñar el Robot PR-1 y sentar base en el desarrollo software dirigido a dispositivos robotizados.

Una de las mayores bazas que ROS posee frente a sus alternativas es lo que lo rodea. Y es que, ROS no es solamente un conjunto de paquetes de aplicaciones que emulan un sistema operativo para los componentes de robots, se compone además de una enorme comunidad, una abundante cantidad de paquetes que además es compatible con el hardware robótico más utilizado, inversores que están detrás del proyecto y la reputación obtenida de haber sido implantado en numerosos proyectos dentro del ámbito académico y comercial.

Las características principales que hacen de ROS un gran entorno para desarrollar software robótico se sustentan en el hecho de que la funcionalidad que debe ofrecer un robot es más compleja y amplia que la que se pide de otro tipo de sistema multitarea, como un ordenador personal o un dispositivo móvil.

Comunidad

Típicamente, lo que destaca a los robots de los demás sistemas multiproceso es cómo se relacionan con el mundo real (awareness) y su capacidad de tener presencia en el mismo (embodiment). Así, al tratarse de problemas de gran envergadura, es necesario el esfuerzo de muchos expertos para resolverlos. Esto conllevaría grandes costes si se quisiese realizar en el seno de una sola empresa. De esta forma, es necesario que ROS se fundamente en el código abierto. Siendo así la única forma de tener a un gran número de especialistas dedicados al avance del framework y a la resolución de problemas de tan elevada dificultad.

Las plataformas más importantes en las que se se lleva a cabo el intercambio de información en la comunidad ROS son: `wiki.ros.org`. Donde se encuentra la información pertinente a los repositorios ROS a nivel de usuario. Suelen incluir amplias explicaciones del funcionamiento de los mismos. Incluso tutoriales de aprendizaje y fundamento teórico. Es una de las razones por las que ROS triunfa, accesibilidad. `answers.ros.org`. Siguiendo el modelo stackexchange se trata de un portal en el que plantear y responder dudas relacionadas con ROS. Existe karma y reputación. `discourse.ros.org` - `ros-users@lists.ros.org`. Los portales de discusión de ROS, anteriormente se utilizaba el buzón de correo pero ahora existe el foro `discourse.ros.org`, donde los usuarios pueden debatir sobre temas derivados de la experiencia de usuario ROS.

Código

El código ROS se distribuye bajo la licencia estándar Berkeley, por lo que su uso se permite a nivel comercial o no comercial. [1] Aproximadamente se encuentran en ROS unos 3000 repositorios en total.[2] ROS es compatible en mayor o menor medida con diversos lenguajes de programación. Los más utilizados y mejor integrados en el ecosistema son C/C++ y Python, pero lenguajes como Java, Lisp y en menor medida Javascript, PHP, o incluso Haskell también lo son, aunque con mayor dificultad de adaptación del usuario.

Nivel de utilización

ROS resulta indudablemente apropiado para la investigación en el ámbito académico al ofrecer una plataforma accesible y una vía de comunicación entre investigadores. Universidades en todas partes del mundo abandonan o adaptan su trabajo acumulado con el tiempo para acercarse al ecosistema ROS. Por ejemplo, la universidad de Oregón es la que hospeda mucho de los sitios del ecosistema ROS. El MIT utiliza ROS en muchos de sus proyectos, contando con un famoso paquete para el uso de cámaras Kinect dentro del sistema. . En la Universidad Politécnica de Barcelona, ROS se emplea integrándose con diferentes frameworks sobre plataformas robóticas comerciales. Este trabajo de fin de grado es ejemplo de integración de ROS con un framework ampliamente usado (MRPT)

Sin embargo, también está implantado en múltiples proyectos comerciales. Ejemplo de esto es la integración de múltiples modelos de robots comerciales con el framework Moveit dentro del entorno ROS.[3]

Incluso, desde el ministerio de defensa de Estados Unidos, se promueve que todos los proyectos de su instituto de investigación tengan compatibilidad con ROS. Asimismo, la compañía detrás del desarrollo ROS, Open Robotics, cuenta con el patrocinio de empresas conocidas. (Fig. 2.1).

2.1.2 Terminología

De forma breve, el entorno ROS consiste en un sistema de paquetes que se instancian durante ejecución como nodos. Estos nodos establecen su comunicación bien entre ellos en un sistema

[1] La lista de paquetes de ROS según distribución (más adelante se definirá esto) se encuentra en: `https://github.com/ros/rosdistro`

[2] `http://rosindex.github.io/stats/`

[3] `https://moveit.ros.org/robots/`

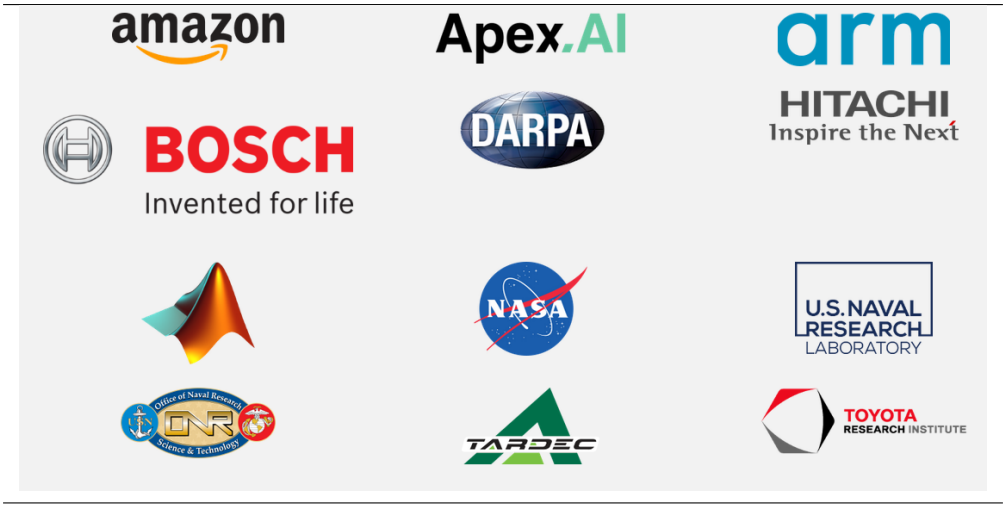


Figura 2.1: Lista de empresas patrocinadoras detrás de ROS.

de comunicación p2p o mediante mensajes enviados hacia un servidor. Estos mensajes no son otra cosa que contenedores xml, así se asegura la integración de tecnologías muy diferentes.

Paquete

Un paquete ROS es simplemente un directorio que contiene un archivo *manifiesto*. Dependiendo de la versión de catkin que se utilice, el nombre es diferente. Para catkin-0.7.5, su nombre es *package.xml*. Este archivo contiene la descripción necesaria del paquete y la que lo identifica para el nodo maestro. También se utiliza para especificar dependencias que el código requiera para su compilación y ejecución. Generalmente los paquetes ROS son utilizados para instanciar nodos, crear librerías o portar archivos de configuración. Por esto, además del manifiesto, el paquete contiene siempre un archivo *CMakeLists.txt*, que se utiliza para vincular las diferentes dependencias del paquete con el mismo. Un **meta-paquete** es un paquete cuya función es contener y organizar otros paquetes.

catkin

Catkin es el sistema de compilación de paquetes que ROS trae por defecto. Se trata de un sistema que mediante cmake compila los paquetes según se indica en sus CMakeLists.txt. Se trata de un sistema muy amigable que permite reducir el esfuerzo para lograr la compilación. Al crear un paquete se emplea la instrucción **catkin_create_pkg** y a partir de ahí se modifica el paquete según las dependencias del proyecto y la compilación se asegura.

Nodo

Son el equivalente en el ecosistema ROS a los procesos en los sistemas operativos modernos. Suelen ejecutarse directamente mediante el comando **rosrun** o bien desplegándose con roslaunch.

```
rosrun rqt_graph rqt_graph
roslaunch beginner_tutorials launch_file.launch a:=1 b:=5
```

Cada nodo necesita identificarse con nodo, dirección URI, puerto y tipos de mensajes para la comunicación.

Nodo maestro

El nodo maestro es el servidor que media en todas las comunicaciones ROS. Además, registra los nombres de los nodos en ejecución, por lo que sin el previo despliegue del maestro, ningún nodo puede ejecutarse en el sistema.

Como antes se ha comentado, la comunicación se realiza en lenguaje xml, concretamente utilizando el protocolo XML-RPC. Como xml es compatible con muchos lenguajes de programación, la comunicación entre nodos se hace sencilla, pues este xml sirve de puente entre ambos lenguajes.

Asimismo, el nodo maestro se identifica como un nodo común. (nombre, URI, puerto)

Mensajes y Topics

Los mensajes en ROS son estructuras de datos que encapsulan tipos de datos más simples. En el ecosistema ros se utilizan lenguajes intermedios de descripción de lenguajes que se archivan en ficheros .msg que luego son traducidos a xml.

El principal mecanismo que se utiliza en ROS para establecer comunicación entre varios nodos son los topic (o temas). El topic es un sistema de comunicación asíncrona unidireccional que se establece entre nodos.

Para enviar mensajes a través de un topic, un nodo debe declararse el emisor (publisher) para un topic y declara el tipo de mensaje que enviará en ese topic. Primero el manejador del nodo se crea y luego se publicita el topic "versos"de tipo String, de std_msgs. [1] El tipo std_msgs es el tipo de mensaje que encapsula strings:

```
ros::NodeHandle manejador;
ros::Publisher emisor_versos = manejador
.advertise<std_msgs::String>("versos", 1000)
```

Para recibir mensajes, un nodo debe declarase subscriptor (subscriber) para un topic. Como la comunicación entre nodos es asíncrona, el subscriptor no necesita esperar a recibir el mensaje, mi_interrupcion es una función que entra en ejecución cuando un mensaje del topic "versos.[es] recibido. A diferencia del emisor, el subscriptor no declara el tipo del mensaje al subscribirse, ya se define en la función:

```
ros::NodeHandle manejador;
ros::Publisher emisor_versos = manejador
.subscribe("versos", 1000, mi_interrupcion)
```

Los roles suscriptor y emisor no son intercambiables, por lo que la comunicación es unidireccional.

Para comunicaciones de módulos complejos es necesario identificar cada mensaje y dotarlo de un sello de tiempo, por eso en muchos casos se incluye el mensaje Header, que se identifica con el frame de un robot, en otros más complejos. Esto permite realizar transformaciones entre

[1] Diferente de std::string, que no es un mensaje de ROS sino una clase básica de la librería estándar de C++.

diferentes frames o detectar fallos de sincronización.

Servicios

Las acciones y los servicios son utilizados en ROS para comunicaciones complejas, bidireccionales y que se alargan en el tiempo. Utilizan el modelo cliente-servidor y a diferencia del sistema emisor-subscriptor, que se declara y continúa durante toda la vida del nodo, los sistemas cliente-servidor duran lo que la comunicación entre estos. Una buena alternativa para no sobrecargar el sistema de comunicaciones.

Los servicios se utilizan en comunicaciones síncronas. El cliente inicia la comunicación con una petición (request), utilizando el formato .srv. El archivo .srv define el tipo de los mensajes para la petición y para la respuesta. El servidor recibe la petición y ejecuta una rutina asociada. Finalmente el servidor responde (response) la conexión con otro mensaje y se termina la conexión.

Acciones

Las acciones se utilizan en comunicaciones asíncronas, de forma que servidor y cliente puedan realizar otras funciones paralelamente a la ejecución. Son muy utilizadas para labores de navegación, tanto que ROS adopta su terminología para los comandos. Principalmente un servidor de acciones ofrece cinco diferentes comandos mediante los que comunicarse con el cliente. Estos comandos son: meta, estado, cancelar, feedback y resultado. Cada uno de estos son topics, por lo que los comandos se ejecutan en una sola dirección.

Para la comunicación entre cliente y servidor, al igual que en los puntos anteriores, es necesario especificar el formato del mensaje en un archivo .action . A continuación aparece un ejemplo de especificación de este archivo. Cada definición se separa con tres guiones –", la primera es la definición del mensaje goal, el segundo el mensaje feedback y el último resultado.

```
#goal definition
int32 mandato
---
#result definition
int32[] resultado_de_la_accion
---
#feedback
int32[] info_del_proceso
```

Para crear acciones dentro del sistema ROS, se emplea el meta-paquete **actionlib**, que incluye diferentes interfaces de las que heredar y que ofrecen comportamientos genéricos establecidos ya en relación a los comandos mencionados.

Parámetros

Para archivar datos concretos que no varían durante la ejecución es frecuente utilizar parámetros. Los parámetros se pueden especificar de forma directa o mediante ficheros .yaml. Son valores de tipos simples, como la velocidad máxima de determinado actuador, la incertidumbre de un sensor o el nombre de un topic. Estos parámetros se registran en el **servidor de parámetros** del nodo maestro.

Plugin

En sistemas robóticos muy complejos, se busca reducir el número de dependencias al mínimo y sustituir componentes fácilmente. Es por esto que ROS ofrece mecanismos para abstraer módulos y funciones software del resto mediante el uso de interfaces y clases abstactas o virtuales. Concretamente, un mecanismo dentro del propio ROS que logra esto de forma que sea ordenada y monitorizable es **pluginlib**. Se trata de una librería de plugins en C++ para la carga y descarga de plugins en nodos ROS. De esta manera, una librería o nodo ROS no necesita tener constancia de la existencia del nodo o librería que ejecuta una función, el sistema se encarga de vincular ambas registrando para diferentes interfaces las diferentes clases que se exportan como plugins.

tf

El paquete tf de ROS es muy importante para situar y medir puntos concretos en relación a un origen de coordenadas (*frame*) seleccionado. La palabra tf es una abreviatura de *transformation*, pues tf encapsula los mecanismos para *transformar* la métrica de la posición de un objeto con respecto a otro *frame*. Al estar sincronizadas con el reloj del sistema ROS, además es posible proyectar objetos de un frame a otro en el pasado.

Por otra parte, existen dos tipos de tf, estáticas y dinámicas. Las primeras establecen una referencia fija que no necesita actualizarse en el tiempo como por ejemplo la relación entre la posición de un sensor y la base robótica sobre la que se dispone. Las segundas establecen una referencia caduca, que es variable en el tiempo, por ejemplo la posición de un robot móvil con respecto al eje del mapa.

2.2 MRPT

El kit de utilidades de programación para robótica móvil (mobile robot programming toolkit, MRPT) es un conjunto de librerías y herramientas multi-plataforma de código abierto que tienen la finalidad de servir de marco de trabajo para el desarrollo de aplicaciones robóticas complejas. También incluye entre sus haberes multitud de aplicaciones de propósito variado y que son comúnmente empleadas en robótica. Así como funcionalidades genéricas de geometría, algebra lineal y análisis probabilístico.

En la figura 2.2 se encuentra un grafo con las librerías que contiene MRPT y sus dependencias. El grafo representa las librerías y dependencias que se incluían en la versión 1.5. Este proyecto se construye sobre la versión 1.9.9, pero también conlleva el mantenimiento y sobretodo la actualización de un paquete con dependencias a MRPT, así que se analizarán los cambios entre versiones y cómo afectó al código.

2.2.1 Módulos

Principalmente, los módulos utilizados para este proyecto ha sido mrpt-nav y mrpt-kinematics, pero para dejar patente la abundancia de librerías y la diversidad de funciones que ofrecen se listarán y definirán brevemente.

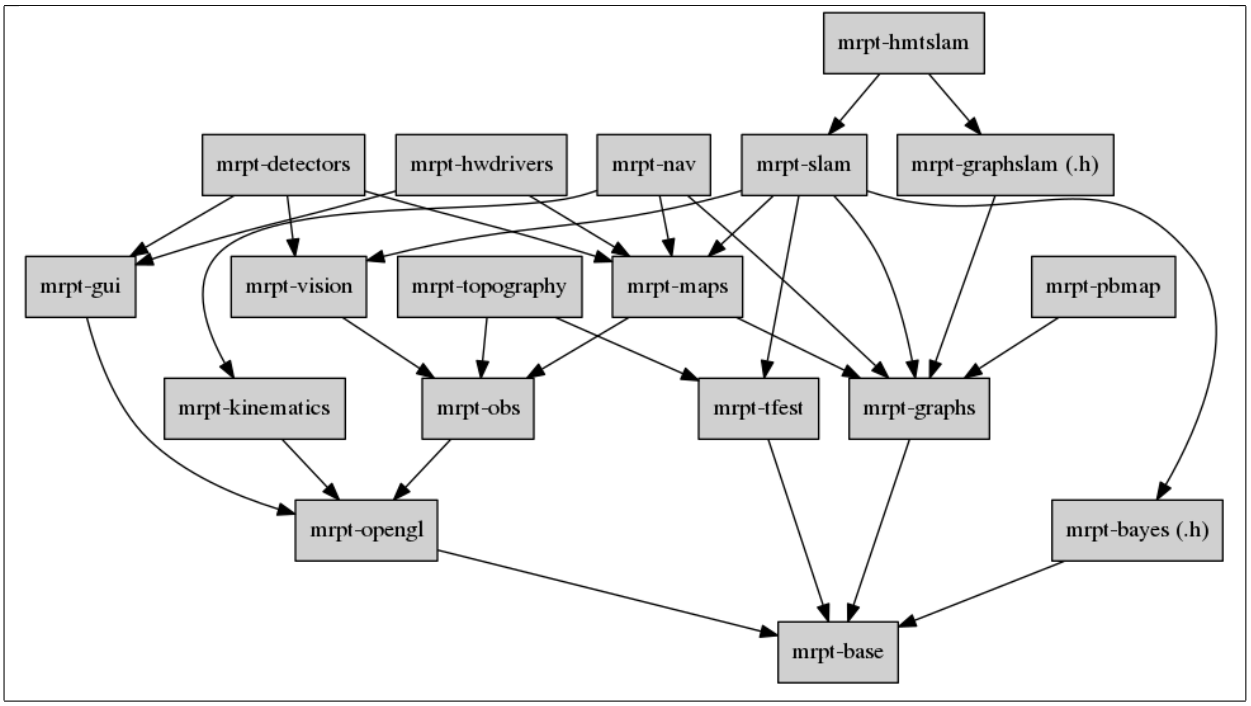


Figura 2.2: Módulos de MRPT 1.5 y sus dependencias.

mrpt-base

Se trataba del módulo principal de MRPT. En la versión actual está obsoleto, las funcionalidades se han dispersado para ganar en modularidad. Incluía los submódulos utils, math y system, entre otros. La desaparición de este módulo causó la eliminación de esas dependencias

mrpt-opengl

Incluye numerosas para la representación de objetos 3D. Basada en el motor de renderizado OpenGL.

mrpt-bayes

Agrupa algoritmos basados en el filtro de Kalman y en filtros de muestreo y partículas.

mrpt-kinematics

Son las clases utilizadas para las operaciones cinemáticas del robot. Contiene diferentes clases dependiendo del vehículo, haciendo hincapié en el modelo diferencial.

mrpt-obs

Relacionada con el uso e integración de lecturas de sensores, ya sea, escáner, cámara rgbd, datos de gps, odometría, sónar, etc.

mrpt-tfest

Se encarga de manejar las transformaciones entre frames. Similar a tf de ROS.

mrpt-graphs

Para estructuras arbóreas y de grafos. También se incluyen algoritmos de recorrido y recorrido óptimo de los mismos.

mrpt-gui

Provee clses para la representación de GUIs con dependencias de mrpt-opengl y wxWidgets.

mrpt-vision

Algoritmos relacionados con visión por computador, generalmente extensiones de operaciones de OpenCV.

mrpt-topography

Para la utilización de estructuras topográficas complejas. Basado en la librería Eigen.

mrpt-maps

Incluye clases de los mapas usados por los algoritmos de construcción de mapas y localización de las clases de MRPT.

mrpt-pbmaps

Mapas basados en planos. Incluye una clase especial de mapas que incluyen características especialmente útiles en la aplicación de localización y mapeado simuláneas (SLAM).

mrpt-detectors

Relacionada con mrpt-vision, se trata de clases para la detección de objetos a través de visión por computador.

mrpt-hwdrivers

Clases relacionadas con hardware robótico, comunicaciones vía puerto serie, usb etc... (Por ejemplo, SwissRanger, Kinect, Hokuyo...) [1]

[1] `https://www.mrpt.org/tutorials/supported_hardware_and_sensors/`

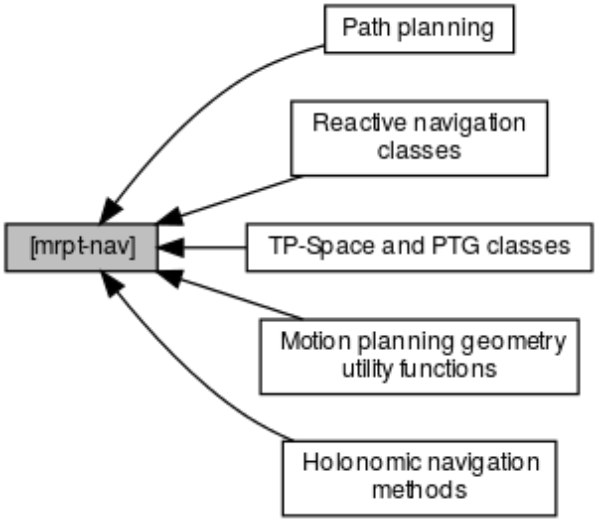


Figura 2.3: Grafo de navegación de los módulos de navegación de MRPT

mrpt-slam y acabados en slam

Para localización y mapeado simultáneos.

mrpt-nav

La libería más importante para este proyecto. Incluye submódulos que a su vez incluyen algoritmos para la navegación. Está pensada para utilizarse como motor de navegación híbrida, por eso se incluyen módulos tanto para navegación deliberativa como para reactiva (Fig. 2.3). Concretamente:

1. Plan de camino. Algoritmos para la generación de caminos en navegación deliberativa.
2. Navegación reactiva. Donde se encuentra el motor de navegación reactiva y sus interfaces para utilizarlo. Está adaptado para la navegación 2D y 3D (realmente 2.5D, ya que la dimensión de alturas se toma discreta y finita).
3. TP-Space y PTG. Algoritmos para la generación de trayectorias reactivas. En concreto, clases para cada tipo de trayectoria.
4. Utilidades de planificación geométrica. Para la planificación de movimiento incorporando la geometría del robot. Relacionado con el espacio de configuraciones del robot.
5. Navegación holonómica. Clases para movimiento holonómico, esto es, sin restricciones cinemáticas.

Capítulo 3

Estado del Arte: Navegación en ROS y algoritmos reactivos de generación de trayectorias

Dentro de ROS existe un meta-paquete muy popular llamado $navigation$. Este meta-paquete ofrece los paquetes y nodos necesarios para que un robot pueda integrar la información odométrica y sensorial que genera en pos de encontrar los comandos de movimiento necesarios para navegar hacia una meta tolerando cambios que puedan suceder en el ambiente.

Un robot puede comunicarse con $navigation$ gracias a los topics y mensajes de ROS. Por ejemplo: El robot publica la información de sus sensores en un topic, $navigation$ se subscribe a este topic y así, el robot es capaz de utilizar estos datos para lograr esquivar un obstáculo repentino surgido en su camino.

Como requisito principal para utilizar $navigation$, un robot ha de publicar los frames de de sus componentes hardware más significativos, generalmente, los emisores de mensajes. De no contar con estas referencias, los mensajes y, en general, cualquier información recogida por el robot, no tendría interpretación en el mundo real. La manera de integrar las referencias de los componentes del robot en ROS es mediante un árbol de tf.

Sin embargo, como este conjunto de paquetes surgió para realizar navegación en plataformas robóticas de dos ruedas y para habitaciones, funciona principalmente en vehículos diferenciales y holonómicos y en 2D. Como Siegwart y Nourbakhsh resumen en [20], dichos vehículos tienen dos y tres grados de libertad. En ROS, los comandos de velocidad son de tipo $geometry\_msgs/Twist.msg$, pero por restricciones cinemáticas, en el caso de $navigation$, solo son utilizados x, y, θ_z, por lo que deben ser traducidos si es necesario antes de ser enviados hacia los actuadores del robot.

Quede presente la importancia en la navegación que tiene reconocer el ambiente en el que se desplaza y conocer el efecto que tiene en el mundo su manifestación física robótica. Estos son los tan discutidos problemas de generación de mapas y localización del robot. Tales problemas se han asumido resueltos [21, 22], pues no es el objeto del presente proyecto como así lo referente a desplazamiento.

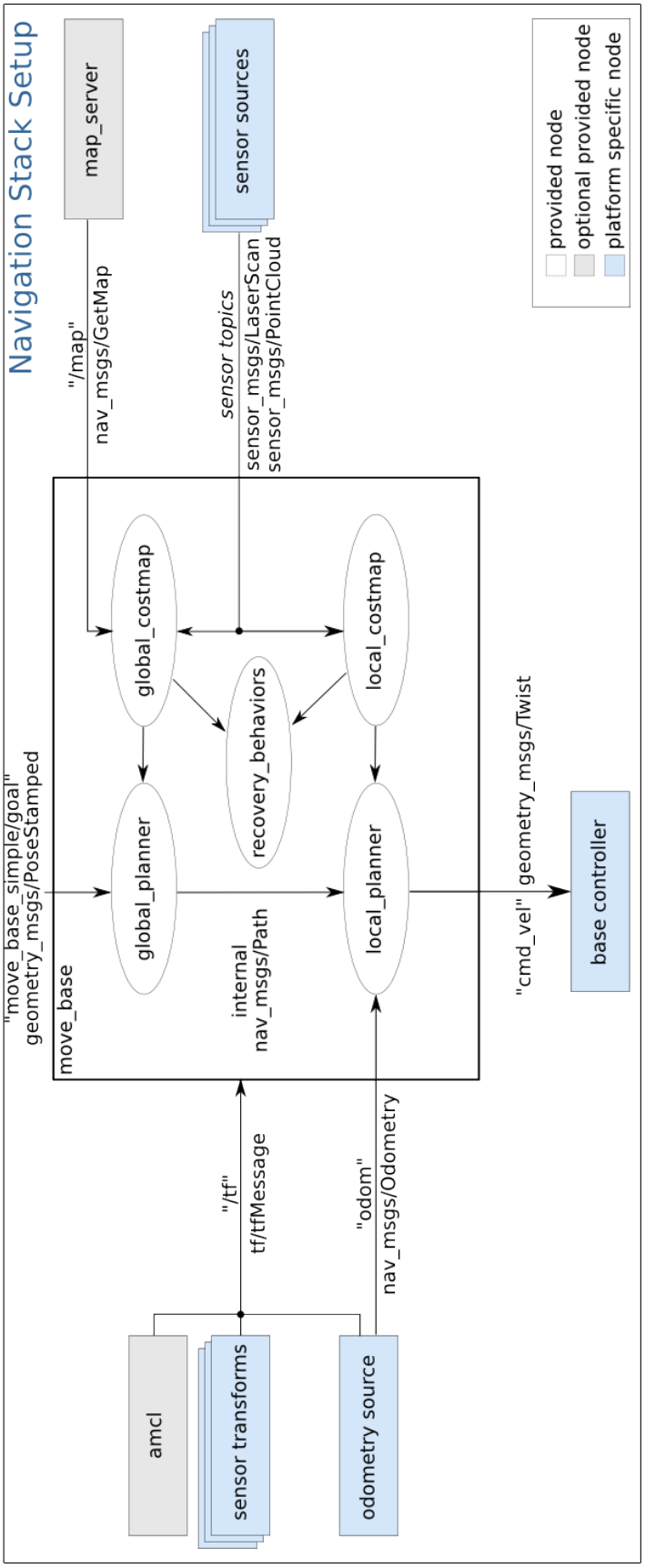


Figura 3.1: Conexiones entre move_base y el meta-paquete navigation.

3.1 move_base

El núcleo de $navigation$[1] se encuentra en move_base. El paquete $move\_base$ representa un servidor de acciones, con el que el robot (cliente) ha de conectarse para recibir comandos de velocidad.

En la Figura 3.1, queda patente la estructura del conjunto de paquetes $navigation$ y cómo $move\_base$ es el núcleo. En la zona izquierda se encuentra la información de la estructura del robot en forma de mensajes tf y su odometría. En la zona derecha la información sensorial y de mapeado. Arriba las metas enviadas al robot, abajo los comandos de movimiento que se generan como salida.

Dentro de el recuadro que engloba a $move\_base$, se encuentran los componentes que hacen funcionar al paquete. En la parte izquierda del recuadro, se encuentran $global\_planner$ y $local\_planner$, los encargados de planificar el movimiento del robot. Son interfaces provenientes del paquete nav_core que permiten a $move\_base$ importar plugins externos de forma que estos sean los encargados de implementar dichas interfaces. En la parte derecha, $global\_costmap$ y $local\_costmap$, con la función de enviar a los anteriores la información necesaria para completar su tarea. En el centro se encuentra $recovery\_behaviours$, que es el conjunto de interfaces utilizadas para responder a situaciones improvistas utilizando la información proveniente de los mapas de coste.

3.1.1 Mapas de coste

La estrategia elegida para resumir y presentar la información sensorial y el mapa construido a $move\_base$, son los mapas de coste (costmaps). Para explicar qué es un mapa de coste, es útil explicar antes qué es un mapa de celdas de ocupación, pues el concepto está intrínsecamente relacionado.

Un mapa de celdas de ocupación (grid occupancy map [23]), es por un lado un mapa de celdas, esto es, una representación aproximada del espacio continuo del mundo real en un espacio discreto que son las celdas del mapa. Por otra parte, lo que se quiere representar en este tipo de mapa es si una celda está ocupada por un objeto o no, puesto que las celdas no ocupadas en principio serían libres de ser recorridas por el robot. Dada la incertidumbre anexa siempre a las medidas tomadas en el mundo real, es imposible adjudicar con certeza a una celda dos únicos valores, en su lugar, en ellas se dispone una probabilidad de ocupación. Esta probabilidad se realimenta con la información sensorial aplicando regla de Bayes. En un mapa de costes se muestran, en lugar de probabilidades, costes o riesgos, enriqueciendo así su semántica intrínseca. Esto es especialmente interesante para robots de servicio, ya que se pueden incorporar al mapa restricciones sociales o meramente circunstanciales del entorno que se modela. En [24], se muestra un uso de este tipo de mapas de forma eficiente.

David V. Lu et al. publicaron en [25] la descripción y explicación del funcionamiento de la implementación de los mapas de coste más popular que existe actualmente en ROS, pues son sus principales contribuyentes. Con su aportación, se incorpora al sistema de mapas de cote un sistema de capas que permite organizar en cada una de ellas información útil para la navegación de forma que es más accesible y dinámica para $move\_base$. Un caso muy básico pero bastante habitual, consiste en tener la información del mapa en una capa y los costes en otra, así, cambios en los costes no requerirían modificar el mapa y este podría seguir considerándose.

[1] `https://github.com/ros-planning/navigation`

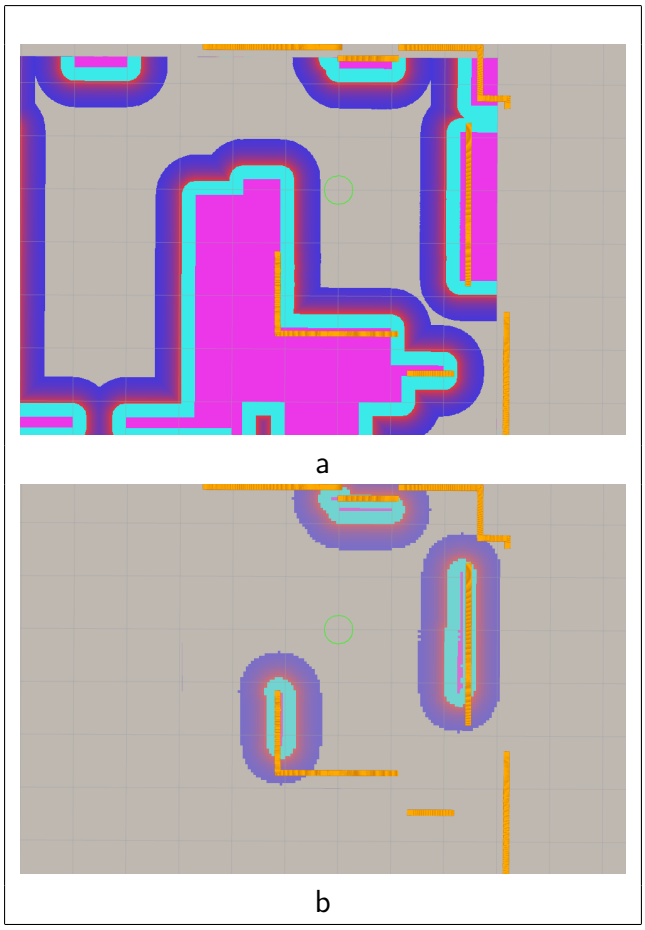


Figura 3.2: Representación de los mapas de coste por capas de $move\_base$ sobre la lectura de un sensor láser en simulación. a:global, b:local

El nodo $move\_base$ utiliza dos mapas de costes, uno para la fase deliberativa (global) y otro para la fase reactiva (local). El mapa de costes global, utiliza la información de la reconstrucción del escenario publicada por el meta-paquete $navigation$ y las tf del robot para calcular los costes de forma que se puedan considerar por el planificador global. El mapa de costes local, por su parte, incorpora información sensorial en tiempo real, teniendo en cuenta que, al no utilizarse todo el escenario, los datos son manejables a altas frecuencias de cómputo.

En la figura 3.2, se muestra cada uno de estos mapas, representando sobre ellos con colores el sistema de capas. La zona magenta es la reconstrucción del escenario donde se mueve el robot. La zona cian agrega la forma del robot a los bordes del escenario. Es la llamada $inflationlayer$, parámetros. Dicha forma, además, está representada como una circunferencia verde aproximada con un polígono de muchas aristas situada en la posición del robot. El degradado rojo a azul representa el incremento del riesgo del robot al acercarse a esa zona. La zona amarilla es la información sensorial, en este caso, producida por una simulación de láser.

3.1.2 Planificador global

La misión del planificador global es encontrar una ruta a la vez óptima y segura para la navegación del robot o vehículo. Recordando la figura 3.1, el planificador global utiliza un mapa de costes. Como se mencionaba con anterioridad, este mapa de costes incorpora la información de la reconstrucción del escenario en su capa estática y genera una capa de

relleno en función de parámetros de configuración.

El servidor de acciones $move\_base$, como se indicaba con anterioridad, importa plugins para los planificadores de movimiento. Así, se evita depender de una implementación concreta y se deja al usuario la elección del algoritmo adecuado a sus necesidades. Exactamente, $move\_base$ utiliza una interfaz llamada $BaseGlobalPlanner$. Las implementaciones más utilizadas, incluidas además dentro del meta-paquete $navigation$, están en los paquetes $global\_planner$ y $navfn$. Siendo la primera una versión actualizada y más estable de la segunda.

El paquete $navfn$ basa su funcionamiento en el algoritmo de camino mínimo de Dijkstra [26] utilizando el mapa de costes como fuente de conocimiento. En el paquete $global\_planner$, se ofrece, además, la posibilidad de utilizar el algoritmo A*, [27] sin embargo, aunque en la teoría, Dijkstra es un caso concreto de A* en el que no se cuenta con información heurística, [28] particularidades de la implementación en $global\_planner$, hacen que $navfn$ pueda mejorar la eficacia de $global\_planner$.[1]

3.1.3 Planificador local

Durante cada intervalo de tiempo dado, el planificador global busca la ruta óptima más segura desde la posición actual del robot a la meta. Esta ruta se transmite al planificador local dentro de $move\_base$. La misión del planificador local es encontrar los comandos de velocidad necesarios para seguir la ruta, esquivando obstáculos y sobreponiéndose a imprevistos. En general, reaccionar al entorno siguiendo el camino.

Al igual que con el planificador global, existe una interfaz en el paquete nav_core que mediante plugins es empleada por $move\_base$ para realizar la planificación local. Dicha interfaz es $BaseLocalPlanner$.

Las implementaciones más populares, objeto de estudio durante el presente escrito están en los paquetes $dwa\_planner$ y $base\_local\_planner$. DWAPlanner. *Dinamic Window Approach*. El enfoque de la ventana dinámica. TrajectoryPlanner. *Trajectory Roll-out*. Muestreo de trayectorias.

3.2 Dynamic Window Approach

3.2.1 Movimiento

El enfoque de la ventana dinámica *(Dynamic Window Approach, DWA)* es un algoritmo de planificación de movimiento reactivo diseñado por D. Fox et al. [29, 30] de cierta longevidad y que se encuentra bastante extendido dentro de la comunidad robótica, en parte por su implementación dentro de ROS, en parte por su sencillez y capacidad de integrarse con otros algoritmos [31, 32]. Este algoritmo consiste en el muestreo discreto de diferentes velocidades sujetas a restricciones cinemáticas y dinámicas así como en la búsqueda de la muestra que maximice unos criterios dados y que permitan alcanzar un punto 2D.

El enfoque de la ventana dinámica se plantea como una solución con la máxima de la eficiencia. Toma como modelo de movimiento las ecuaciones del vehículo de ruedas sincronizadas (Fig. 3.3). Utilizar este modelo permite reducir la búsqueda de velocidades a un espacio real

[1] Lu comenta esto en: `https://answers.ros.org/question/98511/global_planner-package-with-a-planner-question/?answer=98516#post-id-98516`

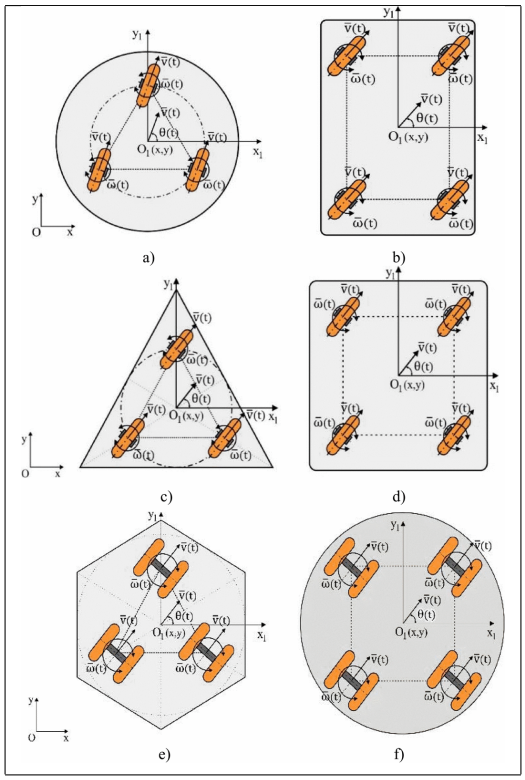


Figura 3.3: Esquema de robots no holonómicos de ruedas sincronizadas. [1]

de dos dimensiones cuyas componentes son velocidad lineal v y velocidad angular w. Sin embargo, la dirección de v depende de la pose del robot: $P = [x\ y\ \theta]^T$, concretamente depende su orientación θ. Esto provoca que para trasladarse, el robot siga la siguiente ecuación:

$$P(t_n) = \begin{bmatrix} x(t_0) \\ y(t_0) \\ \theta(t_0) \end{bmatrix} + \begin{bmatrix} \int_{t_0}^{t_n} v(t)\cos\theta(t)dt \\ \int_{t_0}^{t_n} v(t)\sin\theta(t)dt \\ \int_{t_0}^{t_n} w(t)dt \end{bmatrix} \tag{3.1}$$

Sustituyendo θ la traslación queda:

$$P(t_n) = \begin{bmatrix} x(t_0) \\ y(t_0) \end{bmatrix} + \begin{bmatrix} \int_{t_0}^{t_n} v(t)\cos\left(\theta(t_0) + \int_{t_0}^{t} w(k)dk\right)dt \\ \int_{t_0}^{t_n} v(t)\sin\left(\theta(t_0) + \int_{t_0}^{t} w(k)dk\right)dt \end{bmatrix} \tag{3.2}$$

Obsérvese que el algoritmo DWA acepta solo metas como puntos, por lo que se deja fuera del cálculo la orientación. (Eq. 3.3). En aras de simplificar aún más, se toma la integral como suma discreta de intervalos continuos:

$$P(t_n) = \begin{bmatrix} x(t_0) \\ y(t_0) \end{bmatrix} + \begin{bmatrix} \sum_{i=0}^{n-1} \int_{t_i}^{t_{i+1}} v(t)\cos\left(\theta(t_i) + \int_{t_i}^{t} w(k)dk\right)dt \\ \sum_{i=0}^{n-1} \int_{t_i}^{t_{i+1}} v(t)\sin\left(\theta(t_i) + \int_{t_i}^{t} w(k)dk\right)dt \end{bmatrix} \tag{3.3}$$

D. Fox, W. Burgard y S. Thrun muestran en [30] que cualquier trayectoria de estas características puede aproximarse con trayectorias circulares si los intervalos de aproximación son lo suficientemente pequeños. (Las trayectorias circulares son especialmente útiles para un algoritmo reactivo de navegación ya que su cálculo es inmediato utilizando los comandos de

velocidad de $move\_base$. También es sencillo encontrar puntos de corte). Asumir esto, conlleva que la velocidad sea constate, (Eq. 3.3) puede simplificarse en:

$$P(t_n) = \begin{bmatrix} x(t_0) \\ y(t_0) \end{bmatrix} + \begin{bmatrix} \sum_{i=0}^{n-1} \int_{t_i}^{t_{i+1}} v_{t_i} \cos\left(\theta(t_i) + \int_{t_i}^{t} w_{t_i} dk\right) dt \\ \sum_{i=0}^{n-1} \int_{t_i}^{t_{i+1}} v_{t_i} \sin\left(\theta(t_i) + \int_{t_i}^{t} w_{t_i} dk\right) dt \end{bmatrix} \tag{3.4}$$

Si resolvemos las integrales:

$$P(t_n) = \begin{bmatrix} x(t_0) \\ y(t_0) \end{bmatrix} + \begin{bmatrix} \sum_{i=0}^{n-1} \int_{t_i}^{t_{i+1}} v_{t_i} \cos\left(\theta(t_i) + w_{t_i}[t - t_i]\right) dt \\ \sum_{i=0}^{n-1} \int_{t_i}^{t_{i+1}} v_{t_i} \sin\left(\theta(t_i) + w_{t_i}[t - t_i]\right) dt \end{bmatrix} \tag{3.5}$$

Para $w_i \neq 0$:

$$P(t_n) = \begin{bmatrix} x(t_0) \\ y(t_0) \end{bmatrix} + \begin{bmatrix} \sum_{i=0}^{n-1} \frac{v_i}{w_i} [\sin\left(w_i(t_{i+1} - t_i) + \theta(t_i)\right) - \sin(\theta(t_i))] \\ \sum_{i=0}^{n-1} -\frac{v_i}{w_i} [\cos\left(w_i(t_{i+1} - t_i) + \theta(t_i)\right) - \cos(\theta(t_i))] \end{bmatrix} \tag{3.6}$$

Para $w_i = 0$:

$$P(t_n) = \begin{bmatrix} x(t_0) \\ y(t_0) \end{bmatrix} + \begin{bmatrix} \sum_{i=0}^{n-1} v_i \cos\theta(t_i)[t_{i+1} - t_i] \\ \sum_{i=0}^{n-1} v_i \sin\theta(t_i)[t_{i+1} - t_i] \end{bmatrix} \tag{3.7}$$

De forma simple:

$$P(t_n) = P(t_0) + \Delta P(t_n) \tag{3.8}$$

3.2.2 Espacio de búsqueda

Las expresiones en (Eq. 3.7, 3.8) resultan ahora útiles para llevar a cabo la búsqueda de la trayectoria óptima. Después de desarrollar la expresión que se relaciona con la trayectoria generada, se reduce la búsqueda de velocidades a solo el primer $[i, i+1]$ y se asume constante para el resto. Por ello (v_i, w_i) pasa a ser (v, w).

Espacio de trayectorias circulares

Como antes se introdujo, solo se consideran trayectorias circulares, ya que cualquier trayectoria genérica se puede aproximar como una sucesión de trayectorias circulares cuando el intervalo de cada una de ellas es suficientemente pequeño. Eso provoca que cada trayectoria se pueda caracterizar como el vector (v, w), donde v es la velocidad lineal constante durante la trayectoria y w la velocidad angular bajo las mismas condiciones. Así, la búsqueda de la trayectoria óptima resulta análoga a la búsqueda del vector (v, w) óptimo.

Por tanto, así como queda descrito en la figura 3.4, la búsqueda de velocidades comienza en ese espacio V_s en el algoritmo DWA.

Espacio de velocidades admisibles

Como segundo paso del algoritmo, se plantea restringir este espacio de búsqueda fijando las velocidades a aquellas que no provoquen colisiones (V_a). Una de las razones por las que D.

1. Se establece el espacio de búsqueda:

$$V = V_s \cap V_a \cap V_d \tag{3.9}$$

   (a) Restricción a trayectorias circulares.

$$V_s = \{(v, w) \mid |v|, |w| \leq v_{max}, w_{max}\} \tag{3.10}$$

   (b) Se eliminan las velocidades que impliquen acercarse demasiado a un obstáculo.

$$V_a = \{(v, w) \mid v \leq \sqrt{2 \cdot dist(v, w) \cdot \dot{v}}\} \tag{3.11}$$

   (c) Se aplica la ventana dinámica. Restricción a velocidades alcanzables teniendo en cuenta los límites de aceleración de los motores.

$$V_d = \{(v, w) \mid v \in [v_a - \dot{v} \cdot t, v_a + \dot{v} \cdot t], w \in [w_a - \dot{w} \cdot t, w_a + \dot{w} \cdot t]\} \tag{3.12}$$

   donde v_a representa la velocidad actual y $\dot{v}, \dot{w}$ aceleraciones lineal y angular máximas.

2. Se optimiza la trayectoria generada:

$$G(v, w) = \sigma(\alpha \cdot heading(v, w) + \beta \cdot dist(v, w) + \gamma \cdot vel(v, w)) \forall v, w \in V \tag{3.13}$$

   donde $heading$ es la diferencia de la orientación entre la meta y la posición resultado de aplicar (v, w), $dist$ es la distancia al obstáculo más próximo y vel la distancia euclídea (sin tener en cuenta orientación) desde la posición resultado de aplicar (v, w) a la meta. Los valores de estas funciones tienen que estar normalizados.

Figura 3.4: Resumen del algoritmo de la ventana dinámica.

Fox, W. Burgard y S. Thrun, cuando diseñaron DWA, [30] decidieron restringir las trayectorias posibles a solo circulares es la facilidad de encontrar puntos de corte de funciones circulares con puntos concretos. Pues, Eq. (3.8) permite establecer la ecuación de una trayectoria circular con la siguiente expresión:

$$(\Delta X(t_n) - M_x^n)^2 + (\Delta Y(t_n) - M_y^n)^2 = (\frac{v}{w})^2 \tag{3.14}$$

donde $M_x^n = -\frac{v_i}{w_i}\sin(\theta(t_i))$ y $M_y^n = \frac{v_i}{w_i}\cos(\theta(t_i))$. [1]

La función $dist$, que representa la distancia mínima a una colisión y aparece en el segundo apartado de la figura 3.4, emplea la ecuación (3.14) en la búsqueda de dicha colisión. Aquí ha de intervenir información sensorial, pues es la manera de obtener la posición de los obstáculos. Ha de contemplarse los errores derivados. En [33], en sus inicios, se plantearon soluciones a este problema que dieron resultados positivos.

Se establece la condición de Eq. (3.11) para formar V_a, que no es más que la relación entre velocidad, espacio y aceleración sin depender del tiempo. La constante $\dot{v}$ es la aceleración máxima que pueden ejercer los motores.

Restricción de la ventana dinámica

Para incorporar la dinámica del robot al espacio de búsqueda, se crea la ventana dinámica, que da nombre al algoritmo. Se trata de intersectar el conjunto $V_s \cap V_a$ con V_d, que se define en Eq. (3.12).

Cabe destacar que la región que define V_d, es rectangular en el espacio V_s y por eso recibe el nombre de ventana. Por otro lado, nótese que la única relación que existe entre V_a y V_d es la de no ser disjuntos[2]. Dependiendo de si existen o no, obstáculos dentro de la ventana, pueden existir elementos de V_d que no se encuentren en V_a.

3.2.3 Optimización de trayectorias

Tras obtener el espacio de búsqueda, solo falta encontrar un criterio cuyo valor óptimo represente la mejor trayectoria para alcanzar la meta. Este criterio se establece en Eq. (3.13).

Cada una de las funciones que forman parte de la definición, son necesarias para informarse suficientemente. Sin información sobre la orientación, no se puntuará teniendo en cuenta la restricción holonómica del robot. Sin la distancia hacia el obstáculo más cercano, no se beneficiarían las trayectorias seguras en cuanto a colisiones. Por último, sin la distancia hacia la meta, faltaría motivación para realizar la navegación.

Cada uno de los sumandos se ponderan con un peso, representados por (α, β, γ), elegidos de forma experimental. Además, a G se le aplica una función σ, con la misión de conseguir resultados más suaves, disminuyendo así el impacto de errores.

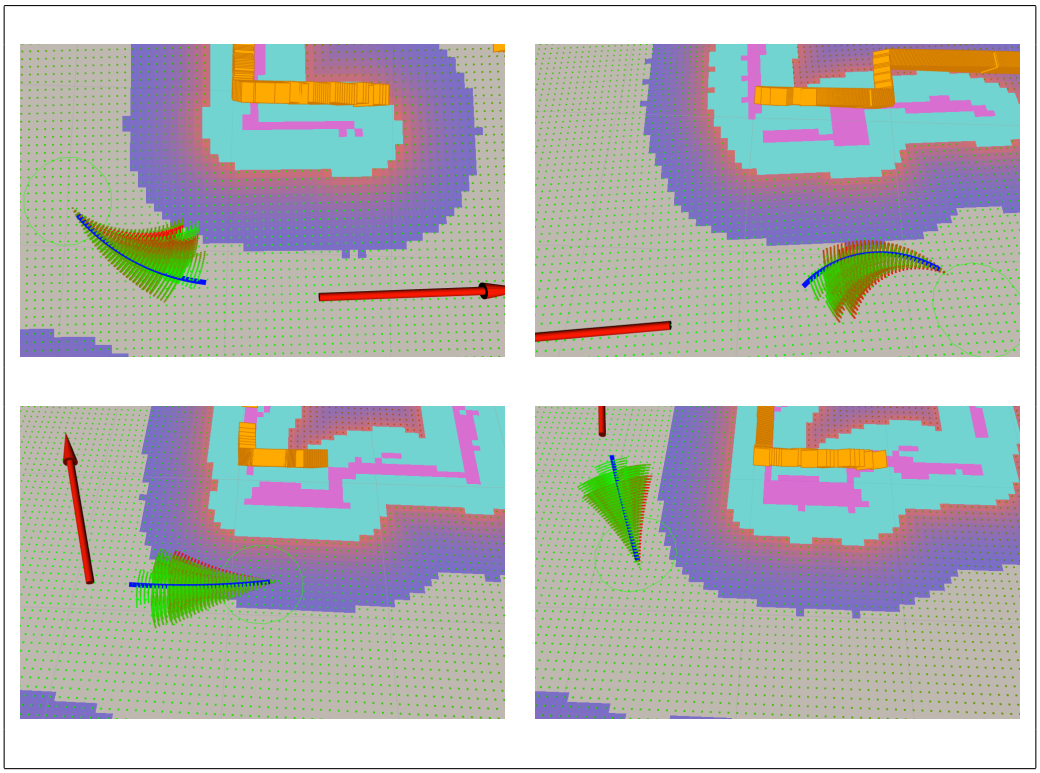


Figura 3.5: Representación del muestreo de trayectorias realizado por DWAPlanner en rviz.

3.2.4 Implementación en ROS

Dentro del repositorio de paquetes de ROS se encuentra $dwa\_local\_planner$, donde existe una implementación del algoritmo DWA. Como se introdujo en el capítulo 2[1], mediante el sistema de plugins de ROS, el nodo $DWAPlannerROS$ sobrescribe $BaseLocalPlanner$. De esta forma, el planificador logra integrarse de forma transparente con el controlador $move\_base$. La misión de este plugin es calcular para cada instante las velocidades necesarias para seguir la trayectoria que optimice el criterio seleccionado.

Sin embargo, para introducirse en armonía dentro de $move\_base$, que conlleva estar en el meta-paquete $navigation$, así como ROS en general, el algoritmo necesita someterse a ciertas modificaciones.

Las más importantes son:

- El algoritmo establece el espacio de búsqueda V_s en tres dimensiones (v_x, v_y, v_θ), permitiendo aprovechar la capacidad holonómica de algunos vehículos. Además, el espacio de búsqueda está discretizado. Se personaliza con los parámetros $samples\_vx$, $samples\_vy$ y $samples\_vtheta$. También se puede fijar el valor mínimo y máximo de los mismos.

- Trabaja seleccionando sucesivas metas locales procedentes del planificador global. Además, como información sensorial utiliza la información que aporta el mapa de coste local.

- El criterio de elección de la mejor trayectoria está adaptado al caso anterior. Ahora en la expresión intervienen $path\_distance$, $goal\_distance$ y $occdist$ y en menor medi-

[1]Nótese que por el teorema de Pitágoras: $(\cos A)^2 + (\sin A)^2 = 1$
[2]Se comprueba por el caso trivial $(v_a, w_a) \in V_a \cap V_d$
[1]TODO, cambiar a referencias del capítulo

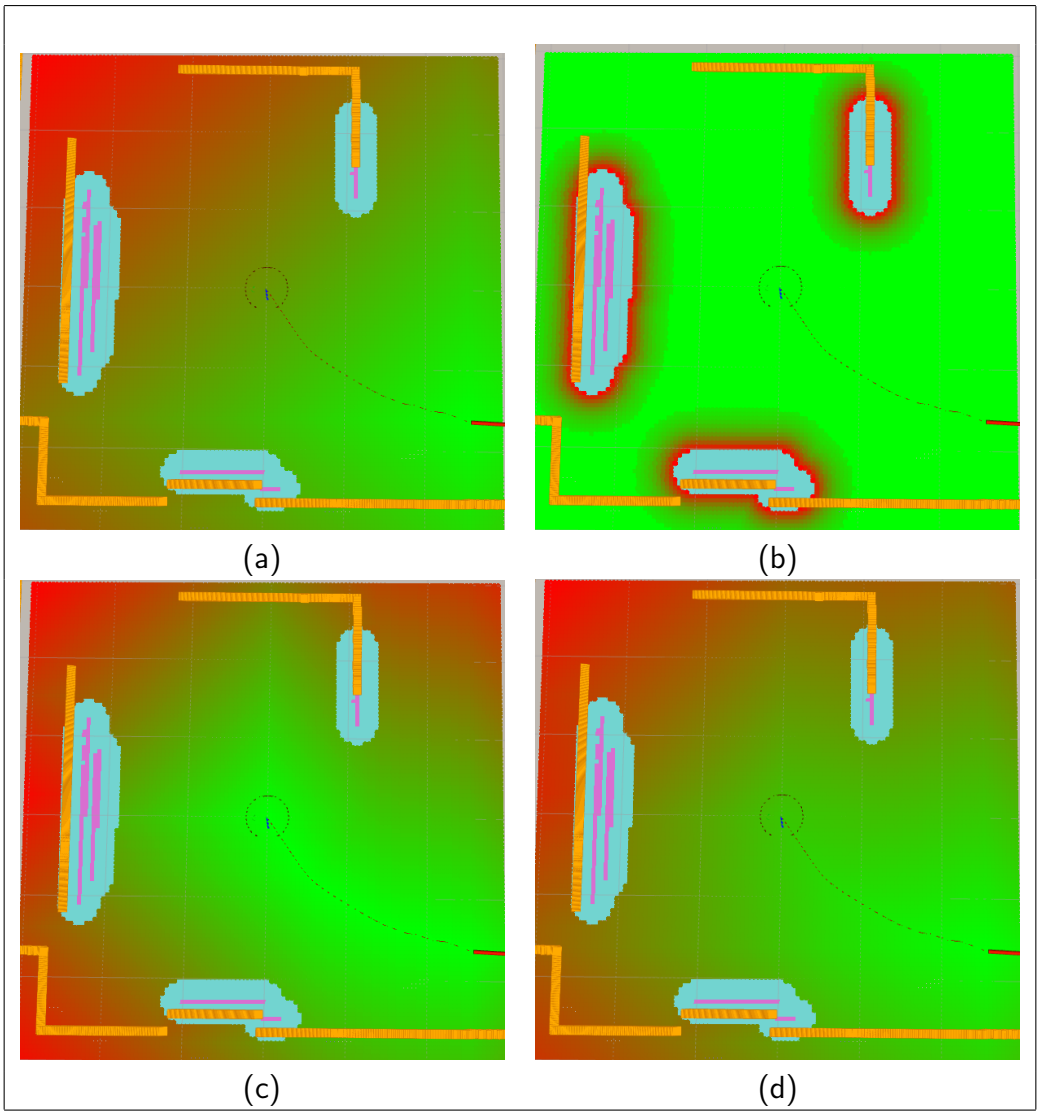


Figura 3.6: Estimación del coste DWA sobre costmap local. (a) Coste a la meta. (b) Coste de obstáculos. (c) Coste de camino. (d) Coste total.

da $twirling$. Es posible regular el impacto de cada uno en la elección de trayectorias modificando los parámetros $*\_scale$.

Existe la posibilidad de visualizar los parámetros asociados a cada trayectoria. Esto se muestra en la figura 3.5. En ella se muestra en una nube de puntos cada trayectoria muestreada. Dicha nube queda coloreada con un degradado de verde a rojo, según su función de coste total, donde el valor menor se representa por verde y el valor mayor por rojo. La mejor trayectoria aparece coloreada de azul. Para lograr visualizar estos resultados es necesario asignar a $true$, el valor del parámetro de ROS $publish\_traj\_pc$. Es posible seleccionar cada uno de los factores que intervienen en la función de coste para su visualización.

También mediante el parámetro $publish\_cost\_grid\_pc$, es posible obtener estos resultados. (Fig. 3.6). Se trata de la calificación de costes según la posición en el costmap local con los siguientes valores para generar cada mapa:

(a) $goal\_distance\_bias$: 24.0

(b) $path\_distance\_bias$: 32.0

(c) $occdist\_scale$: 0.01. La escala de este valor es mucho menor ya que por defecto se descartan los valores ocupados del costmap.

Se puede observar cómo el coste de acercarse a la meta disminuye para (a), se concentra alrededor de los obstáculos para (b), disminuye para valores cercanos al camino para (c) e incorpora trazas de todos ellos (d).

3.3 Trajectory Rollout

El algoritmo de Muestreo de Trayectorias (Trajectory Rollout, TR) surge para solucionar el problema de la navegación en exteriores que funcionaba dentro de una arquitectura de navegación híbrida, por lo que las máximas a seguir son un tanto diferentes a DWA [34]. Sin embargo, presentan muchos puntos en común.

3.3.1 Movimiento

Como se indicaba en el punto 3.2.1, en DWA se plantea el movimiento definiendo cómo predecir trayectorias. De ahí se restringen las posibles trayectorias a trayectorias circulares y finalmente se permite caracterizar dichas trayectorias con un par (v, w) de velocidades.

En TR se sigue el camino opuesto, se parte con la idea de movimiento generado a bajo nivel como comandos de velocidad (v, w) considerando una restricción no holonómica con la orientación. Las ecuaciones de movimiento son las siguientes:

$w_i \neq 0$

$$P(t_{i+1}) = \begin{bmatrix} x(t_i) \\ y(t_i) \end{bmatrix} + \begin{bmatrix} \frac{v_i}{w_i}[\sin(w_i \Delta t_i)] \\ \frac{v_i}{w_i}[1 - \cos(w_i \Delta t_i)] \end{bmatrix} \tag{3.15}$$

$w_i = 0$

$$P(t_{i+1}) = \begin{bmatrix} x(t_i) \\ y(t_i) \end{bmatrix} + \begin{bmatrix} v_i \Delta t_i \\ 0 \end{bmatrix} \tag{3.16}$$

A partir de estas ecuaciones se obtienen las trayectorias, pues estas son la suma de los movimientos del robot.

3.3.2 Espacio de búsqueda

Si bien ambos algoritmos realizan la búsqueda de un valor óptimo en un espacio de búsqueda, desde su concepción, dejando a un lado posteriores implementaciones, en TR se define dicho espacio de búsqueda como un conjunto de muestras discreto, finito y de tamaño fijo de dos dimensiones. Las muestras están formadas por los comandos de velocidad necesarios para realizar cada movimiento.

De esta forma, sustituye las restricciones sobre el espacio que realiza DWA delegando en otros módulos como la función de optimización de trayectorias, el hinflado de obstáculos [35] o el planificador global.

3.3.3 Optimización de trayectorias

Para obtener las trayectorias que posteriormente serán evaluadas, se necesita fijar dos parámetros: la duración de la trayectoria (n) y el tiempo de duración de cada movimiento (ϵ). De este modo, se generan n/ϵ intervalos en los que ejercer comandos de movimiento. Sea M, el tamaño del conjunto de muestras, el número de posibles comandos de movimiento sería $M^{\frac{n}{\epsilon}}$ dentro de una sola trayectoria. Como el tamaño es exponencial para valores de ϵ distintos de n, es necesario mantener velocidad constante. Se establece prioridad a las trayectorias según su velocidad lineal asociada, de mayor a menor.

El criterio de elección de la mejor trayectoria es el siguiente:

$$C(t) = \alpha Obs + \beta Gdist + \gamma Pdist + \delta \frac{1}{\dot{x}^2} \tag{3.17}$$

Donde Obs es la suma de costes de toda la forma del robot por la trayectoria. $Gdist$ representa la distancia a la meta, $Pdist$ al camino y $\dot{x}$ la velocidad lineal de la trayectoria. Si algún punto de la trayectoria coincide con el de un obstáculo, dicha trayectoria queda rechazada

Sin embargo, no es físicamente posible para un vehículo acelerar indefinidamente, por lo que es necesario restringir la velocidad según los límites de aceleración del robot. Esto ocasiona que para robots con límites de aceleración reducida, las trayectorias no sean circulares y por tanto, la calificación de la trayectoria podría ser inexacta.

3.3.4 Implementación en ROS

Dentro del paquete $base\_local\_planner$, se encuentra una implementación de este algoritmo en la clase $TrajectoryPlannerROS$. Del mismo modo que con DWA, la implementación se pone a disposición de $move\_base$ bajo el sistema de plugins de ROS.

En su mayor parte, se respeta el algoritmo presentado, sin embargo existen particularidades y extensiones a tener en cuenta:

- Existe soporte para movimiento holonómico, aunque no tan desarrollado como en DWA. Para activarlo, el parámetro $holonomic_robot$ debe fijarse como $true$. Una vez hecho esto, el parámetro y_vels permite introducir las muestras de velocidad en este sentido de forma explícita.
- El criterio de optimización de trayectorias no incluye término δ.

3.4 PTG Navigator

Como extensión y mejora de los anteriores algoritmos de navegación reactiva basados en el muestreo de trayectorias, J.-L. Blanco et al. [2] propuso un método de búsqueda que conllevaba modificar intensivamente el espacio de búsqueda. Concretamente destaca por transformar el espacio de búsqueda en uno donde movimientos lineales se corresponden con trayectorias específicas en el plano cartesiano. Además, se asegura de que cada restricción aplicada al espacio de búsqueda sea biyectiva, por tanto, invertible.

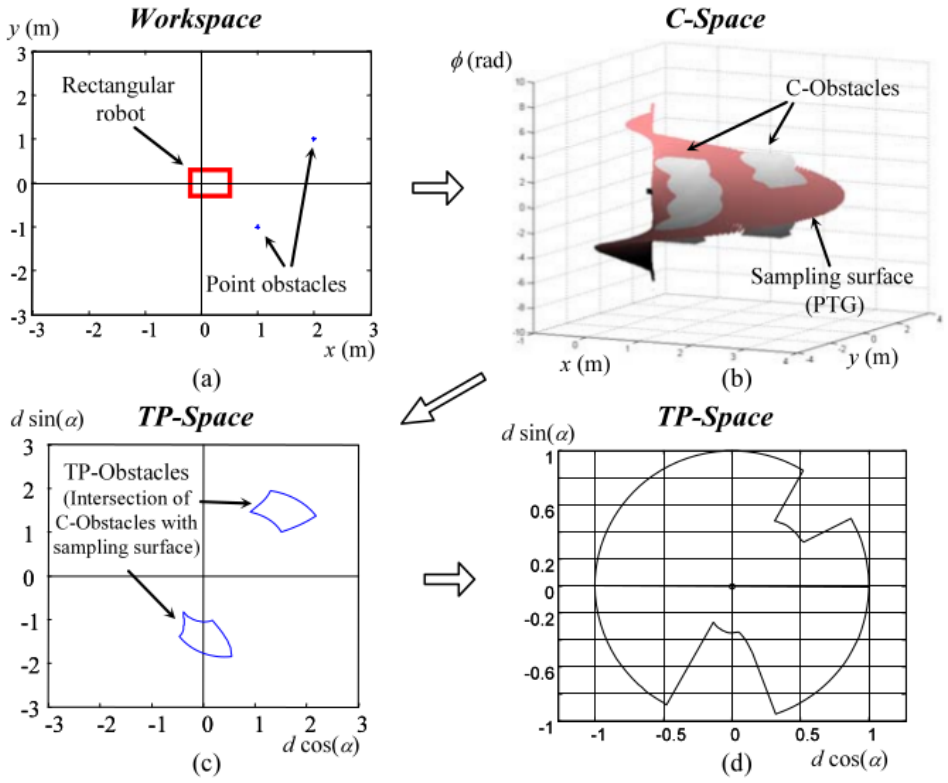


Figura 3.7: Restricciones en el espacio de búsqueda durante el algoritmo PTG. (a) Representación inicial del robot e info. sensorial sobre el mapa. (b) Superficie PTG sobre C-Space. Obstáculos representados con un degradado blanco a negro. (c) TP-Space (d) TP-Space normalizado. Fuente [2]

Lo que antes se denominó hinflado de obstáculos es la incorporación al entorno de las restricciones en la forma del robot para que algoritmos de movimiento libre (esto es, la suposición del robot como punto en el espacio) sean aplicables. Dependiendo de la forma del robot y de su orientación actual, el área de obstáculos puede variar en gran medida, es por esto que se suele aproximar la forma del robot como círculo que lo incluya. Esta suposición restringe en abundancia el movimiento, marginando zonas del mapa que deberían ser accesibles por el robot.

La solución a este problema pasa por representar todas las posibles orientaciones del robot para cada posición en cada obstáculo. Esto genera un espacio de tres dimensiones llamado espacio de configuraciones del robot. (C-Space, Fig. 3.7)

Sin embargo, como sucede en numerosos problemas de clasificación, aumentar la dimensionalidad del problema conlleva un incremento en el coste computacional. Para el caso de la navegación reactiva, que exige tiempos de respuesta muy cortos, hace inviable el proceso.

La solución propuesta, pasa por restringir el espacio de configuraciones a un modelo de trayectoria fijo. La representación de esta restricción en el espacio es una superficie. De esta forma, además, se consigue reducir el número de obstáculos y el volumen de los mismos, que conlleva más espacio libre por cada trayectoria. (Fig. 3.7 (b)). La superficie representada recibe el nombre de Generador de Trayectorias Paramétrico, PTG. Este proceso es similar a la definición de espacio de búsqueda en los dos algoritmos anteriores. Cada trayectoria se caracteriza con un vector de velocidades (v_0, w_0) y unas constantes configurables en la fase de

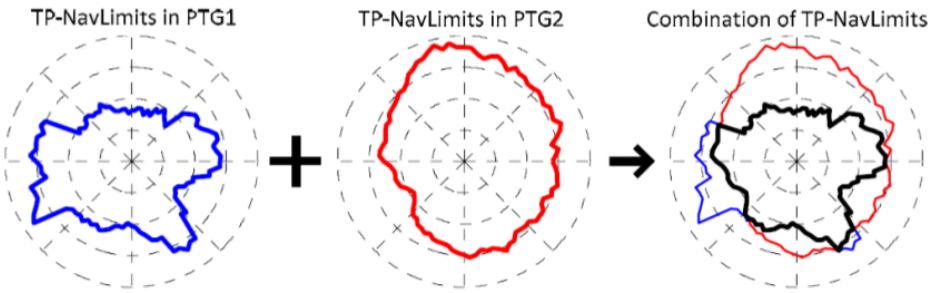


Figura 3.8: Calculo de la frontera de obstáculos del algoritmo TPSpace para un robot con dos bloques. Fuente [3]

diseño de PTG. Para obtener la superficie sobre C-Space, se integra numéricamente la función PTG en el intervalo $[0, t]$.

Al ser una superficie, Blanco et al. [2] representan este espacio utilizando dos dimensiones. Dan además una función distancia para cada superficie que da a la orientación más peso si dos puntos pertenecen a la superficie. De esta forma se genera un espacio vectorial no euclídeo de dos dimensiones. Por comodidad se representa en coordenadas polares. (TPSpace, Fig. 3.7 (c)) Los obstáculos son representados como la intersección de los mismos en el espacio de configuraciones con la superficie de trayectorias.

Finalmente, para obtener el espacio libre asociado a la trayectoria, el espacio se normaliza por el máximo de profundidad del sensor. Obsérvese que valores tras los obstáculos no son considerados. (TPSpace normalizado, Fig. 3.7 (d))

Sobre este espacio restringido se aplica un algoritmo de movimiento holonómico que traza caminos hasta el objetivo. Hágase hincapié en que el movimiento holonómico es equivalente a moverse en línea recta, pero que sobre la PTG equivale a cada muestra de trayectoria. Se proponen dos algoritmos, Campo de fuerzas vectorial (VFF) y Diagrama de cercanías. (ND). En ambos casos cada candidato de movimiento se califica con una heurística y se selecciona aquel candidato cuya heurística es máxima.

Para obtener los comandos de velocidad que serán enviados a los motores, se obtiene el par v, α resultado del método holonómico seleccionado, donde v es el módulo de velocidad y α la dirección de esta. Se evalúa la función de trayectoria PTG para α y se pondera escala dicho vector usando v dividido por el α asociado a la velocidad mayor sobre la superficie de PTG.

En [3], Jaimez et al. propusieron una extensión de este algoritmo con el fin de detectar colisiones del robot teniendo en cuenta su forma tridimensional. Básicamente la labor consiste en permitir la configuración de la forma del robot en segmentos tridimensionales y replicar el funcionamiento de las PTG en cada uno de ellos. De esta forma, se obtiene una capa TP-Obstacles diferente por cada bloque. Con cada uno de estos bloques se reducen eliminan los obstáculos coincidentes que más el movimiento del robot. En la figura 3.8, se ilustra un ejemplo para un robot equipado con una PTG y dos bloques diferentes, donde el primero de ello, la base robótica, se destaca con el color azul y el segundo, la parte superior, con el color rojo.

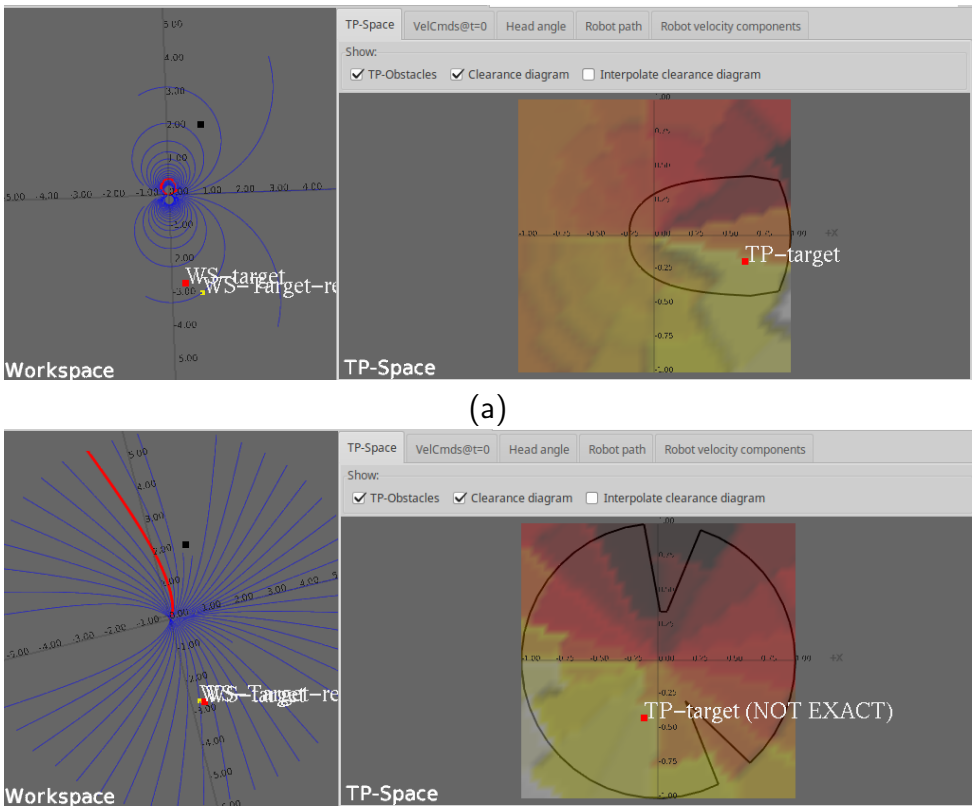


Figura 3.9: Interfaz gráfica del programa PTG-configurator. (a) PTG circular. (b) PTG α-asintótica.

3.4.1 Implementación en MRPT

Calcular la intersección de dos lugares geométricos en tres dimensiones puede ser una tarea computacionalmente costosa, especialmente inviable para un algoritmo de navegación reactiva. Razón por la que este proceso se realiza en MRPT mediante tablas de consulta con datos preprocesados. Es decir, se recorre el espacio de trabajo original y se traducen los obstáculos utilizando dicha tabla.

En la Figura 3.9, se pueden observar las dos principales PTG del sistema reactivo de MRPT. La trayectoria con el punto amarillo es la seleccionada.

Capítulo 4

Descripción e Integración de un planificador de movimiento reactivo dentro de la arquitectura ROS

Se cuenta con un nodo que realiza navegación 2D reactiva utilizando como fuente sensorial láseres. Como se puede observar en la figura 4.1, el planificador reactivo de mrpt, no logra alcanzar la meta y se detiene, incapaz de encontrar una trayectoria válida. La presencia de obstáculos, la necesidad de esquivarlos y es desconocimiento por parte del robot de la escena general en la que navega son las razones que provocan esta situación.

Para resolver el problema se plantea la integración del reactivo de mrpt dentro del controlador de movimiento $move\_base$ y del meta-paquete $navigation$. Además, como propuesta para hacer más atractivo al reactivo se pretende lograr la compatibilidad con el sistema 3D de MRPT.

4.1 Mantenimiento y actualización del reactivo 2D de MRPT para ROS

Durante el comienzo del desarrollo del plugin, se encontraron una serie de obstáculos a tener en cuenta y que se debieron solventar para llevar a cabo el resto del proyecto:

1. Los nodos del meta-paquete $mrpt\_navigation$, no se encontraban con total compatibilidad con la versión de mrpt de ese momento.
2. No existía un simulador funcional con el que lanzar MRPT en ROS. Dado que $mvsim$[1] no se encontraba totalmente integrado con la versión de mrpt de ese momento.

Para solucionar el primer punto, fue necesario modificar el código de forma que se respetase y utilizase la nueva estructura de MRPT. Dada la versión actual de MRPT no es totalmente compatible con el proyecto debido a problemas con cmake[2], se optó por intentar hacer la

[1] `https://github.com/ual-arm-ros-pkg/mvsim`
[2] `https://github.com/MRPT/mrpt/issues/892`

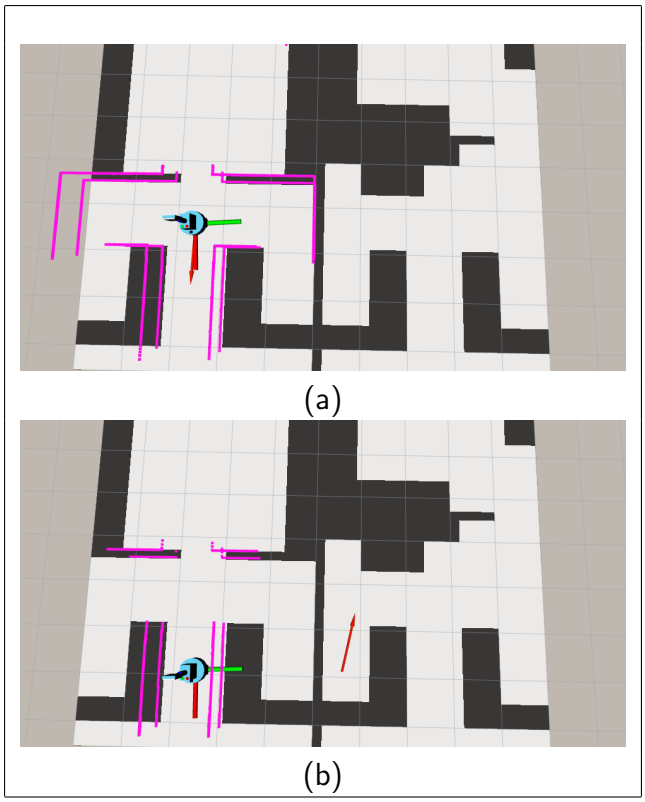


Figura 4.1: Inicio y final de navegación reactiva utilizando nodo ros reactivo de MRPT. (a) Inicio. (b) Final.

implementación flexible para las dos versiones. Esto se llevó a cabo gracias a las directivas condicionales de preprocesado compatibles con el compilador gcc. [1]

En cuanto al segundo punto, se barajaron tres posibles soluciones:

La primera de ellas fue mantener el paquete ROS $mvsim$[2] dentro del flujo de la aplicación. Esto conllevaría: la actualización de dicho paquete con el seguro de hacerlo funcional con las dos versiones de MRPT y la creación de un modelado nuevo sobre el que realizar los experimentos.

La segunda alternativa fue utilizar el paquete $stage\_ros$, este paquete ofrece el simulador $stage$, así como la plataforma $player$. Para este simulador se partía de modelos compatibles con él. Sin embargo, no se contaba con soporte para cámaras RGBD dentro de él, por lo que a priori se asumía que las simulaciones con obstáculos en 3 dimensiones no serían posibles.

La tercera opción fue utilizar el simulador $gazebo$[3]. El simulador $Gazebo$ es uno de los productos principales de Open Robotics, siendo mantenido y actualizado con frecuencia[4]. Esta opción es claramente la más completa dada toda la funcionalidad que ofrece este simulador, (simulación dinámica, 3D, compatibilidad con los sensores más populares, computación en la nube...), además, ofrece una interfaz amigable y gráficos en 3D de última generación. Por otro lado, se contaban con modelos ya realizados con anterioridad para este simulador. Sin embargo, utilizarlo tiene unos requisitos hardware superiores a los del computador utilizado.

[1] `https://gcc.gnu.org/onlinedocs/gcc-3.0.1/cpp_13.html#SEC58`

[2] Fuente de imagen en 4.2: `https://github.com/ual-arm-ros-pkg/mvsim/tree/master/docs/imgs`

[3] Fuente de 4.2: `http://answers.gazebosim.org/question/9238/gazebo-starting-view-pose-control/`

[4] Estado de gazebo: `http://gazebosim.org/#status`

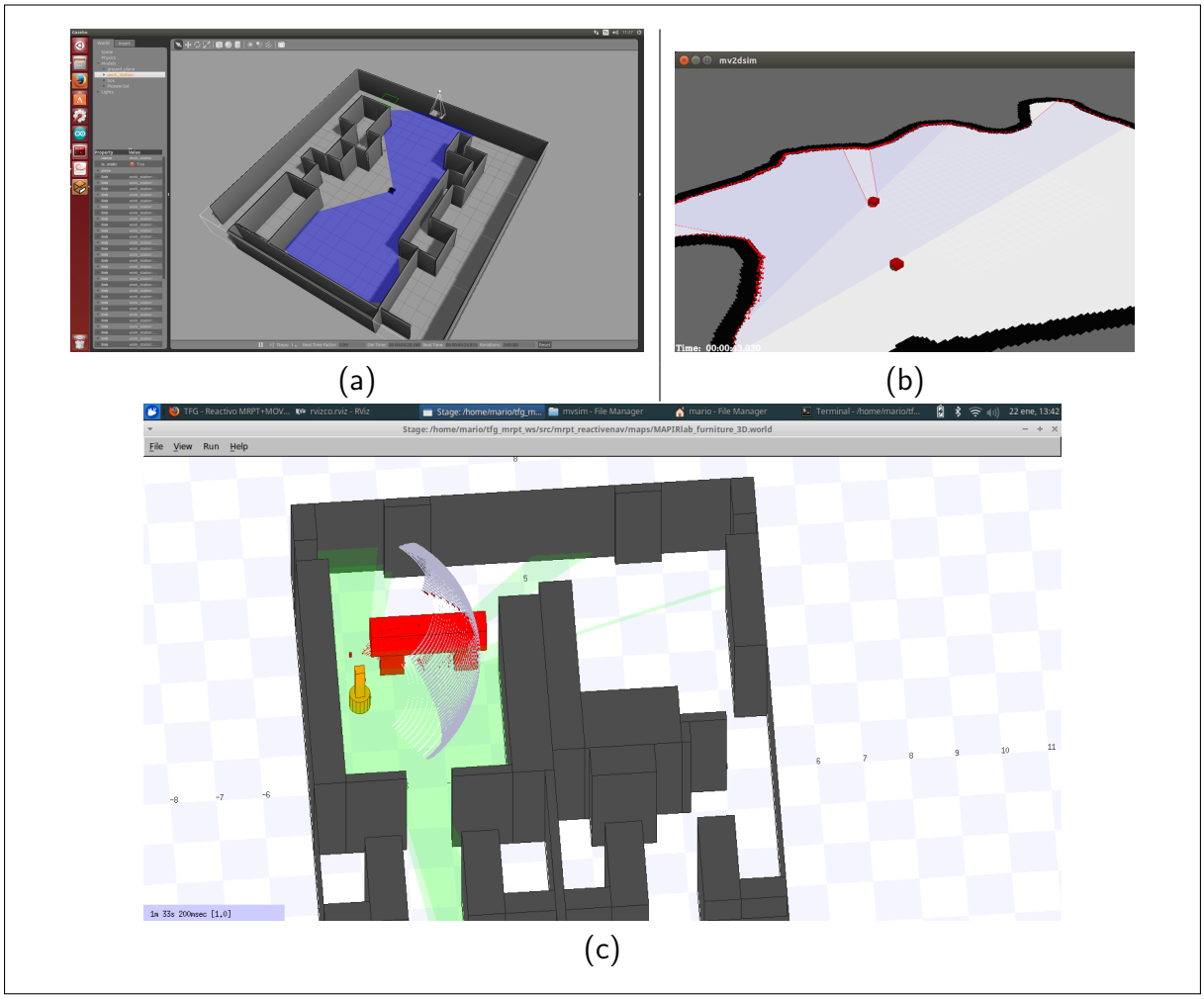


Figura 4.2: Interfaz gráfica de los tres simuladores robóticos planteados para tres escenas diferentes. (a) Gazebo, (b) Mvsim, (c) Stage.

Actualizar $mvsim$ tuvo que descartarse, ya que corregir la versión actual implicaba un conocimiento no superficial de la biblioteca de gráficos que emplea MRPT, quedando fuera del objetivo del presente proyecto.[1] Introducir $gazebo$ fue rechazado porque los requisitos técnicos no permitían la utilización del mismo de forma fluida. La opción elegida fue utilizar el simulador $Stage$, contar con el modelado de escenarios compatibles y ser un entorno ya familiar fueron los motivos determinantes. Si bien, se pensaba imposible realizar simulaciones para cámaras RGBD, al final resultó posible. Así, la elección del mismo no supuso una limitación en los resultados, como se pensaba en primera instancia.

4.2 Integración de MRPT con move_base

ROS pone a disposición un sistema de interfaces y plugins con el fin de hacer accesibles mecanismos sencillos de reutilización de código para el desarrollador.

Mediante este sistema, cualquier persona puede definir unos interfaces abstractos y toda una aplicación en base a los comportamientos derivados de ellos sin preocuparse por la implementación de los mismos. Esta es simplemente cargada al inicio de dicha aplicación utilizando una clase especializada en para cargar la implementación de la interfaz concreta.

Para el caso de integración que se enmarca en este proyecto, estos mecanismos resultan muy adecuados. En atención a la Figura 4.3, se observa el sistema de extensiones que interviene en el controlador de movimiento $move\_base$. Idealmente, se pretendería sustituir el módulo $local\_planner$ por el software de MRPT. Esto no resulta trivial dado que MRPT ofrece un sistema de navegación completo externo a ROS y que se adecúa a comportamientos diferentes.

Concretamente, la interfaz $BaseLocalPlanner$ define los siguientes comportamientos:

1. Computar comandos de velocidad. En una función se crearán los comandos de velocidad que serán transmitidos al controlador $move\_base$.

2. Comprobar si se ha alcanzado la meta. Esa función será utilizada por $move\_base$ para elevar el evento correspondiente y finalizar la navegación.

3. Guardar el camino global. Observando de nuevo la Figura 4.3, el planificador local escucha al planificador global. El mensaje intercambiado durante esta comunicación es el camino global. Como se indica, la comunicación es interna, no se realiza con topics y llamadas de interrupción, se realiza secuencialmente dentro del flujo de $move\_base$.

4. Debe tener una función de inicialización. Tomando un nombre para identificar el planificador, una referencia a tf y el mapa de costes local. Además, la clase debe contar con un constructor sin argumentos.

Se contaba con una aproximación del reactivo de MRPT basado en TPSpace a ROS en forma de paquete con dependencias a la librería. Este código se toma como punto de partida para el proyecto.[2] Se referirá en lo presente a este paquete como $mrpt\_reactivenav$.

[1] El progreso puede encontrarse en: `https://github.com/MariotoA/mvsim`

[2] Para ilustrar el estado inicial del proyecto, obsérvese `https://github.com/mrpt-ros-pkg/mrpt_navigation/tree/fbf5a784b439e31f20e1cf2bd8d77fb22f7c7a70/mrpt_reactivenav2d` más info en: `http://wiki.ros.org/mrpt_reactivenav2d?distro=kinetic`

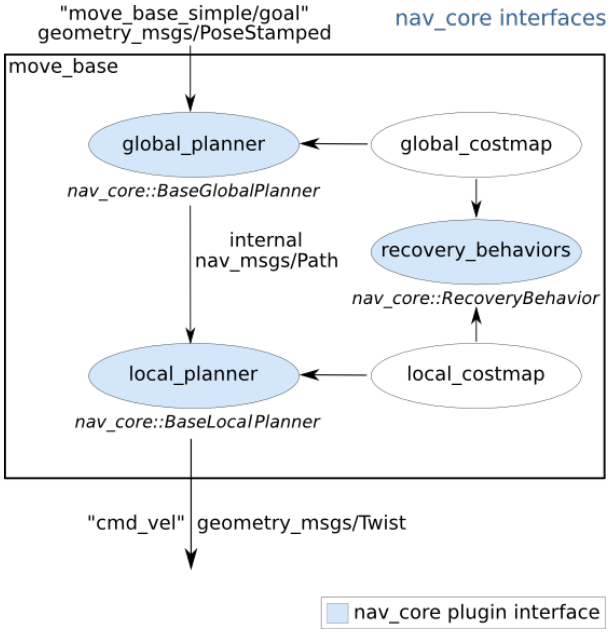


Figura 4.3: Sistema de plugins aceptados por $move\_base$. Señalados en azul.

MRPT cuenta con un meta-paquete destinado a la navegación en ROS que encapsula sus módulos más relevantes.[1] Dado que se opera dentro de ROS, muchos de ellos son sustituibles por los que otorga el meta-paquete $navigation$. Sin embargo, el paquete $mrpt\_obstacles$ está ligado fuertemente con el reactivo y su substitución no se contempla. Se trata de un nodo que obtiene la información de los sensores y las reúne en una nube de puntos durante una ventana de tiempo configurable.

Esto se debe a, por otro lado, que el algoritmo de navegación sobre $TPSpace$ de MRPT no trabaja con mapas de costes, trabaja con mapas de celdas de ocupación, además, de un formato no compatible. Por tanto, la información de los mapas de coste que se genera dentro de $move\_base$ no se puede utilizar para la navegación.

Teniendo esto en cuenta, se plantean tres vías diferentes de integración.

En la primera de ellas, se toma la decisión de ignorar toda la información proveniente de $move\_base$, tomar la información del $navigation$ necesaria mediante topics y trazar una ruta en base a la posición actual del robot, la meta y la información sensorial y de mapeado. Hágase notar que $mrpt\_reactivenav$ es un navegador de punto único y no recibe soporte para navegación sobre metas locales. MRPT, sin embargo, ofrece soporte para este tipo de navegación, por lo que sería necesaria la modificación del proceso de navegación que se invoca en el mismo, ya sea, para enviar de forma secuencial al reactivo los puntos clave de la ruta trazada o en forma de lote. El envío secuencial de metas requeriría un bucle de control externo a MRPT y el envío por lotes la creación de una estructura de datos adecuada para la transmisión de datos entre ROS y MRPT.

En la segunda, la comunicación con $move\_base$ se mantendría en cuanto a planificación global y comportamientos de recuperación, (Fig. 4.3). De esta forma, no sería necesario generar una ruta, ya se utilizaría la generada por el algoritmo de planificación global seleccionado. Requeriría sin embargo implementar las funciones de interrupción de necesarias para $move\_base$ de forma precisa.

[1] `http://wiki.ros.org/mrpt_navigation?distro=kinetic`

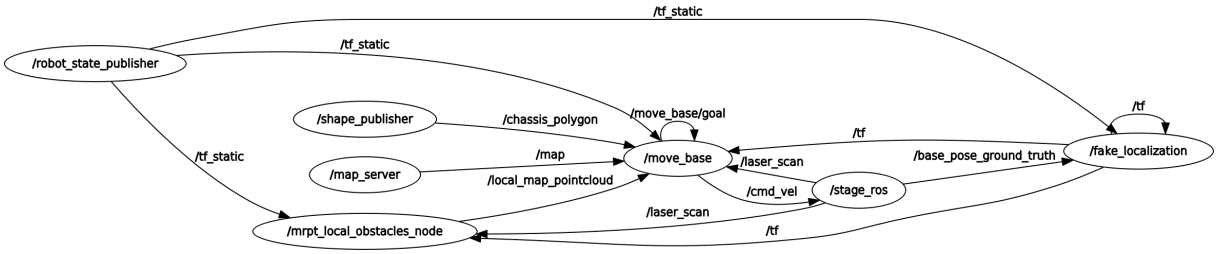


Figura 4.4: Estructura de nodos y topics que intervienen durante la aplicación.

La tercera forma, consistiría en una integración completa de $mrpt\_reactivenav$ en $move\_base$, requeriría idear una función que tradujese, o bien, los mapas de coste locales a mapas de celdas de ocupación y se transmitiesen a $move\_base$ en un formato aceptable. Otra opción sería adaptar el nodo reactivo para respetar los costes del mapa.

Por último cabe plantearse la exportación del plugin a $move\_base$. Puede modificarse $mrpt\_reactivenav$ para que implemente la interfaz, o bien crear una clase planificador que implemente el interfaz y que mediante composición tenga acceso al nodo reactivo de MRPT. De esta manera el código se encuentra más desligado y las funciones se pueden separar en dos niveles, el primero referente a $move\_base$ y el segundo al algoritmo reactivo en sí. Esta estructura es la utilizada por los planificadores $DWAPlanner$ y $TrajectoryPlanner$, por lo que además pueden utilizarse como referente a la hora del desarrollo de la aplicación.

Dado que cada opción resulta más completa que la siguiente, el desarrollo software puede plantearse de forma incremental. Principalmente se llegó a completar hasta la opción segunda y con el plugin como envoltorio del propio nodo reactivo de MRPT. En la Figura 4.4, pueden observarse los nodos que participan en el ciclo de vida de la navegación. Puede observarse que la comunicación entre envoltorio y reactivo es mediando mensajes y topics, pues el envoltorio se encarga de enviar metas locales progresivamente. Nótese la desconexión entre el mapa local y el envoltorio, pues solamente es empleado por este para obtener una referencia a la pose del robot. Cabe destacar también la inclusión de $mrpt\_local\_obstacles$, que solamente se relaciona con $mrpt\_reactivenav$.

4.3 Extensión a navegador 3D

Como se comentaba durante el primer apartado de esta memoria, en el sistema de costmaps que utiliza $move\_base$ los obstáculos se comparan en altura con el robot y se selecciona el saliente que más peligro plantee según la forma definida en la configuración de los mapas de coste y se proyecta en el suelo.

Este enfoque, para robots cuya forma difiere a diferentes alturas, como GIRAFF (Fig. 5.3), circular en la base y variada en altura, resulta muy conservador. Un ejemplo de situación donde esto puede pasar factura es una habitación con alta densidad de obstáculo y que contiene mesas. Como se observa en la figura 4.5, un robot con forma variable teóricamente debería poder acceder a zonas bajo la mesa. Si el espacio escasease, como podría suceder en una casa pequeña o estudio, esto podría ser crucial para llevar a cabo la navegación.

El módulo de navegación reactiva de MRPT ofrece clases para modelar la altura en un entorno en tres dimensiones como bloques de espacio discreto. Para dotar al paquete ROS desarrollado en este proyecto de esta capacidad se tuvieron que llevar a cabo tres modificacio-

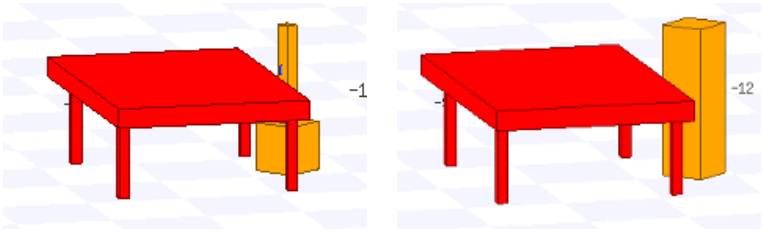
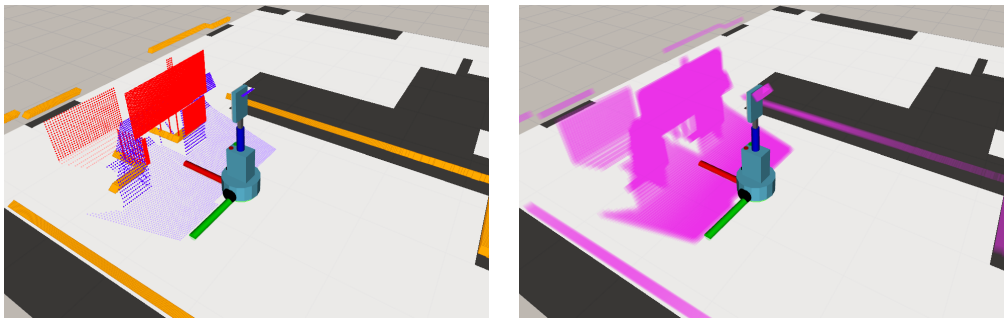


Figura 4.5: Representación de simulación en Stage de un robot uniforme (a) frente a un robot formado por dos prismas (b) al acercarse a una mesa con espacio libre bajo ella.



(a) (b)

Figura 4.6: Input (a) vs Output (b) para el nodo $mrpt\_local\_obstacles$.

nes:

- Substitución del sistema de navegación reactiva de 2D a 3D. Esto se traduce en cambiar la instancia de una clase por otra, pues la implementación ya está disponible en MRPT. [3]
- Desarrollo de un breve programa para la cargar en el servidor de parámetros de ROS la una forma de un robot robot tridimensional compuesta por bloques de puntos. Tras su lectura, traducción a la estructura de MRPT equivalente.
- Corrección del paquete $mrpt\_local\_obstacles$. Las nubes de puntos no se situaban correctamente en un entorno en 3 porque no se aplicaba una tf en el frame correcto. También existían incompatibilidades con la versión de MRPT.

Para llevar a cabo la integración del mismo con cámaras RGBD en simulación, se tuvo que emplear un nodo encargado unificar en una nube de puntos [36] la información de la cámara publicada por el simulador Stage. En la Figura 4.6, puede observarse el trabajo de unión de información sensorial que realiza ahora $mrpt\_local\_obstacles$ gracias a las modificaciones.

Capítulo 5

Validación experimental y Discusión de resultados

5.1 Fase de experimentación en 2D

5.1.1 Simulación

Uno de los objetivos de este trabajo consiste en establecer una comparación entre el planificador local que se ha integrado en el proyecto y el algoritmo reactivo DWA de ROS, ampliamente utilizado, referente dentro de la comunidad ROS.

Para ello, se crea una serie de escenarios y se observa el comportamiento de ambos planificadores reactivos para una configuración dada. En el caso de la figura 5.1, cada reactivo ha sido equipado con configuraciones de ya probada eficacia. Para el caso de $mrpt\_reactivenav$, se ha restringido su configuración al uso de trayectorias circulares para partir de la misma base. Se observa que el camino recorrido por DWA es más suave y preciso que MRPT. Sin embargo, durante la ejecución del caso (b), se pudo observar que presenta problemas cuando aparecen numerosos obstáculos en el camino (representados como cajas en la simulación). La presencia de obstáculos obliga al robot a disminuir su velocidad en torno a ellos, esto puede ocasionar que el robot se bloquee si entra dentro del radio de coste de un obstáculo dependiendo de su configuración. Esto también sucede para el reactivo de MRPT en cuanto a la cercanía a los obstáculos en lugar de al radio de hinflado. Sin embargo, no tuvo lugar en este ejemplo debido a, de nuevo, presentar una configuración menos conservadora que la configuración de DWA.

En la Figura 5.2, se observa una escena menos favorable para la navegación. En esta escena, las rutas trazadas computadas más arriesgadas (*challenging*) para los algoritmos reactivos por su cercanía a los obstáculos del mapa. Es en estos casos donde MRPT puede sacar a relucir su cualidad de navegar utilizando trayectorias no circulares. Para este ejemplo, se configuró un reactivo con una PTG circular y una PTG α-asintótica. Esta última PTG es seleccionada cuando existen abundantes obstáculos en el rango de visión del robot. Permite al robot ejecutar giros muy cerrados y acelerar cuando se alinea con la trayectoria, escapando de obstáculos. Estos giros pueden observarse en la gráfica roja de la figura, donde quedan representados por las subidas y bajadas rápidas.

Sin embargo, en las gráficas puede observarse una anomalía, la sucesión de intervalos de velocidad constante. Esto sucede por la estructura paralela de los nodos $mrpt\_reactivenav$ y $move\_base$. Donde, al ser el algoritmo TPSpace lento en su naturaleza cuando aumenta el

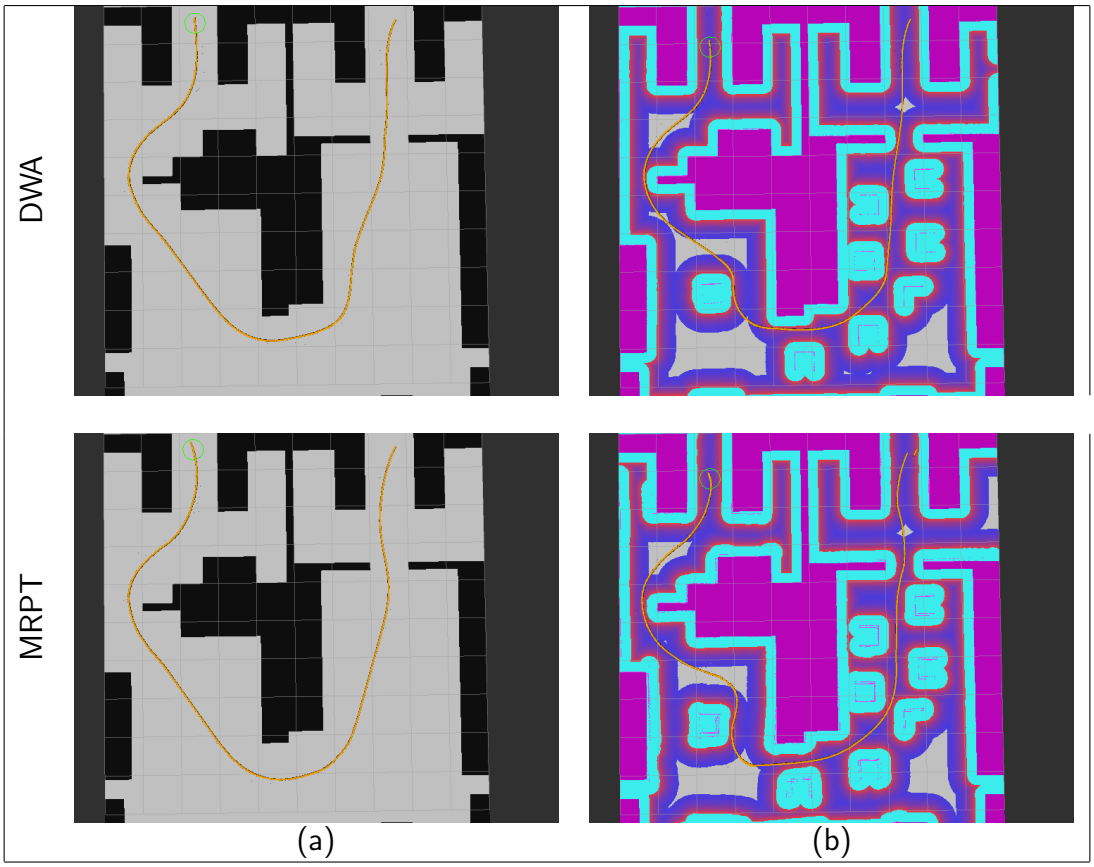


Figura 5.1: Recorrido del robot en 2 simulaciones distintas para dos algoritmos de navegación reactiva diferentes. (a) Escena sin obstáculos. (b) Escena con obstáculos.

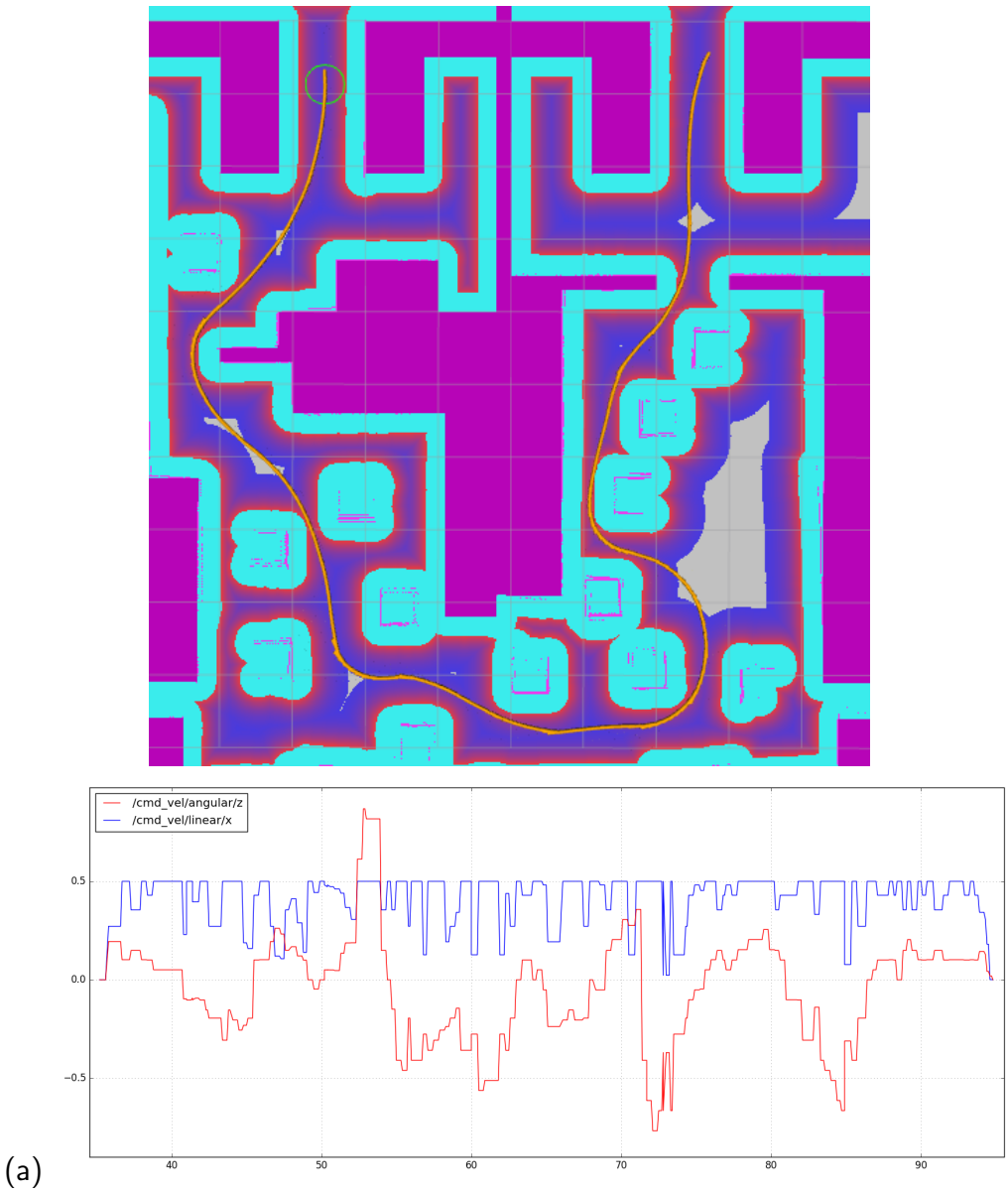
número de PTGs, no llega a actualizarse el comando de velocidad lo suficientemente rápido para $move\_base$. Por lo que el movimiento permanece constante hasta el siguiente comando generado por $mrpt\_reactivenav$. Esto provoca imprecisiones a la hora de alcanzar ciertos objetivos cuando el número de PTGs aumenta. Señalar que en la Figura 5.2 se ha omitido DWA porque no llegaba a la meta.

5.1.2 Ejecución en robot real

Para realizar experimentos en el mundo real, sería necesario elegir una plataforma móvil hardware sobre la que ejecutar los algoritmos presentados. Sin embargo, debido al elevado coste asociado al hardware robótico y a limitaciones de acceso al mismo, no se llevó una elección y en su lugar se utilizó el material disponible.

Las pruebas fueron llevadas a cabo en el laboratorio del grupo de investigación MAPIR[1] del departamento de Ingeniería de Sistemas y Automática de la Universidad de Málaga, que se encuentra en el módulo 2 de la Escuela Técnica Superior de Informática y Telecomunicaciones. Dicho laboratorio se encuentra poblado de puestos de trabajo, muebles, puertas y personas, por lo que resulta un ambiente convulso en el que para el robot se presentan muchos obstáculos, tanto estáticos como en movimiento. Por otra parte, este es el tipo de ambiente para el que están destinados los robots de servicio, no resulta inadecuado, aunque sí que presenta dificultades añadidas, utilizar estas habitaciones como escenarios de pruebas.

[1] `http://mapir.isa.uma.es/`



(a) (b)

Figura 5.2: (a) Recorrido del robot utilizando el plugin propuesto con una configuración que incluye α-trayectorias. (b) Comandos de velocidad durante la ejecución del recorrido. Velocidad lineal (azul) Velocidad angular (rojo)

(a) (b) (c)

Figura 5.3: Hardware robótico empleado durante las pruebas en el mundo real. (a) Robot *GIRAFF* de frente. (b) De perfil. (c) Láser *Hokuyo*

La plataforma robótica motorizada disponible para este proyecto es el robot *GIRAFF*. *GIRAFF* (Fig. 5.3 (a,b)) es un robot móvil de telepresencia con el objetivo de ofrecer apoyo a personas mayores o con discapacidad. Para interaccionar con personas cuenta con módulos de conversación y detección de personas y reconocimiento de caras, aunque todo esto escapa a la finalidad de este TFG y por tanto solo se utilizará su módulo relacionado con la navegación, esto son los drivers de los motores. Este robot está equipado con odómetros en las ruedas, útiles para estimar el movimiento del robot, sin embargo, son imprecisos, por lo que se utiliza una alternativa, la odometría láser.

Como fuente de información sensorial se utilizó el láser Hokuyo (Fig. 5.3 (c)), que provee un rango de 240° a una resolución de 0,352°, una profundidad de 5,6 m, con unas incertidumbres aceptables. Hokuyo es un láser ampliamente utilizado por ser relativamente barato, con un precio en torno a los 1000€[1].

Para medir la efectividad de los algoritmos de planificación de movimiento reactivo, se han establecido cuatro situaciones de interés: espacio vacío, obstáculos alrededor del camino, obstáculos en el camino y cruzar una puerta. Para llevar a cabo estos experimentos se ha utilizado un mapa estático del módulo 3.2 y el laboratorio 3.2.6 generado a partir de información sensorial láser. Esto provoca que sillas y mesas no están representadas salvo por sus patas. (Fig. 5.4) El espacio transitable se representa de color blanco, los puntos obstáculo de color negro y gris la zona indefinida.

Recorrido en espacio libre

En la Figura 5.5, puede observarse el recorrido que realiza el robot GIRAFF a lo largo del pasillo del módulo 3.2. Idealmente, al ser un pasillo sin muchos obstáculos y la meta situarse frente a él, el recorrido debería ser rectilíneo. Se puede observar que el algoritmo que más se acerca a esta trayectoria es el propuesto (b), mientras que DWA produce curvaturas durante la mitad de recorrido así como en el principio. La diferencia de tiempo que se muestra en la figura no sirve para extrapolar conclusiones pero se explica por el recorrido curvo que hace en el inicio (a).

[1]Fuente de los datos así como de la imagen: `https://www.roscomponents.com/en/lidar-laser-scanner/83-urg-04lx-ug01.html`

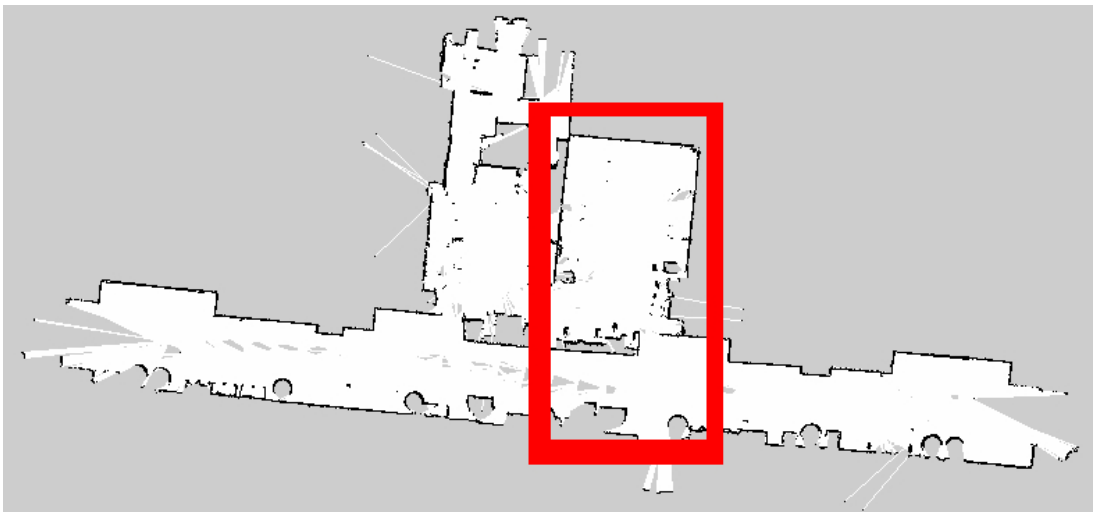
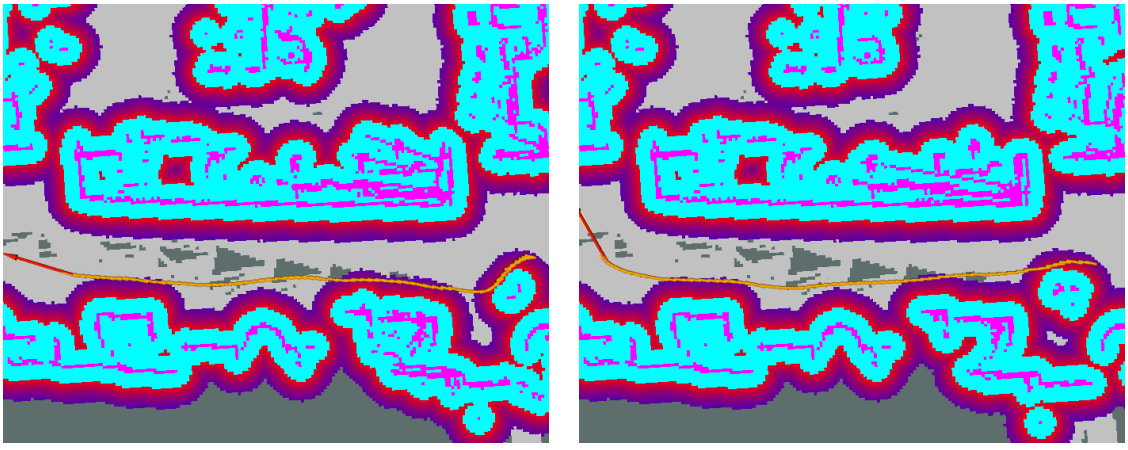
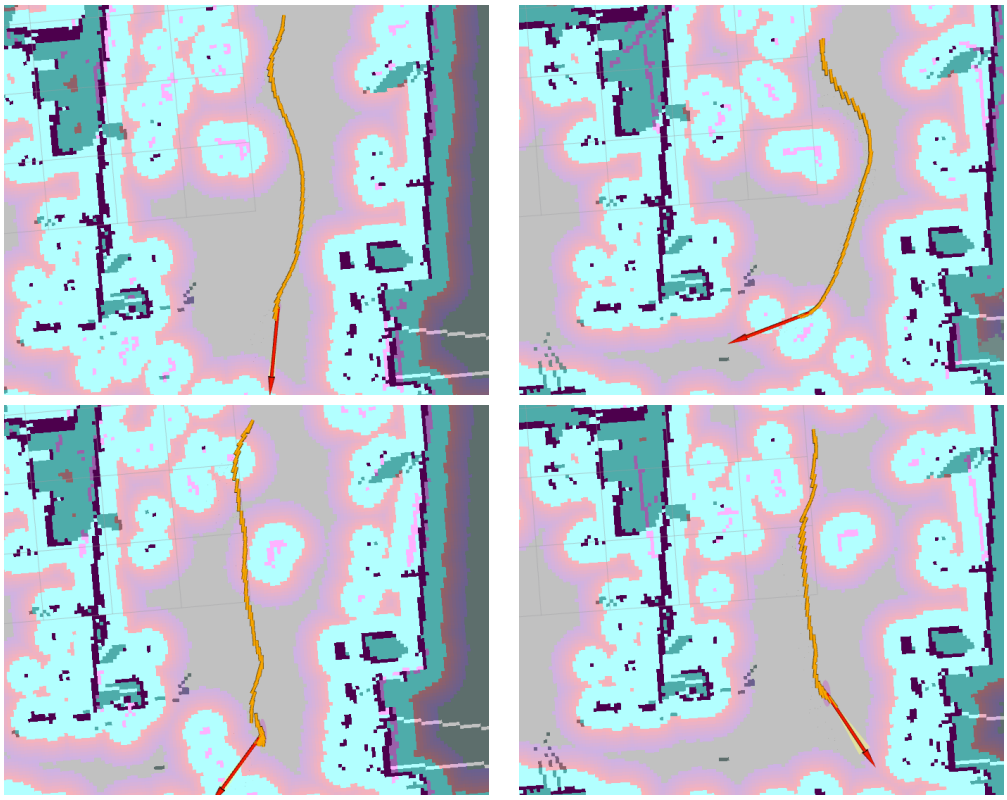


Figura 5.4: Mapa utilizado en las pruebas sobre el robot real. (Cuadro rojo) Zona concreta de los experimentos.



(a) $t = 39{,}42$ s (b) $t = 36{,}72$ s

Figura 5.5: Caso de recorrido de espacio libre anotando cada tiempo de llegada. (a) DWA. (b) MRPT



(a) (b)

Figura 5.6: Caso de recorrido con obstáculos. (a) DWA. (b) MRPT

Recorrido con obstáculo

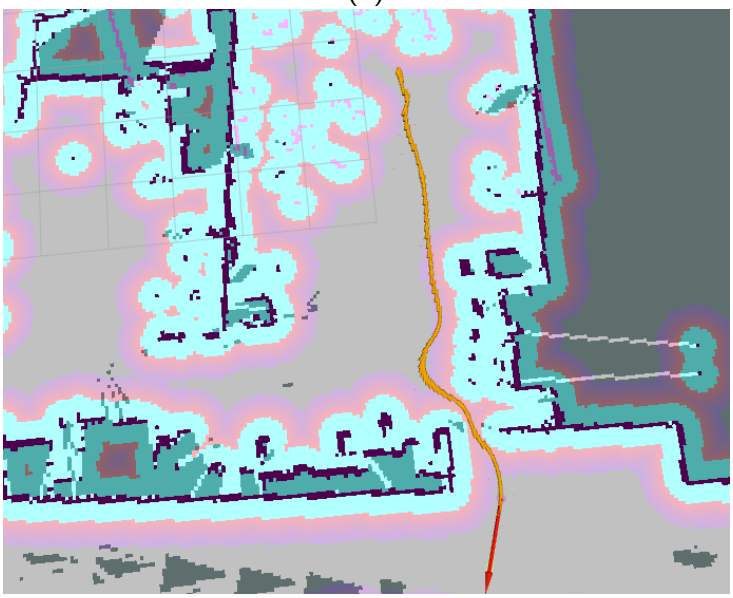
En la mitad del mapa, en la Figura 5.6, puede observarse un área de costes en forma de cuadrilátero. Este espacio representa el coste de acercarse a una caja de cartón de dimensiones (52.5x52.5x24.3) que se ha colocado a la izquierda de la supuesta ruta que trazará el algoritmo en la fila superior y a la derecha en la fila inferior.

Para (Fig. 5.6 (a)), puede observarse que el recorrido se compone de pequeñas curvaturas, no así (b), que tiende a generar a alejarse de los obstáculos con curvaturas más amplias. La configuración de MRPT utiliza PTG circulares con una gran disparidad entre velocidad angular y lineal, siendo la primera llegar el máximo permitido por el robot y la segunda una velocidad manejable a la que poder modificar la trayectoria con seguridad.

Recorrido atravesando una puerta

En la Figura 5.7, puede observase el recorrido que realiza GIRAFF para el reactivo DWA (a) y el reactivo basado en TPSpace (b). Durante las diferentes ejecuciones, se observó que DWA sufre problemas a la hora de atravesar esta puerta concreta. En algunas situaciones, el robot colisionaba contra la puerta abierta. En cuanto al reactivo de MRPT, con PTGs circulares en ocasiones el robot no lograba atravesar la puerta. Tras varias sesiones de experimentación y modificaciones en la configuración, se concluyó que este comportamiento estaba ligado a la selección del movimiento holonómico óptimo. En esta fase del algoritmo, tenía un gran peso la búsqueda de trayectorias en direcciones a espacio libre. Otro factor que entraba en juego era la gran diferencia entre velocidad angular máxima y velocidad linear máxima permitida por

(a)



(b)

Figura 5.7: Caso de transición entre habitaciones. (a) DWA. (b) MRPT

la PTG. Esto ocasiona que, si el número de muestras no es grande y el espacio libre escasea, las trayectorias ocasionen colisiones y se descarten. Obsérvese que si no hay espacio, menor es la probabilidad de encontrar una trayectoria válida si se disminuye el número de estas y la variación es grande.

Como solución a este problema, se introdujo una PTG α-asintótica. Estos son los resultados mostrados en la figura. Puede observarse que, al acercarse a los objetos, como sucedía en simulación, la PTG asintótica entra en juego y se hace notar por giros bruscos en el recorrido.

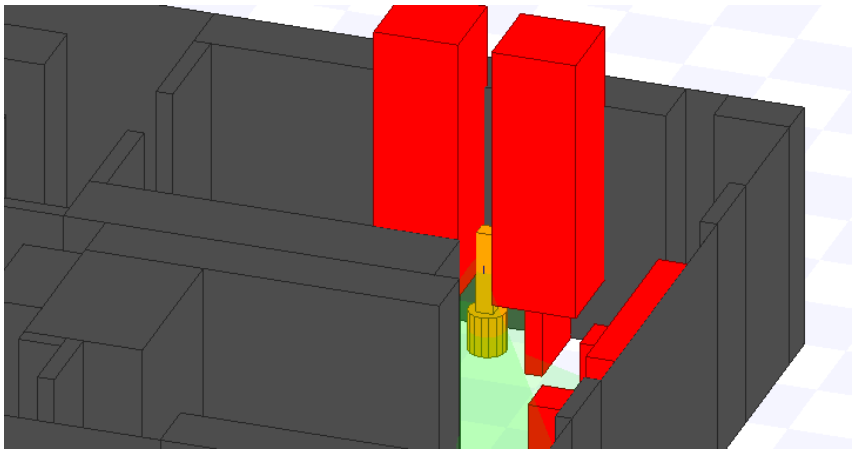


Figura 5.8: Modelado de una escena 3D usando Stage ajustada para dos lásers.

5.2 Fase de experimentación en 3D

5.2.1 Simulación

Tras extender el plugin reactivo para soportar 3D, queda elegir el tipo de sensores que utilizar para realizar la navegación. Primero, se empleó un modelo compuesto de dos bloques, la base robótica, un círculo aproximado y la parte superior, un prisma alargado y estrecho. Para simplificar, cada bloque se equipó con un sensor de rango láser.

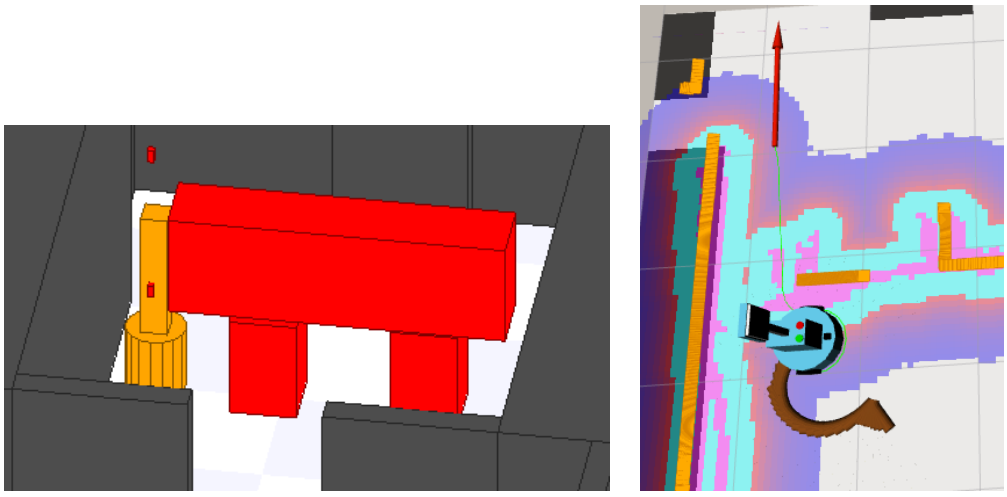
Este modelo era de utilidad debido a su sencillez, sin embargo, dos escáner láser no son suficiente para cubrir la altura del robot. Para hacer pruebas en ese entonces, era necesario exagerar las proporciones de los obstáculos. (Fig. 5.8)

Utilizar una cámara RGBD permite obtener un escenario de nubes de puntos representando el mundo observado a unas proporciones precisas dependiendo de su calibración. Aunque en el caso simulado, esta es perfecta. Sin embargo, como se mostrará en el siguiente apartado, el rango de visión de una cámara no es tan amplio como el de un escáner láser. Por esto, con la finalidad de abarcar mayor campo de visión, es común equipar a los robots con más de una.

En la Figura 5.9, puede observarse en (a) una escena creada para que la forma del robot pueda atravesarla pero solo considerando toda su altura. En (b) se encuentran los resultados de una ejecución de DWA utilizando dos cámaras RGBD. Como se puede apreciar, el robot no puede esquivar el obstáculo por culpa del mapa de costes, que como proyecta los obstáculos más restrictivos a lo largo de la altura en el suelo y toma como forma del robot su base proyectada a lo alto, no puede pasar.

Finalmente, en la Figura 5.10, se muestra como con el plugin propuesto, el robot es capaz de sortear el obstáculo con éxito. No obstante, esto solo tiene valor teórico, en el mundo real rara vez se van a encontrar obstáculos hechos a medida para ser esquivados.

Para llevar a cabo esto, ha sido necesario restringir el uso de PTGs a una de tipo α-asintótica. También tuvo que reducirse la resolución de las cámaras a 80x64 píxeles. Además, tuvo que situarse al robot alineado con el espacio libre, pues, por la influencia del obstáculo en sí y la pared, el robot se tendía a alejar del camino.



(a) (b)

Figura 5.9: Incapacidad de DWA de surcar un obstáculo tridimensional debido no tener en cuenta su forma. (a) Escenario (b) Situación

5.2.2 Ejecución en robot real

Debido a limitaciones de acceso, tiempo, escasez de material disponible y problemas de rendimiento del reactivo, no se ha podido llevar a cabo la validación de los experimentos sobre un robot real. Sin embargo, sí que se ha realizado todo el proceso hasta el punto de llegar a desplegar el sistema sobre el robot.

Como una de las cámaras a las que se tenía acceso estaba rota, solamente se equipó al robot con una de ellas. El modelo de cámara utilizado es el que se encuentra en la Figura 5.11 (a), posee un campo de visión de 58x45 grados y una distancia máxima de 3,5 metros. Con resoluciones aceptadas de (640x480) : 30fps y (320x240): 60fps.[1] La cámara se configuró en la posición que se muestra en la Figura 5.11 (b), sin embargo, en (c) se muestra la que se habría de utilizar. La cámara superior apunta hacia el suelo para observar los obstáculos cercanos al robot y la inferior para obstáculos lejanos y tareas no relacionadas.

Una vez configurada la cámara, es necesario calibrarla [37]. Como ROS por defecto permite utilizar parámetros por defecto como matriz de parámetros intrínsecos, la calibración necesaria es extrínseca. Esto es, encontrar la de rotación-traslación que relaciona la posición de la cámara con la base del robot. Para simplificar y acelerar el proceso, esta calibración se realizó a mano.

Debido a no poder escalar la nube de puntos a una resolución aceptable por $mrpt\_navigation$, solo se pudo validar para DWA. (Fig. 5.12) Según se puede observar, las observaciones parecen situadas en el mapa de forma aceptable y el robot logró pasar la puerta. La calibración extrínsica es válida, por tanto.

[1] Fuente: `https://www.asus.com/me-en/3D-Sensor/Xtion_PRO/specifications/`

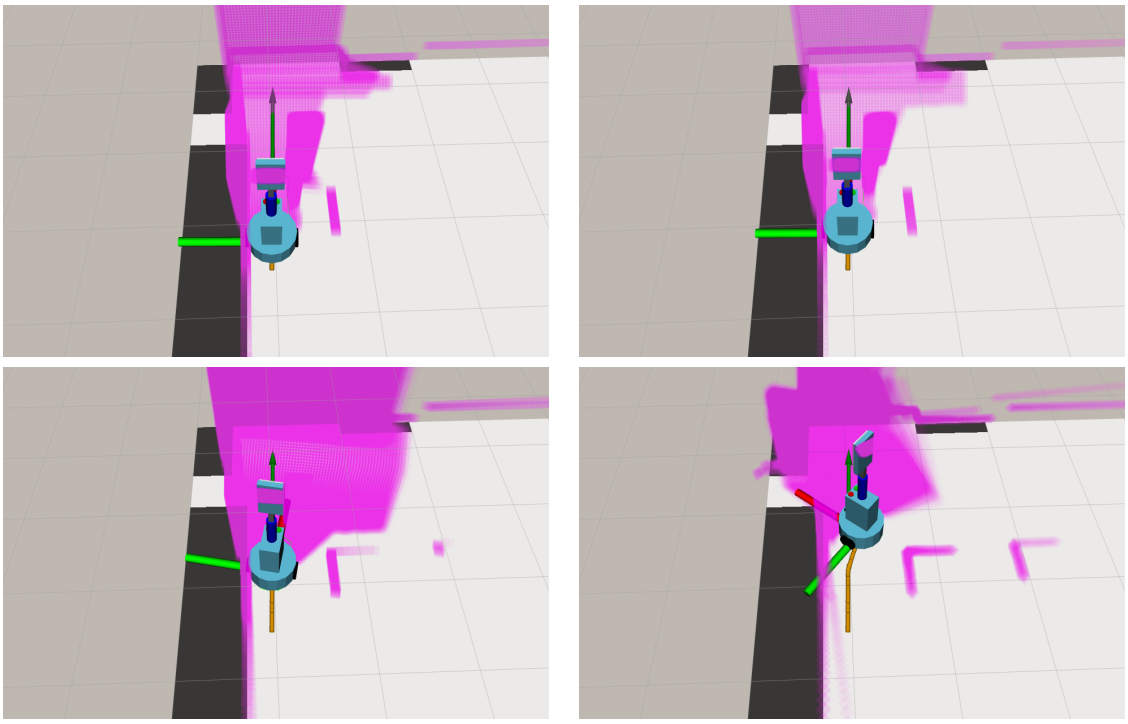


Figura 5.10: Reactivo 3D propuesto sorteando un obstáculo tridimensional en una escena preparada.

(a) (b) (c)

Figura 5.11: (a) Cámara ASUS Xtion PRO. (b) Configuración de dicha cámara usada en GIRAFF. (c) Configuración buscada

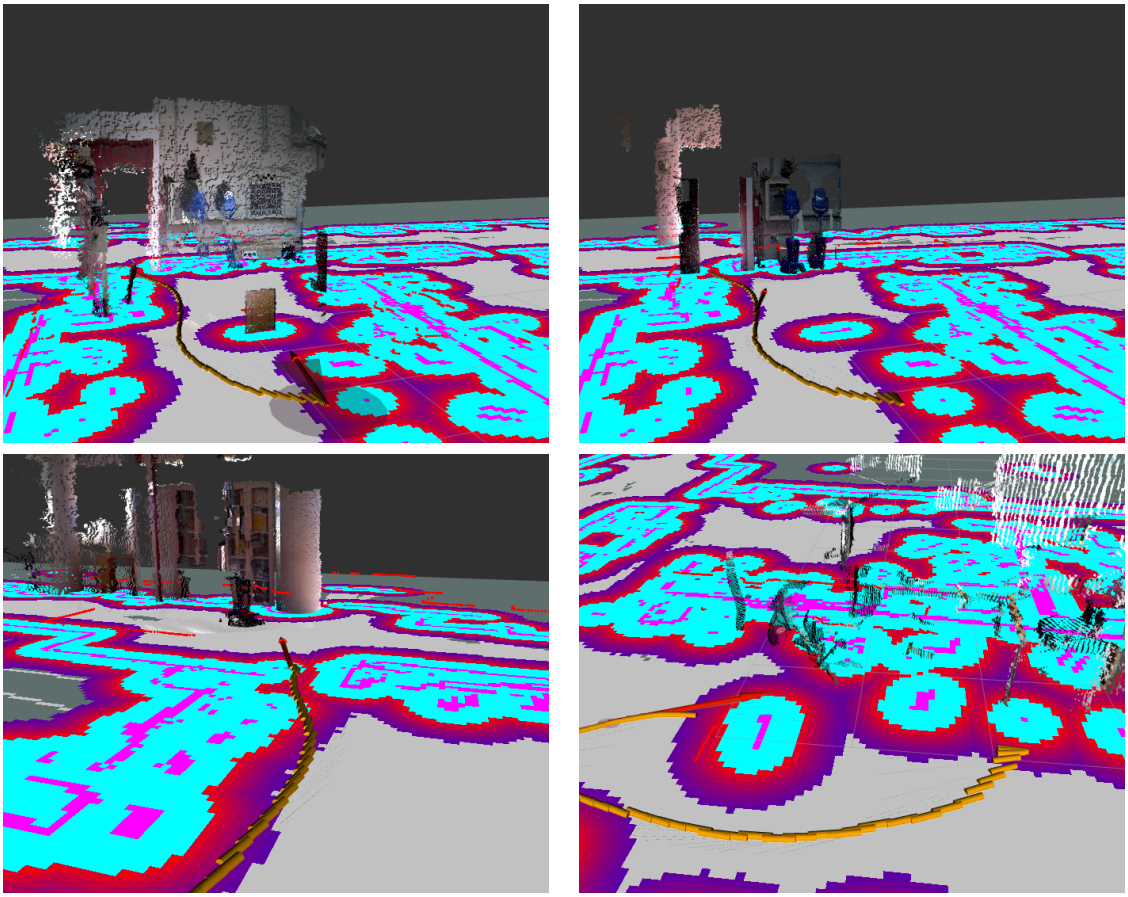


Figura 5.12: Recorrido de ida y vuelta hacia la puerta usando DWA con una cámara RGBD

Capítulo 6

Conclusiones y líneas futuras

En este trabajo se presenta una propuesta de integración del sistema de navegación reactiva basado en aplicar algoritmos de movimiento holonómico en el espacio de muestreo de trayectorias no necesariamente circulares (TPSpace) de MRPT en el meta-paquete $navigation$ de ROS. Una vez implementado, se ha realizado una comparación con el navegador reactivo DWA debido a ser el más utilizado dentro de ROS.

Se ha comprobado por la vía experimental tanto en simulación como en un robot real que las trayectorias no circulares, concretamente α-asintóticas, mejoran la navegación en ambientes con alta densidad de obstáculos. Estas trayectorias ofrecen al vehículo la capacidad de dar un giro cerrado y acelerar súbitamente dirigiéndose hacia espacio libre.

Por otra parte, se ha comprobado que la estrategia de dirigir la navegación reactiva según el envío continuo de metas intermedias extraídas de una ruta global computada paralelamente resulta adecuada para ayudar al algoritmo reactivo a sortear los obstáculos de la escena y alcanzar la meta. No obstante, esta estrategia presenta problemas cuando la meta se sitúa en la dirección opuesta al reactivo en espacios muy cerrados, provocando que, al no poder girar sobre sí mismo, el robot deambule hasta encontrar una zona abierta donde girar u otro camino para alcanzar la meta, produciendo recorridos muy por debajo del óptimo. DWAPlanner, para evitar esto, guarda un subcamino en lugar de un único punto y permite ajustar la trayectoria para parecerse a este subcamino.

En ambientes sin obstáculos, generalmente la implementación sugerida alcanza la meta en un menor tiempo debido a la habitual selección de trayectorias no lineales por DWA en lugar de lineales, favorecidas en el caso de MRPT.

También se ha comprobado que el sistema diseñado produce comandos de velocidad por debajo del ratio que $move\_base$ puede procesar. La solución propuesta opta por mantener el comando de velocidad enviado por el reactivo hacia $move\_base$ hasta recibir el siguiente. Por encima de tres PTG, este problema empieza a a no ser sostenible, pues el tiempo entre comandos de velocidad crece demasiado y pueden producirse colisiones.

Como extensión, este trabajo incluye la integración de la vertiente del mismo algoritmo para 3 dimensiones. Sin embargo, aunque al principio se planteó, no fue posible la utilización de dos cámaras RGBD en conjunción con el reactivo por limitaciones de acceso al material. En su lugar, se planteó la integración con una sola cámara. Sin embargo, esto solamente resultó de utilidad para el algoritmo DWA de ROS. La nube de puntos ralentizaba demasiado el cómputo al igual que el aumento en número de PTGs. En su lugar, se presenta un experimento en simulación con una situación favorable para el modelado de la escena a diferentes alturas para

comprobar su efectividad.

6.1 Vías futuras

En base a las conclusiones anteriores es posible establecer las siguientes vías de desarrollo para el futuro:

- Problema de deambular en espacios muy cerrados de forma poco óptima. Una posible solución sería adaptar el algoritmo para poder tratar subcaminos como DWA. Otra solución, quizá más asequible, sería la adecuada configuración de una PTG de tipo $C \mid C$, introducida en [38] y disponible en MRPT.
- El aumento del número de PTGs utilizables por el algoritmo. En la actualidad, el procesado de las PTG es secuencial dentro de MRPT. Sustituir la secuencia por paralelización podría mejorar el rendimiento del reactivo, permitiendo incorporar mayor número de trayectorias.
- Optimización del tratamiento de la nube de puntos de $mrpt\_local\_obstacles$. Una vía sencilla sería la transformación de este nodo en nodelet. La solución más adecuada quizá sería la creación de un nodelet ROS que de forma eficiente segmentase la nube de puntos en diferentes topics láser según la altura y sus mayores restricciones.
- Alineamiento con la última meta. En la actualidad, MRPT presenta un fallo en el procesamiento de los parámetros de navegación[1] que provoca la no terminación de la navegación. La solución de este problema abriría la puerta para lograr el tratamiento de metas como poses en el plano y no como posiciones.
- El desarrollo de una interfaz gráfica compatible con ROS y en concreto rviz para facilitar la labor de configurar las PTG.

Una vez explorados estos caminos se podrían plantear los siguientes:

- Modificación del algoritmo TPSpace para el uso de mapas de coste en lugar de mapas de celdas de ocupación.
- Calibración de 2 cámaras RGBD para la recreación precisa de escenas 3D durante la navegación.
- Evaluación del rendimiento del algoritmo reactivo frente a obstáculos móviles

[1] Fuente del fallo: `https://github.com/mrpt-ros-pkg/mrpt_navigation/issues/98#issuecomment-410959627`

Bibliografía

[1] M. O. Tătar, F. Haiduc, A. Szalontai, and A. Pop, "Design and development of the synchronous mobile robots," in *2016 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*, May 2016, pp. 1–6.

[2] J.-L. Blanco, J. González, and J.-A. Fernández-Madrigal, "Extending obstacle avoidance methods through multiple parameter-space transformations," *Autonomous Robots*, vol. 24, no. 1, pp. 29–48, Jan 2008. [Online]. Available: https://doi.org/10.1007/s10514-007-9062-7

[3] M. Jaimez, J.-L. Blanco, and J. Gonzalez-Jimenez, "Efficient reactive navigation with exact collision determination for 3d robot shapes," *International Journal of Advanced Robotic Systems*, vol. 12, no. 63, 2015.

[4] T. Litman, *Autonomous vehicle implementation predictions*. Victoria Transport Policy Institute Victoria, Canada, 2017.

[5] K. Bimbraw, "Autonomous Cars: Past, Present and Future - A Review of the Developments in the Last Century, the Present Scenario and the Expected Future of Autonomous Vehicle Technology," *ICINCO 2015 - 12th International Conference on Informatics in Control, Automation and Robotics, Proceedings*, vol. 1, pp. 191–198, 01 2015.

[6] D. W. Matolak, "Unmanned aerial vehicles: Communications challenges and future aerial networking," in *2015 International Conference on Computing, Networking and Communications (ICNC)*, Feb 2015, pp. 567–572.

[7] G. A. Bekey, *Autonomous robots: from biological inspiration to implementation and control*. MIT press, 2005.

[8] R. G. Simmons, "Structured control for autonomous robots," *IEEE transactions on robotics and automation*, vol. 10, no. 1, pp. 34–43, 1994.

[9] J. R. Ruiz-Sarmiento, C. Galindo, J. Monroy, F.-A. Moreno, and J. Gonzalez-Jimenez, "Ontology-based conditional random fields for object recognition," *International Journal of Knowledge-Based Systems*, 2019.

[10] J. R. Ruiz-Sarmiento, C. Galindo, and J. Gonzalez-Jimenez, "A survey on learning approaches for undirected graphical models. application to scene object recognition," *International Journal of Approximate Reasoning*, vol. 83, pp. 434–451, apr 2017.

[11] M. Jaimez, J. Monroy, M. Lopez-Antequera, and J. Gonzalez-Jimenez, "Robust planar odometry based on symmetric range flow and multi-scan alignment," *IEEE Transactions on Robotics*, pp. 1623–1635, 2018. [Online]. Available: http://mapir.isa.uma.es/work/SRF-Odometry

[12] M. Jaimez, J. Monroy, and J. Gonzalez-Jimenez, "Planar odometry from a radial laser scanner. a range flow-based approach," in *IEEE International Conference on Robotics and Automation (ICRA)*, jun 2016, pp. 4479–4485. [Online]. Available: http://mapir.isa.uma.es/mapirwebsite/index.php/mapir-downloads/papers/217

[13] J. Monroy, J. R. Ruiz-Sarmiento, F.-A. Moreno, F. Melendez-Fernandez, C. Galindo, and J. Gonzalez-Jimenez, "A semantic-based gas source localization with a mobile robot combining vision and chemical sensing," *Sensors*, vol. 18, no. 12, 2018. [Online]. Available: https://www.mdpi.com/1424-8220/18/12/4174?type=check_update&version=1

[14] R. Brooks, "A robust layered control system for a mobile robot," *IEEE Journal on Robotics and Automation*, vol. 2, no. 1, pp. 14–23, March 1986.

[15] I. W. on the Synthesis, S. of Living Systems, C. Langton, and K. Shimohara, *Artificial Life V: Proceedings of the Fifth International Workshop on the Synthesis and Simulation of Living Systems*, ser. @Artificial life. PAPERBACKSHOP UK IMPORT, 1997, pp. 34–36. [Online]. Available: https://books.google.es/books?id=0J8kQEjXe38C

[16] M. Quigley, B. P. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Y. Ng, "Ros: An open-source robot operating system," *ICRA Workshop on Open Source Software*, vol. 3, pp. 1–6, 01 2009.

[17] J.-L. Blanco, J. Gonzalez-Jimenez, and J.-A. Fernández-Madrigal, *Foundations of Parameterized Trajectories-based Space Transformations for Obstacle Avoidance*, ser. Mobile Robots Motion Planning: New Challenges. I-Tech Education Publishing, 2008, ch. 2. [Online]. Available: http://www.intechopen.com/books/motion_planning/foundations_of_parameterized_trajectoriesbased_space_transformations_for_obstacle_avoidance

[18] ——, "The trajectory parameter space (tp-space): A new space representation for non-holonomic mobile robot reactive navigation," in *IEEE International Conference on Intelligent Robots and Systems (IROS'06)*, Oct. 2006.

[19] J.-L. e. a. Blanco. Mrpt: Mobile robot programming toolkit. [Online]. Available: https://mrpt.org

[20] R. Siegwart and I. R. Nourbakhsh, *Introduction to Autonomous Mobile Robots*. Scituate, MA, USA: Bradford Company, 2004, pp. 48, 63–65.

[21] D. Fox, W. Burgard, F. Dellaert, and S. Thrun, "Monte carlo localization: Efficient position estimation for mobile robots," 01 1999, pp. 343–349.

[22] H. Durrant-Whyte and T. Bailey, "Simultaneous localization and mapping: part i," *IEEE Robotics Automation Magazine*, vol. 13, no. 2, pp. 99–110, June 2006.

[23] L. Matthies and A. Elfes, "Integration of sonar and stereo range data using a grid-based representation," in *Proceedings. 1988 IEEE International Conference on Robotics and Automation*, April 1988, pp. 727–733 vol.2.

[24] D. Ferguson and M. Likhachev, "Efficiently using cost maps for planning complex maneuvers," in *in Proc. of the Workshop on Planning with Cost Maps, IEEE Int. Conf. on Robotics and Automation*, 2008.

[25] D. V. Lu, D. Hershberger, and W. D. Smart, "Layered costmaps for context-sensitive navigation," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Sept 2014, pp. 709–715.

[26] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numer. Math.*, vol. 1, no. 1, pp. 269–271, Dec. 1959. [Online]. Available: http://dx.doi.org/10.1007/BF01386390

[27] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *Intelligence/sigart Bulletin - SIGART*, vol. 37, pp. 28–29, 12 1972.

[28] M. L. Hetland, *Python Algorithms: Mastering Basic Algorithms in the Python Language*. Apress, 2010, p. 214.

[29] D. Fox, W. Burgard, and S. Thrun, "Controlling synchro-drive robots with the dynamic window approach to collision avoidance," in *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems. IROS '96*, vol. 3, Nov 1996, pp. 1280–1287 vol.3.

[30] ——, "The dynamic window approach to collision avoidance," *IEEE Robotics Automation Magazine*, vol. 4, no. 1, pp. 23–33, March 1997.

[31] Y.-C. Lin, C.-C. Chou, and F.-L. Lian, "Indoor robot navigation based on dwa*: Velocity space approach with region analysis," in *2009 ICCAS-SICE*, Aug 2009, pp. 700–705.

[32] Y. Kang, D. A. de Lima, and A. C. Victorino, "Dynamic obstacles avoidance based on image-based dynamic window approach for human-vehicle interaction," in *2015 IEEE Intelligent Vehicles Symposium (IV)*, June 2015, pp. 77–82.

[33] D. Fox, W. Burgard, S. Thrun, and A. B. Cremers, "A hybrid collision avoidance method for mobile robots," in *Proceedings. 1998 IEEE International Conference on Robotics and Automation (Cat. No.98CH36146)*, vol. 2, May 1998, pp. 1238–1243 vol.2.

[34] B. P. Gerkey and K. Konolige, "Planning and control in unstructured terrain," in *In Workshop on Path Planning on Costmaps, Proceedings of the IEEE International Conference on Robotics and Automation (ICRA*, 2008.

[35] P. Corke, *Robotics, Vision and Control: Fundamental Algorithms in MATLAB*, 1st ed. Springer Publishing Company, Incorporated, 2013, p. 101.

[36] R. B. Rusu and S. Cousins, "3d is here: Point cloud library (pcl)," in *2011 IEEE International Conference on Robotics and Automation*, May 2011, pp. 1–4.

[37] Q. Zhang and R. Pless, "Extrinsic calibration of a camera and laser range finder (improves camera calibration)," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, vol. 3, Sep. 2004, pp. 2301–2306 vol.3.

[38] M. Vendittelli, J. . Laumond, and C. Nissoux, "Obstacle distance for car-like robots," *IEEE Transactions on Robotics and Automation*, vol. 15, no. 4, pp. 678–691, Aug 1999.

Apéndice A

Requisitos técnicos e instalación

El código desarrollado para este TFG tiene como finalidad ejecutarse sobre **Ubuntu 16.04**, por lo que sería necesaria una máquina virtual de no contarse con un equipo con el software instalado.

Suponiendo que se alcance ese requisito, lo siguiente es instalar el framework de programación ROS:

```
# Instruccion para acceder al ppa de ROS
sudo sh -c 'echo "deb \
http://packages.ros.org/ros/ubuntu \
$(lsb_release -sc) main" > \
/etc/apt/sources.list.d/ros-latest.list'

# Verificacion con la clave del mismo
sudo apt-key adv --keyserver \
hkp://ha.pool.sks-keyservers.net:80 --recv-key \
421C365BD9FF1F717815A3895523BAEEB01FA116

sudo apt-get update
sudo apt-get install ros-kinetic-desktop-full

# Se encarga de inicializar las dependencias
# que puedan necesitar los paquetes ROS
sudo rosdep init
rosdep update

# Almacena el acceso a los paquetes ROS en PATH
# siempre que se abra un terminal.
echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

Una vez terminada la instalación, será necesario instalar MRPT. Existe un bug por el cual el compilador por defecto de Ubuntu 16 no es válido, por lo que puede ser antes de proceder con MRPT necesario instalar gcc y g++ versión 7.

```
# Optional, upgrade gcc/g++.
#No recomendable salvo si no hay otra:
add-apt-repository ppa:ubuntu-toolchain-r/test
apt-get update
apt-get install -y g++-7

# MRPT install:
sudo add-apt-repository ppa:joseluisblancoc/mrpt-1.5
sudo apt-get update
sudo apt-get install libmrpt-dev mrpt-apps
```

Ahora solo resta copiar el software del CD a una carpeta del disco, abrir un terminal en esta carpeta e introducir:

```
# Instalacion del simulador utilizado
sudo apt-get install ros-kinetic-stage

# Compilacion del codigo del tfg
catkin_make

# Hacerlo visible para el sistema
source devel/setup.bash
```

Y todo estaría listo. Para jugar con el reactivo en simulación explorar las carpetas *tfg_reactive_pkg* así como *mrpt_reactivenav*. Esta última contiene los archivos para lanzar una simulación con el plugin desarrollado. Para esto, simplemente introducir en terminal:

```
roslaunch mrpt_reactivenav [
test_simbot_navigation3D_cloud.launch |
test_simbot_navigation3D.launch |
test_simbot_navigation3D_only_laser.launch |
test_simbot_navigation.launch ] [map_file:=2cam]
```

Una vez haya emergido la ventana *rviz*. Es posible enviar una meta a *move_base* pulsando *2D Nav Goal* y pulsando de nuevo en el mapa.

De producirse un error en el programa stage si se ha instalado el compilador gcc7, ir a `/opt/ros/kinetic/include/Stage-4.1/stage.hh` y editar los "m"por "m". Tal cual.