Proyecto Fin de Grado
Ingeniería en electrónica, robótica y mecatrónica

Navegación sin mapa y mapeado en robótica móvil para entornos no estructurados

Autor: Luis García Montañés
Tutor: Carlos Bordons Alba

Dep. Ingeniería de sistemas y automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2017

Proyecto Fin de grado
Ingeniería en electrónica, robótica y mecatrónica

Navegación sin mapa y mapeado en robótica móvil para entornos no estructurados

Autor:
Luis García Montañés

Tutor:
Carlos Bordons Alba
Profesor catedrático

Dep. de Ingeniería de sistemas y automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2017

Proyecto Fin de Carrera: Navegación sin mapa y mapeado en robótica móvil para entornos no estructurados

Autor: Luis García Montañés

Tutor: Carlos Bordons Alba

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

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2017

El Secretario del Tribunal

A mi familia

A mis maestros

Agradecimientos

Si quieres mantener la felicidad, tienes que compartirla

- Dale Carnegie -

La robótica es un campo de la ingeniería que está en permanente evolución a día de hoy, con un futuro prometedor. Me siento afortunado de haber tenido la oportunidad de estudiar una carrera fascinante, y esto sin duda se lo debo a mis padres, a los grandes docentes, y a aquellos compañeros y amigos que he ido haciendo por el camino.

A todos aquellos que a diario dan lo mejor de sí a quienes les rodean.

Luis García Montañés

Sevilla, 2017

Resumen

Este trabajo se enmarca en el contexto del inicio de un proyecto de colaboración de 3 años de duración, de la US con el CSIRO (Commonwealth Scientific and Industrial Research Organization, Australia) y el instituto INTA (Instituto Nacional de Técnica Aeroespacial), en la investigación del uso de pilas de combustible en dos robots móviles, un rover y un submarino. El proyecto está bajo el nombre de IUFCV Project.

En primer lugar, se tratará el trabajo realizado en el robot submarino, y posteriormente se pasará al trabajo realizado en el rover SUMMIT-XL, que se ha hecho en la plataforma ROS (Robot Operative System).

Abstract

This work is framed in the context of the beggining of a 3-year collaborative proyect, from the US on behalf of CSIRO (Commonwealth Scientific and Industrial Research Organization, Australia) and INTA institute (National Institute of Aerospace Technology), which investigates the usage of fuel cells in two mobile robots, a rover and a submarine.

In the first place, work done in the submarine robot will be discussed, to then move on to the SUMMIT-XL rover, which has been done in the ROS platform (Robot Operative System).

Indice

ÍNDICE DE FIGURAS

Notación y glosario

ROS	*Robot Operative System*
SLAM	*Simultaneous Localization and Mapping*
SUMMIT XL	*Robot móvil (rover terrestre) fabricado por Robotnik*
Bleeper	*Robot móvil (submarino) fabricado por Praesentis*
IUFCV Project	*Proyecto de investigación: Improving efficiency and operational range in low-power unmanned vehicles through the use of hybrid fuel-cell power systems.*
Costmap	*Mapa de un terreno basado en cuadrículas con diferentes niveles de ocupación según la disposición de los obstáculos*
Shell	*Ventana de comandos de Ubuntu*
Ssh	*Protocolo seguro para comunicación entre PC y servidor remoto*

1 INTRODUCCIÓN

Los vehículos no tripulados ofrecen muchas posibilidades en aplicaciones científicas, siendo sistemas de adquisición de datos en tiempo real menos costosos, más respetuosos con el medio ambiente, y en muchos casos más seguros para las personas que otras técnicas tradicionales. Desafortunadamente, en la actualidad la mayor parte de vehículos autónomos presentan una autonomía limitada debido al uso de baterías tradicionales de polímero de litio.

Las pilas de combustible son un dispositivo electroquímico que convierte energía, produciendo electricidad mediante un proceso de oxidación, utilizando hidrógeno como combustible. La reacción produce agua, siendo un proceso con cero emisiones contaminantes, y además con una eficiencia de alrededor de un 60%, mucho más alta que un motor convencional, ya que no siguen un proceso termodinámico.

En colaboración con el CSIRO y el instituto INTA, se inicia un proyecto de colaboración donde se persigue investigar el uso de las pilas de combustible en robots móviles, con intención de aumentar su autonomía frente a las baterías de polímero de litio, investigación para la que se dispone de un robot submarino Bleeper y un rover terreste SUMMIT-XL.

1.1 Objetivo del trabajo

El CSIC puso en disposición de la universidad un robot submarino completamente desmontado, y necesitando un alto nivel de reconstrucción. Por otra parte el instituto INTA prestó a la universidad un SUMMIT XL en perfecto estado de funcionamiento, con todas las herramientas para poder hacer un desarrollo en ROS.

El objetivo a alcanzar en el Bleeper es realizar un sistema de teleoperación, contando el mismo con una sonda para el envío de información y con un control basado en microcontrolador.

Por parte del Summit XL, el objetivo es implementar un sistema de navegación sin necesidad de mapa, con objeto de que sea capaz de moverse sin colisionar en un entorno no estructurado, al mismo tiempo que obtiene un mapa de la zona. Para ello se utilizará un sistema de mapeo en tiempo real basado en SLAM y un sistema de navegación basado en mapas de coste.

1.2 Metodología

En lo que respecta al robot submarino, se hará un trabajo colaborativo entre compañeros del grupo involucrados en el proyecto de colaboración. En este documento se hará referencia al trabajo total realizado. Se hará una revisión del estado del submarino, realizando tareas puramente mecánicas como

puede ser la sustitución de los cierres del submarino para asegurar la estanqueidad. A continuación se hará una sustitución de componentes, instalando una cámara JPEG, controladores de potencia para los motores (puentes H), y otros elementos electrónicos. En último lugar se realizará el cableado y control del robot con el uso de un microcontrolador Arduino® y un mando inalámbrico.

En el rover terrestre, se hará un desarrollo en ROS Indigo, sistema que funciona bajo Ubuntu LTS 14.04. Este desarrollo perseguirá dotar de autonomía al robot, siendo capaz de mapear cualquier terreno desconocido y desplazarse por él, sin necesidad de ser teleoperado por una persona. Todo esto se hará con ayuda de las herramientas Gazebo, simulador usado principalemente para simulación y aprendizaje, y RViz, que se utiliza para visualización y monitorizado de datos en tiempo real.

2 BLEEPER (ROBOT SUBMARINO)

2.1. Introducción

Esta parte del proyecto tratará el trabajo realizado sobre el robot submarino. Este modelo cuenta con una cámara frontal, orientable verticalmente con servomotor, dos focos situados en el frontal del robot, y tres motores de corriente continua, dos para controlar el movimiento horizontal y uno de elevación. Originariamente el robot traía un cable umbilical para la corriente de alimentación y las señales de control, que se mandaban a través de un maletín.

El trabajo en este robot en colaboración con el compañero del proyecto Victor Cazalla.

2.2. Características del submarino

Ilustración 1: Bleeper de Praesentis

Las características del submarino que proporciona el fabricante son:

- Profundidad operativa de 50m
- Velocidad máxima de 0.7 m/s
- Potencia máxima de 500W
- Chasis de aluminio anodizado con carcasa de polímero técnico
- Temperaturas de trabajo entre -20º y 50º
- 10 Kg de peso
- Medidas: 480 x 395 x 225 mm

2.3. Restauración

El robot que llegó sin el maletín de control, completamente desmontado y con un informe relativo al estado del ROV elaborado por el CSIC, que orientaba sobre qué necesitaba sustituirse y qué funcionaba.

El primer paso fue realizar una evaluación del estado e inventario de piezas a sustituir. Entre ellas estaban:

- Cambio de tornillería oxidada

- Sustitución de las gomas de cierres de compartimentos.

- Componentes electrónicos.

Posteriormente se procedió a la limpieza y reensamblaje del robot.

2.4. Electrónica

Tras concluir la parte más mecánica, se hizo necesaria una actualización y puesta en funcionamiento de los componentes electrónicos del submarino.

2.3.1 Cámara

Tras probar la cámara en un monitor, se concluyó que la cámara que traía el robot estaba dañada. Se hizo necesario por tanto una sustitución por una nueva semejante. Se eligió un modelo JPEG con opción de retransmisión de video en directo (salida RCA) y toma de imágenes.

La cámara se instaló en su módulo, haciendo adaptaciones al soporte, debidas a pequeñas diferencias con la anterior. El cableado se canalizó junto con el del servo para poder extraerlo por la manguera de sellado del módulo.

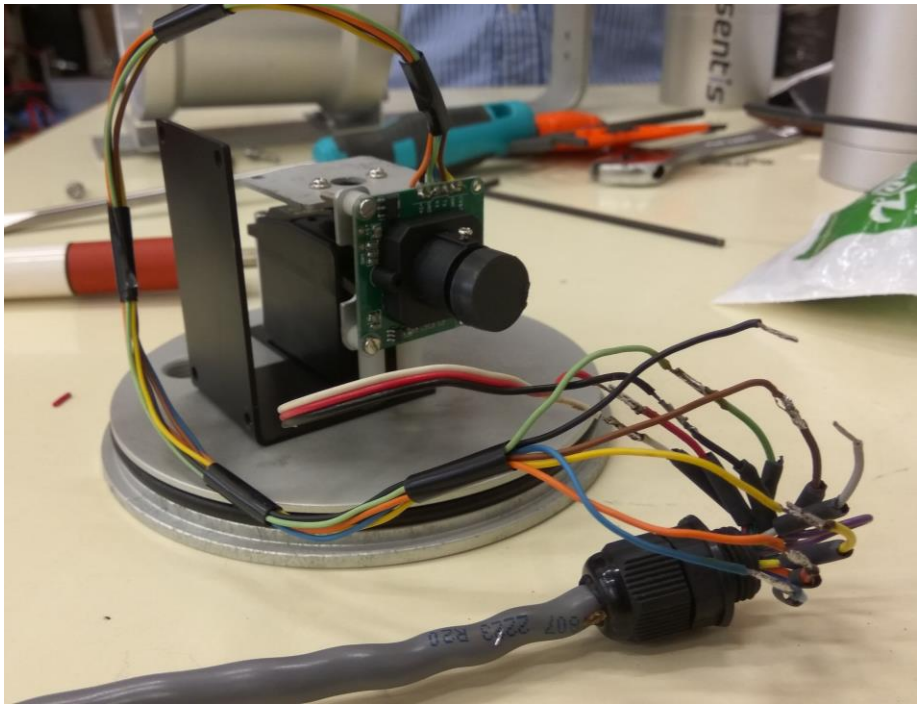


Ilustración 2: Montaje de la cámara con servo

2.3.2 Motores

Los motores se conectaron a unos controladores de media potencia en Puente H, modelo L298N. Además los cables estaban desconectados de dentro de la carcasa del motor, por tanto hubo que hacer las reconexiones de las bayonetas de cada motor.

Por otra parte, el motor de elevación tuvo que ser sustituido por mal funcionamiento, y hubo que buscar un modelo igual, y adaptar el eje en longitud y para fijación de la hélice.

2.3.3 Focos

Los focos traían unas lámparas halógenas, que suponían un alto consumo. Fueron sustituidas por dos lámparas LED de mayor potencia lumínica. Se añadió un ajuste de intensidad lumínica a través de Puente H.

2.5. Cableado final

Tras la reconstrucción e instalación de los componentes se hizo el conexionado y cableado final hasta la alimentación y sistemas de control externos (Arduino, puentes H y conector VGA).

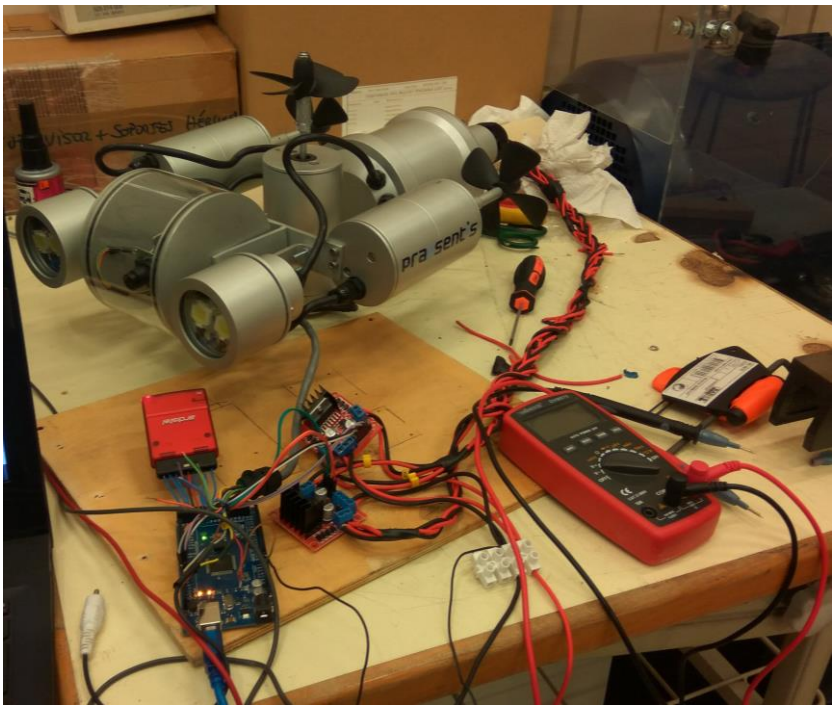


Ilustración 3: Bleeper con electrónica visible

2.6. Programa comentado para control remoto mediante microcontrolador

El control remoto se realizó mediante un mando inalámbrico de Playstation 2® conectado a los pines del Arduino Mega. Se hizo un programa que recibe la información del mando y la convierte en actuaciones en los diferentes elementos del robot. Se proporciona este programa en forma de anexo, cuyos comentarios se hicieron en inglés con idea de que se pueda entender de forma universal.

3 SOFTWARE RELATIVO AL SUMMIT XL

En este capítulo se abordarán el sistema operativo y herramientas utilizadas con el rover terrestre. El elemento principal es el sistema operativo ROS, que funciona bajo Ubuntu 14.04 LTS. Se hablará brevemente del funcionamiento del sistema y la organización de archivos, a fin de ilustrar cómo trabaja, ya que éste no es nada convencional y guarda muchas diferencias con la mayor parte de programas software que se suelen utilizar habitualmente.

Se hablará también de las herramientas de visualización RViz y de simulación Gazebo.

3.1 Sistema operativo: Ubuntu

Ubuntu es un sistema operativo de software libre que se basa en Linux. ROS sólo funciona bajo este sistema operativo. En particular se elige la versión 14.04 LTS por ser una versión desarrollada y estable, es decir, carente de posibles errores que puedan dificultar el desarrollo de esta actividad.

La característica del software libre es que se distribuye de forma gratuita y que suele estar mantenido, actualizado y respaldado por una comunidad de personas, que también suelen proporcionar ayuda en forma de tutoriales, foros y comparten su trabajo para poder respaldar otros proyectos.

3.2 Plataforma de desarrollo: ROS

ROS es un framework de desarrollo de software para robots, originalmente desarrollado en Willow Garage, Laboratorio de Inteligencia Artificial de Stanford. La traducción a castellano de ROS sería Sistema Operativo Robótico, y ciertamente lo es, contando con sus comandos, sistema de archivos con jerarquía propia, subprogramas, librerías y todo lo que cabe esperar de un SO.

La justificación del uso del mismo es que es una herramienta muy potente. Es compatible con gran cantidad de robots y modelos de actuadores y sensores, y tiene una gran comunidad de desarrolladores detrás que proporcionan paquetes para los mismos, de forma que para hacer un proyecto como este no hay que empezar desde el cero absoluto, sino que hay personas que dan una base para poder trabajar en la aplicación específica que buscamos.

Por otro lado la dificultad de ROS reside en que carece de interfaz gráfica, y tiene un funcionamiento difícil de comprender para alguien que no esté habituado a trabajar con sistemas operativos. El sistema de archivos tiene una jerarquía especial a entender, y modificar un simple parámetro se puede hacer un mundo para quien está aprendiendo. Para ayudar a superar esta curva de aprendizaje está la página web wiki.ros.org que contiene tutoriales para aprender ROS desde cero hasta un nivel avanzado y otra

página answers.ros.org donde formular preguntas y obtener respuestas de otros usuarios de la plataforma.

3.2.1 Estructura básica de ros

La forma más sencilla de entender ROS es mediante el uso de grafos. En un grafo encontramos varios nodos, que son programas o procesos independientes que se ejecutan de forma paralela. Estos programas cuentan con sockets TCP, que les permiten comunicarse entre ellos, a través de los topics, que son "temas de conversación", como pueden ser las lecturas de un sensor o la transformada de unas coordenadas a otras.

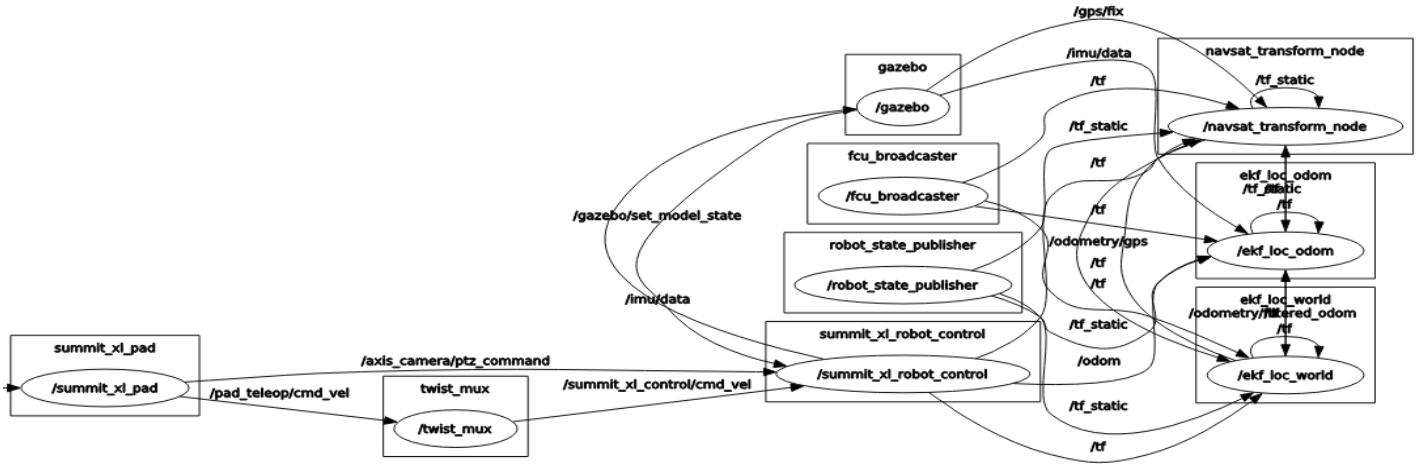


Ilustración 4: Ejemplo de nodos y topics

En estos topics hay un nodo que publica información y otros nodos que se subscriben, porque les interesa contar con esa información para su propio proceso. También es posible comunicarse mediante servicios, donde un nodo puede pedir a otro información en un momento determinado, no de forma constante como en un proceso de subscripción.

3.2.2 ROS como comunidad

El universo ROS se compone de repositorios, que contienen stacks, que a su vez contienen paquetes. A continuación vamos a describir qué es cada uno.

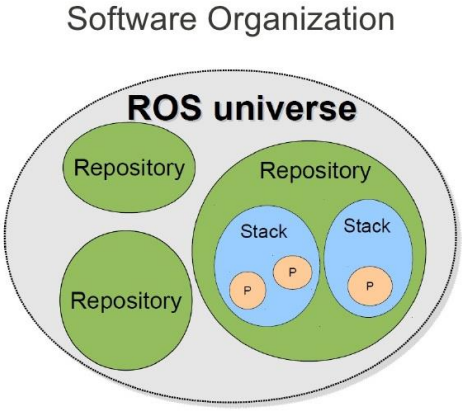


Ilustración 5: Jerarquía del universo ROS

- **Repositorio**: Lugar donde se recopila una serie de pilas relacionadas entre sí, de forma que sea posible para los usuarios acceder a ellas y descargarlas para su uso. Github es el lugar más habitual donde encontrar stacks de ROS.
- **Stack** o pila: Contienen paquetes bajo una temática concreta. Por ejemplo, el Summit XL tiene su propio stack, publicado por la empresa Robotnik, con paquetes para simulación y programación del robot físico.
- **Paquete**: Contiene los programas ejecutables y lanzadores preparados (archivos .launch) que permiten la inicialización de nodos. Como ejemplo, el Summit XL trae el paquete de navegación, que es el más importante de este trabajo, y dentro del mismo encontramos el lanzador move_base_nomap.launch que nos permite lanzar el nodo de navegación autónoma. Los paquetes pueden depender de otros paquetes para funcionar, que es lo que se llaman dependencias.

3.2.3 ROS a nivel software

Ahora que sabemos qué son los paquetes vamos a analizar lo que hay en su interior. Los elementos principales que encontramos son:

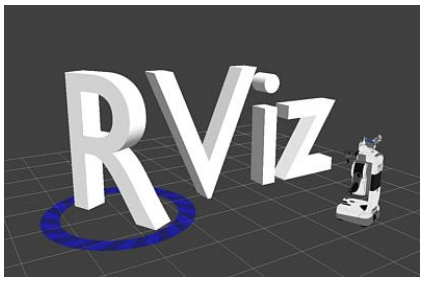
- **CMakeFile**: Es el archivo que se encarga de compilar el paquete. En él están los códigos que el paquete utiliza, librerías y las dependencias del mismo.
- **Package.xml**: Contiene las dependencias del paquete, y se definen los que serán construidos, mediante <*build_depend/*> , y ejecutados, mediante <*run_depend/*>.
- **Src**: es una carpeta que contiene el código que se ejecuta en este nodo.
- **Launch**: es una carpeta que contiene los lanzadores *.launch* que permiten iniciar nodos en conjunto de una sola vez.

3.2.4 ROS a nivel shell

Se llama **Shell** a la ventana de comandos o terminal que sirve de interfaz para utilizar ROS. Hay que conocer y familiarizarse con una serie de comandos para moverse con soltura por el sistema y adoptar un flujo de trabajo rápido y eficiente:

- **roscore**: ejecuta el nodo maestro, sin el cual no funciona el sistema operativo, y del que dependen el resto de nodos, dado que éste se encarga de la comunicación entre ellos. Muchos lanzadores contienen su propio roscore.
- **roscd**: permite cambiar rápidamente de directorio
- **rostopic**: nos permite trabajar con los topics anteriormente explicados. Va seguido de una opción, como puede ser *echo* para ver por pantalla qué publica un nodo, o *list*, para ver una lista de los topics activos.
- **rosrun**: permite lanzar un nodo aislado. Uno muy útil es *rqt_graph*, que dibuja un grafo con el estado actual del sistema, con todos los nodos y topics.
- **roslaunch**: permite el uso de lanzadores *.launch* anteriormente explicados.
- **rosbuild**: construye un paquete, dejándolo preparado para su ejecución.
- **export:** permite definir que un nodo que se está ejecutando en otra máquina.
- **ssh:** establece una conexión ssh con otra máquina.
- **sudo:** da permisos de administrador para editar archivos del sistema que los requieran.

3.3 Herramienta de visualización: Rviz



Rviz es una herramienta que suele utilizarse en conjunto con ROS para poder ver en tiempo real la información que se publica en los diferentes topics. Esta puede ser, por ejemplo, la posición del robot, la lectura del sensor láser, o el mapa que se está generando.

Podemos ir añadiendo herramientas visuales de varios tipos: mapa, punto, conjunto de vectores, descripción de robot… y cada uno de estos tipos se subscribe a un topic compatible, cuya información se muestra en la pantalla.

3.4 Herramienta de simulación: Gazebo

A pesar de tener la fortuna de contar con el robot real para probar los programas, es útil tener una herramienta que nos permita trabajar cuando no estamos en el laboratorio, que lo simule en un entorno que imite a la realidad. Gazebo es la herramienta que suple esta necesidad.

Esta herramienta tiene en cuenta parámetros físicos del robot, sus restricciones holonómicas, y permite generar entornos de interiores y exteriores, proporcionando también algunos de ejemplo.

3.5 Ejemplo de simulación completa

Para poder simular el robot en Gazebo debemos ejecutar la siguiente instrucción:

roslaunch summit_xl_sim_bringup summit_xl_complete.launch

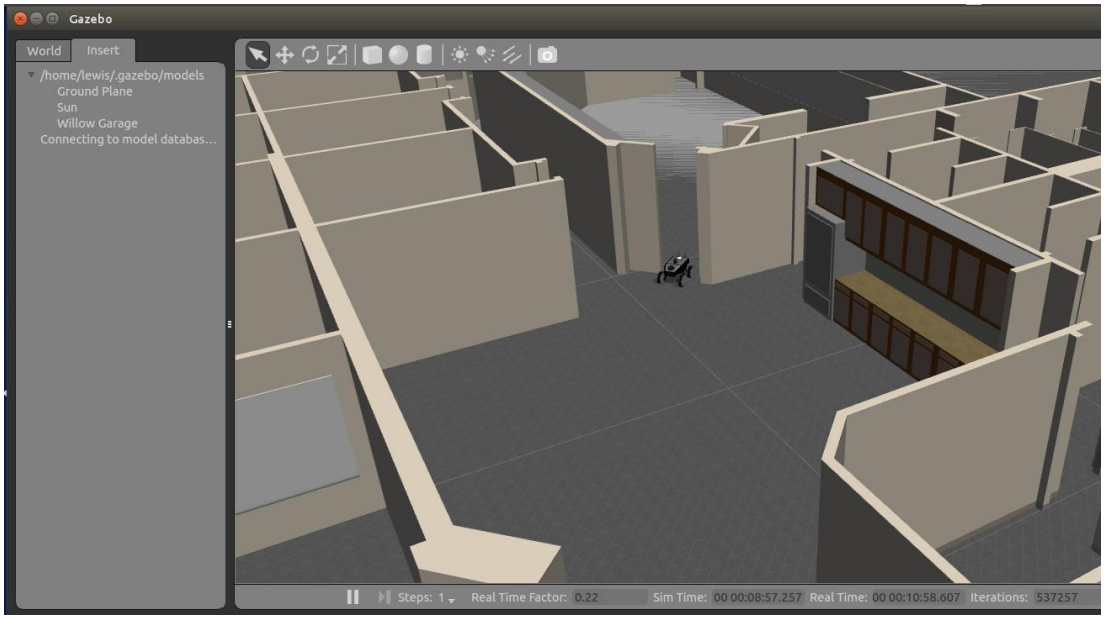


Ilustración 6: Gazebo simulando el Summit en Willow Garage

Esta orden iniciará de forma automática una simulación en Gazebo del robot, en un espacio vacío. Podemos utilizar uno de los mapas que vienen por defecto (como el de Willow Garage) para tener un entorno con obstáculos, o hacer uno nosotros.

Una vez abierta esta simulación podemos utilizar la herramienta Rviz para visualizar lo que esté sucediendo en nuestra simulación:

rosrun rviz rviz

Esta orden abrirá el programa vacío, que tendremos que configurar para poder ver aquellos parámetros que nos sean de interés. Utilizando la herramienta add podemos anadir un tipo de visualización y subscribirla al topic que nos sea de interés visualizar.

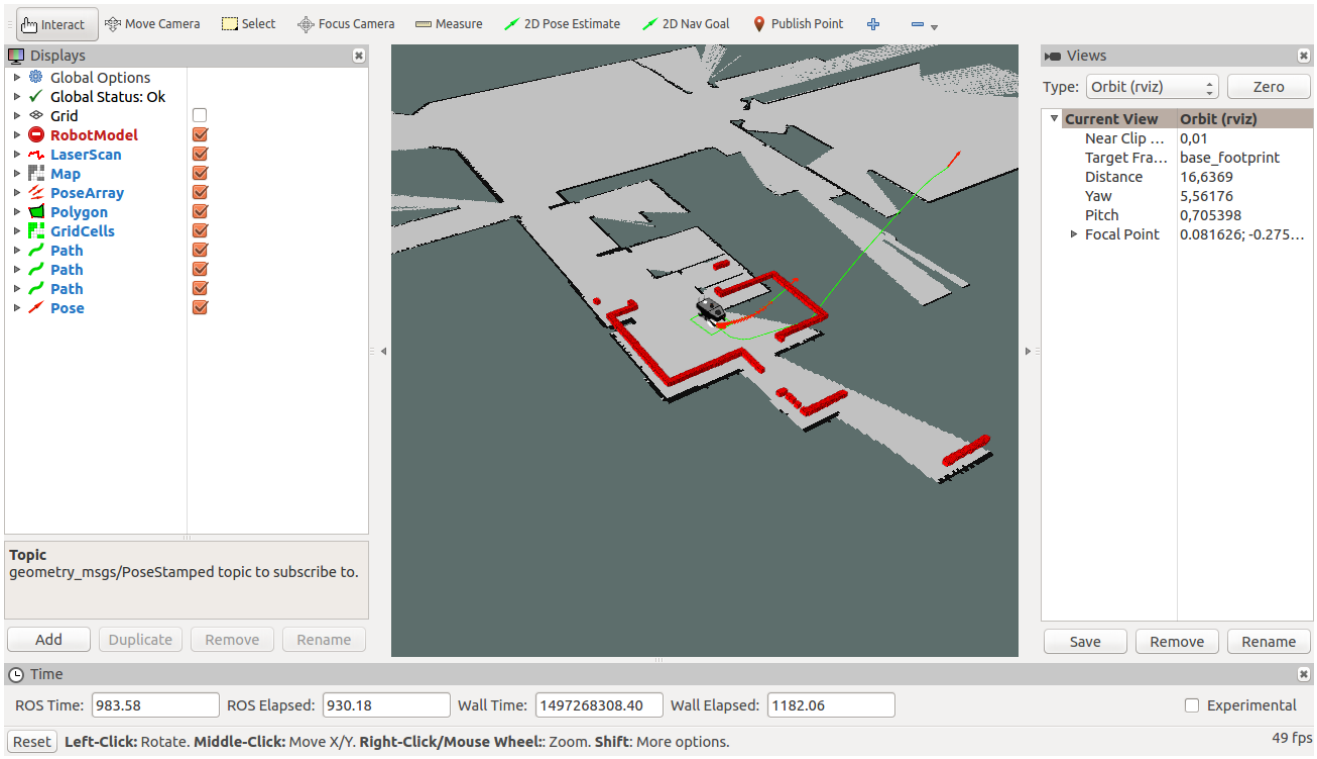


Ilustración 7: Ejemplo de visualización en Rviz

4 COMUNICACIÓN CON EL ROVER

En este capítulo se hablará de los pasos que se han tomado para poder trabajar con el Summit XL desde el inicio, no sólo con el objetivo de ilustrar la labor realizada en su totalidad, sino también teniendo en mente guiar a aquellos que tomen el relevo en este proyecto, dando información, explicaciones y consejos importantes que no figuran en la documentación del robot.

4.1 Descripción rápida del robot

El Summit XL del que se dispone en este proyecto es un modelo que cuenta con:

- Sensor laser SICK TIM551.
- Ruedas en configuración diferencial.
- GPS U-BLOX EVK-7P
- Cámara PTZ AXIS 5514.
- Conexión mediante Wifi 2.4Ghz con doble antena
- Conectividad Bluetooth para el mando inalámbrico

Ilustración 8: Summit XL del INTA

4.2 Conexión del PC al Summit

Para poder trabajar con el robot primero tenemos que conectarlo con el ordenador donde vayamos a trabajar con él. Es importante detallar que estamos conectando dos PC's entre sí, el nuestro y el interno del Summit, ambos corriendo Ubuntu LTS 14.04. Con este fin, tras encender el Summit y dar tiempo suficiente a la CPU a iniciarse, podemos establecer una conexión entre ambos.

Para esto, el Summit emite una red Wifi propia, a la que debemos conectarnos, utilizando la contraseña especificada en el manual. Una vez establecida la conexión podemos comenzar a configurar el robot.

Abrimos un nuevo terminal (Ctrl+Alt+T) y tecleamos la siguiente orden:

export ROS_MASTER_URI=http://summit:11311

Con esta orden estamos indicando al terminal actual que el proceso principal de ros (roscore) se está ejecutando en la máquina remota (Summit) y no en la local (localhost) por el puerto 11311. A continuación escribimos:

ssh summit@summit

Ssh es un comando Shell que establece una conexión útil para acceder de forma segura y remota a servidores privados. Esto nos permite acceder al ordenador de a bordo remotamente desde el terminal que tenemos abierto, como si estuviésemos abriendo un terminal en este.

Ahora el siguiente paso es encontrar el archivo *hosts.txt*, en el que vamos a introducir el nombre de nuestro PC junto con nuestra dirección IP.

Para esto hacemos:

cd ~/etc/hosts

Y entramos en el archivo de configuración:

sudo vim hosts.txt

A continuación tendremos que introducir nuestro nombre y dirección IP. Es fundamental para evitar problemas conocer cómo funciona el editor Vim, leernos el manual y saber qué vamos a hacer antes de editar el archivo, ya que un fallo al editarlo puede suponer una pérdida importante de tiempo recuperándolo a posteriori. En un terminal nuevo accedemos al manual:

man vim

Sabiendo manejar el editor podemos proceder a editar el documento. Escribimos nuestro nombre e IP y asegurándonos de que es correcto, guardamos el archivo con la secuencia:

:wq

Siendo "*:*" el inicio del escritor de comandos, "*w*" para escribir el archivo y "*q*" para cerrar el editor. Si todo ha salido bien deberíamos poder ver los topics que el robot está publicando desde nuestro ordenador:

rostopic list

4.3 El servidor X

Algo más que no se menciona en la documentación del Summit es la existencia del servidor X. Este es un proceso que permite utilizar el ordenador de a bordo del Summit visualizándolo desde el nuestro. Esto es, si ejecutamos un proceso (como puede ser una instancia de Rviz) en el ordenador del Summit, iniciando un ssh con servidor X podemos ver la ventana gráfica que se abre en el Summit en el ordenador remoto e interactuar con ella.

Esto consume mucho ancho de banda en la comunicación entre ambos dispositivos, pero puede ser de utilidad en casos puntuales. La forma de iniciar el servidor X es:

ssh –X summit@summit

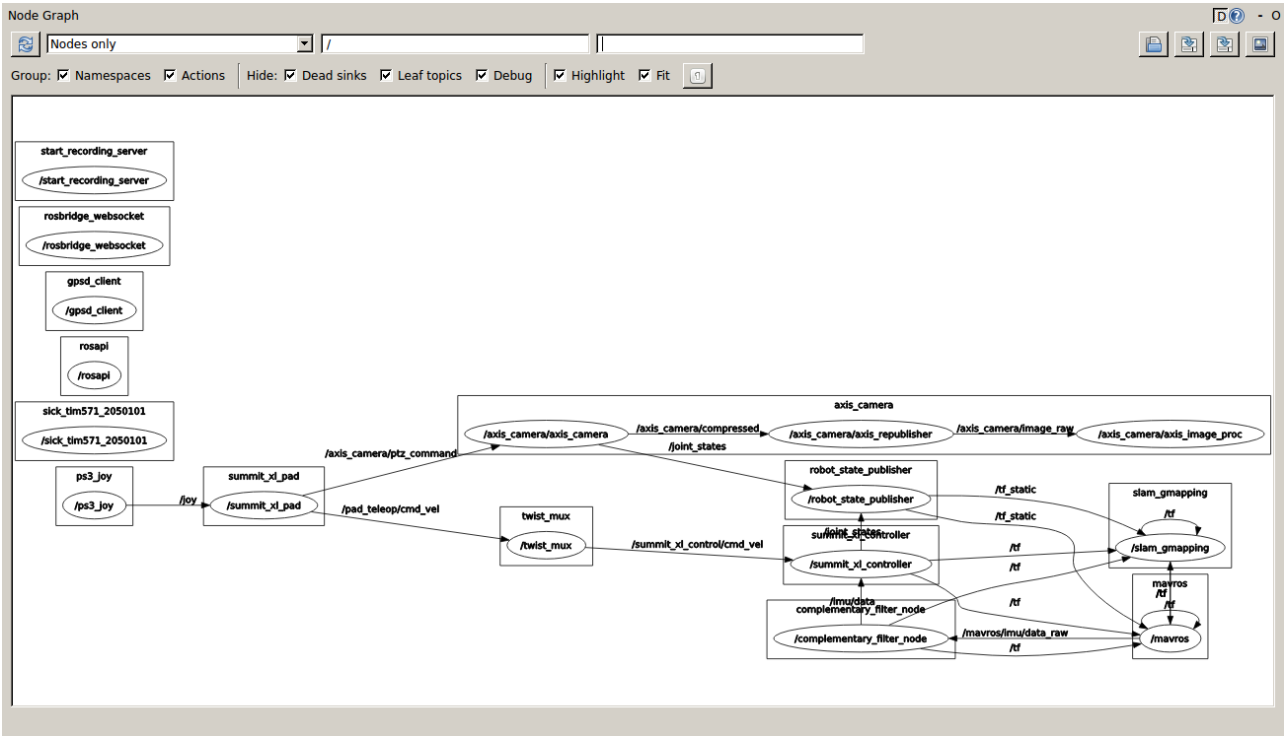


Ilustración 9: Rosgraph ejecutado en Summit visto desde servidor X

4.4 Visualización en Rviz

Una vez que tenemos el robot mandando información de los topics constantemente podemos visualizarla mediante la herramienta anteriormente mencionada Rviz. Para esto escribimos en un nuevo terminal:

export ROS_MASTER_URI=http://summit:11311

Podríamos poner esta orden para que se iniciase cada vez que abrimos un terminal, pero eso nos impediría trabajar en local cuando no estemos con el robot conectados. Para abrir Rviz ponemos a continuación:

rosrun rviz rviz

Esto abre la herramienta sin configuración ninguna. Ahora tenemos que añadir topics con los sensores que queramos visualizar. Pulsamos el botón add en la barra izquierda y empezamos a añadir los siguientes:

- Robot model: contiene la descripción del robot
- Laser_sensor: Nos da la lectura del laser en tiempo real
- Image: Proporciona la imagen en tiempo real de la cámara
- TF: Dibuja la ubicación física de los diferentes marcos de referencia entre los cuales se realizan las transformadas.

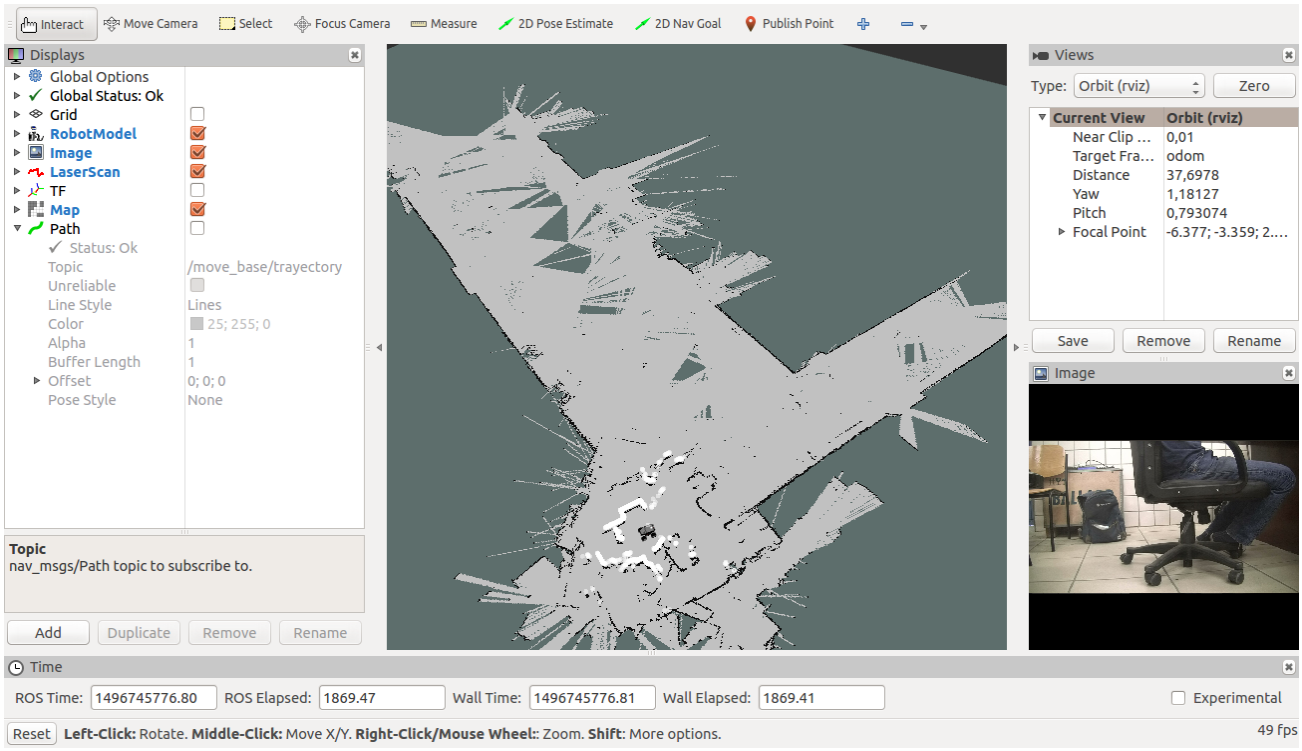


Ilustración 10: Ejemplo de mapa real en Rviz

5 MAPEADO Y NAVEGACIÓN AUTÓNOMA

Para tareas de exploración en terreno hostil, o para realizar sistemas de navegación basados en mapa como puede ser AMCL, el reconocimiento del terreno por parte del rover es una parte fundamental de la tarea. Además, en este capítulo se tratará el fin último de este trabajo con ROS: el sistema de navegación autónoma para el Summit.

Se explicará en primer lugar cómo se puede hacer un mapa moviendo el robot por un espacio y visualizándolo en tiempo real, y en segundo lugar se desarrollará el sistema de navegación implementado en el robot.

5.1 SLAM (simultaneous localization and mapping)

El SLAM, en castellano localización y mapeado simultáneo, es un sistema que se centra en resolver el problema de hacer un mapa en un territorio desconocido para un robot móvil. Este sistema realiza un proceso basado en sistemas con espacio de estados.

En primer lugar extrae Landmarks o puntos característicos del terreno (paredes, esquinas…), luego estima el estado, lo actualiza, y actualiza las landmarks. La práctica más extendida es utilizar como medidas tanto la odometría del robot (que sufre de error acumulativo por deriva) como la medida de los sensores (que pueden o no ser precisos), y combinar estas mediante un filtro de Kalman extendido.

5.2 Mapeado utilizando gmapping

Una vez hemos conectado el robot al pc y tenemos Rviz preparado para funcionar, el siguiente paso es lanzar un nodo de slam_gmapping. Gmapping es un paquete que corre un proceso basado en SLAM para hacer un mapa haciendo uso de las medidas del sensor láser y posición del robot. Este proceso publica en el topic */map* de tipo mapa.

Para iniciar el nodo tenemos que lanzarlo en el Summit, no en el ordenador local, para que el proceso se quede funcionando aunque el ordenador pierda la conexión wifi, y podamos llevar el rover a explorar un territorio (dejando nuestro pc atrás) y tener el mapa a la vuelta. Ejecutamos:

export ROS_MASTER_URI=http://summit:11311

ssh summit@summit

rosrun gmapping slam_gmapping

Esto iniciará un proceso que toma las lecturas del topic "/scan" y las de la transformada "/tf" para generar un mapa. Podemos visualizarlo en Rviz añadiendo una visualización de tipo mapa y subscribiéndola al topic "/map". De momento, podemos mover el Summit por la zona de interés de forma manual utilizando el mando inalámbrico.

En caso de querer guardar el mapa hecho, podemos hacerlo utilizando la herramienta map_server:

rosrun map_server map_saver <mapa>

Esta instrucción guardará el mapa bajo el nombre que demos en <mapa> con la extensión *.yaml.* Si abrimos Rviz y queremos ver ese mapa tenemos que ejecutar el map server, que lo que hace es publicar en el topic */map* el mapa guardado, al igual que anteriormente teníamos en este topic el mapa que se estaba generando con *slam_gmapping*. La instrucción es:

rosrun map_server map_server <mapa.yaml>

5.3 Sistemas de navegación sin mapa

El sistema de navegación, al no poder contar con un mapa, se basará sólo en la información de posición del robot y las lecturas del láser para moverse por el entorno. Para esto utilizará un sistema de mapas de coste, que básicamente se compone de los obstáculos que el láser detecta, inflados un determinado radio, para tener en cuenta las dimensiones del robot y evitar colisiones.

5.3.1 Zonas de coste

Estos mapas se dividen en cuadrículas, y a cada una de estas se asignan valores (que se muestran en colores en Rviz), calificando cada cuadrícula como espacio libre, circunscrito, inscrito, letal o desconocido. Se establecen valores de coste entre 0 y 254 según el riesgo de colisión.

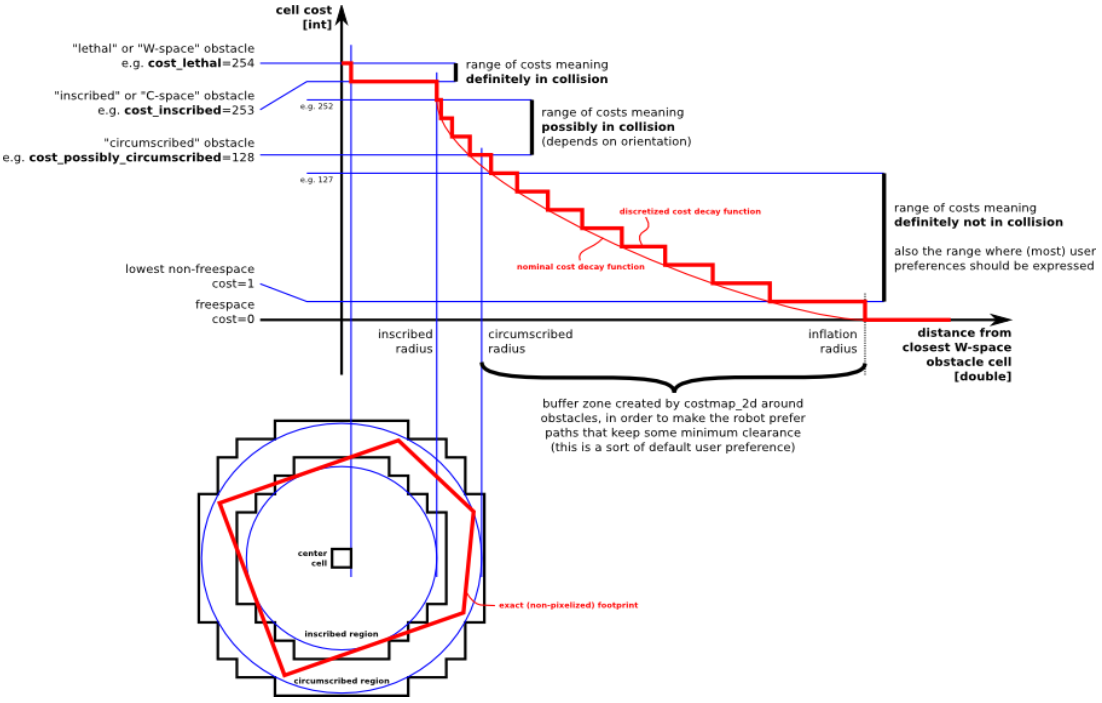


Ilustración 11: Relación distancia-coste en costmaps

5.3.2 Estructura del sistema de navegación

Para ir a una meta tenemos que elegir un camino. Ahí entran en juego los planner global y local, que sirven para determinar el camino teniendo en cuenta el costmap. El local utiliza una cuadrícula pequeña de información a su alrededor, y sirve para decidir el camino "a corto plazo", considerando un costmap más detallado. El planner global considera todos los obstáculos conocidos y decide cuál será la dirección a tomar por el planificador local. Ambos se nutren de la información de sensores para insertar información de ocupación (mark) o eliminarla (clear).

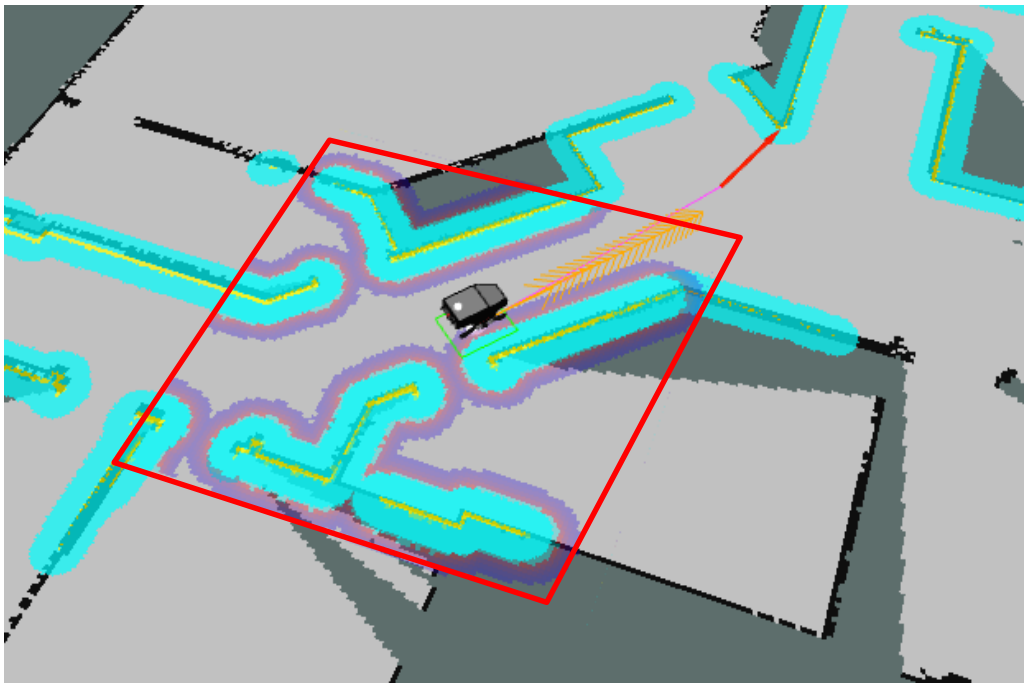


Ilustración 12: Planners en Rviz

En la ilustración 12 podemos ver los mapas de coste de los planners en Rviz. El planner local está señalado con una cuadrícula roja que marca su extensión, y es fácilmente distinguible por el grado de detalle del costmap, que tiene 4 colores, correspondientes con los 4 niveles de coste que se pueden asignar a una cuadrícula del mapa. El global sólo tiene dos colores, almacena información de donde el robot ha visto obstáculos, y tiene una extensión equivalente a la cantidad de mapa que el robot haya visitado. También se pueden ver las rutas del planner local (vector de flechas de color naranja) y del global (línea morada), que sale de la cuadrícula y llega hasta la meta (flecha roja).

Como hemos visto, cada uno de estos planificadores tiene su propio mapa de coste, que a su vez están conectados a los los comportamientos de recuperación, de los que se hablará más adelante. En la ilustración 13 se pueden ver dos bloques opcionales que son amcl, un sistema de navegación conocido el mapa, y map server, que es un nodo que proporciona el mapa al sistema. Su adición es posible, pero no es el objetivo de este trabajo.

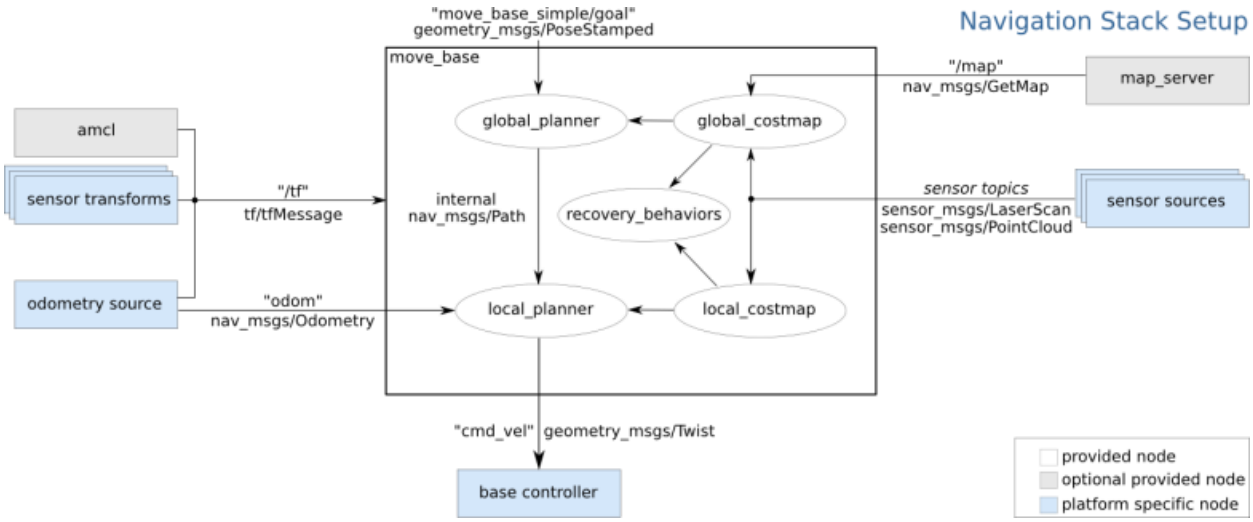
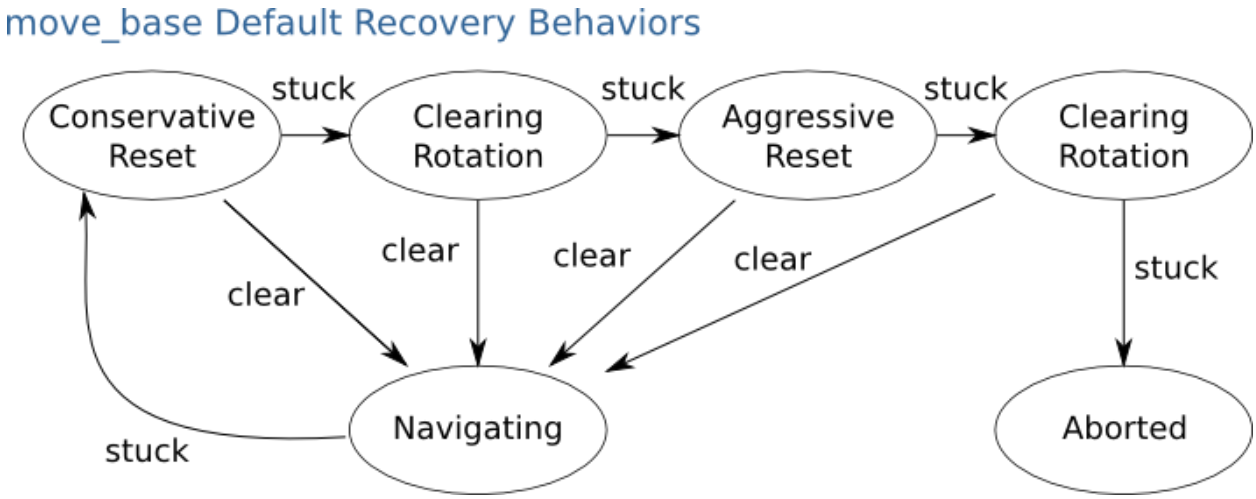


Ilustración 13: Estructura del stack de navegación

5.3.3 Comportamientos de recuperación

Cuando el robot no encuentra camino posible hasta el objetivo, intenta hacer una rutina antes de dar la meta por inalcanzable. Esta se compone de los comportamientos de recuperación por defecto.



En primer lugar, los obstáculos fuera de una región que se especifica en los parámetros del costmap se eliminan del mapa (acción Clear_Costmap_Recovery). A continuación el robot ejecuta una rotación sobre sí mismo para asegurarse de tener espacio para moverse (acción Rotate_Recovery).

Si esto también falla, el robot eliminará de forma agresiva todos los obstáculos que estén fuera del espacio sobre el que gira sobre sí mismo, terminando con otra rotación sobre sí mismo. Si finalmente no hay éxito el robot considerará el objetivo incalcanzable y nos nostificará por terminal de que le es imposible llegar.

Todo esto es configurable mediante un parámetro del nodo move_base que se llama *recovery_behaviours*, donde hay una lista con estos comportamientos.

5.4 Navegación autónoma sin mapa

Para realizar la navegación autónoma se utilizará el paquete de navegación del Summit. Para utilizar el sistema de navegación sin mapa se abre lanzando el paquete *move_base_nomap.launch*. Este paquete de navegación ejecuta la rutina de movimiento basado en mapas de coste.

Abriendo el lanzador de nodo encontramos los archivos que se lanzan al ejecutarlo. Estos archivos son archivos tipo *.yaml* que se encuentran en la carpeta config, y definen una serie de parámetros genéricos y específicos para los planificadores.

Al ejecutar el nodo de navegación el comportamiento del robot no era el esperado. El robot aceptaba metas de navegación, y los planificadores hacían su trabajo y daban una ruta que llevaba al robot al objetivo. Sin embargo, si había algún obstáculo en el camino el robot no lo tenía en consideración, y colisionaría con él, ya fuese un obstáculo móvil (como una persona andando) o fijo (como una pared). Por otra parte, en Rviz no se veían los mapas de coste.

Esto hace sospechar que algo no está funcionando en el nodo move_base, dado que al probarlo en simulación sí evita obstáculos y los mapas de coste se pueden ver en Rviz. Dando un vistazo al grafo podemos ver que el nodo */scan* no está comunicando con ningún elemento de move_base.

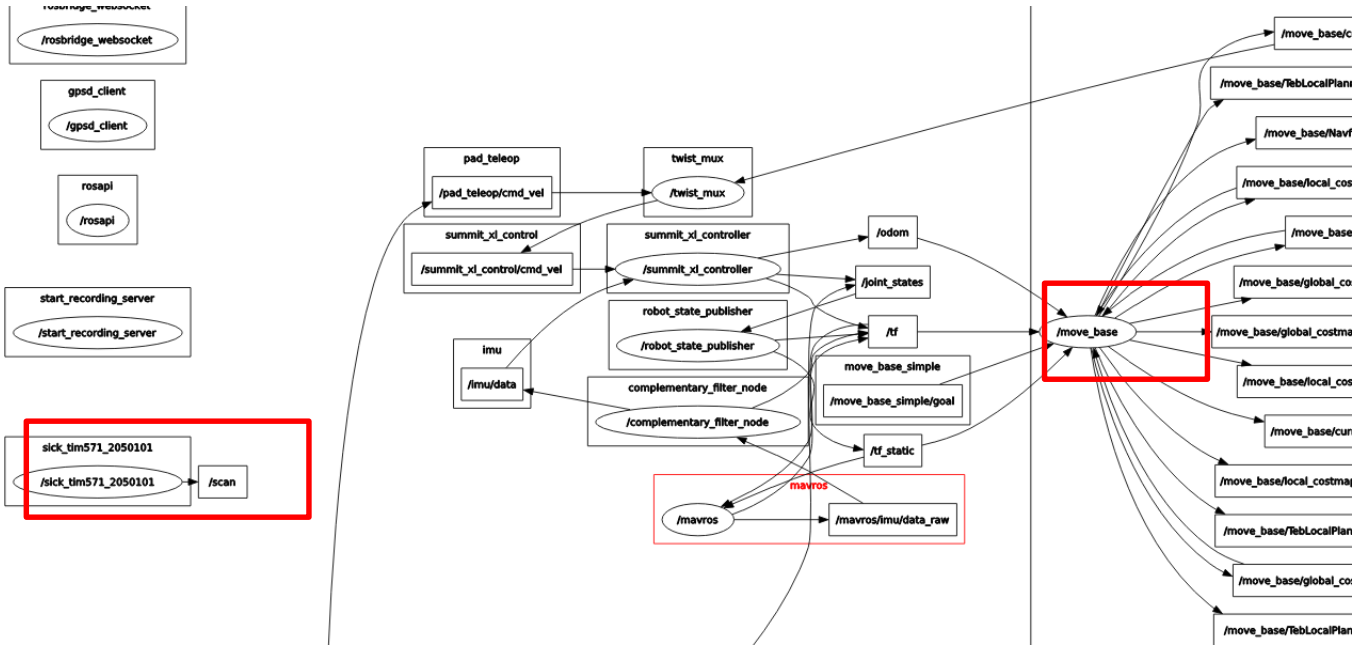


Ilustración 14: La lectura del láser no llega a move_base

Por otra parte, el terminal donde estamos ejecutando este nodo nos aconseja abrir el logger en modo debug, para recibir mensajes de error. Al hacerlo se puede entrever el problema. Move_base se está subscribiendo a un topic llamado *hokuyo_base/laser*, y nuestro sensor laser no es un hokuyo, es un sick.

Dicho de otra forma, el problema es que el sick está publicando información por una parte, y por otra move_base está solicitando datos de un topic que no publica, llamado *hokuyo_base/laser*.

Por tanto la solución al problema pasa por hacer que el nodo de navegación reconozca la lectura del sensor láser de forma correcta, leyendo la que el sick publica. Para corregir este problema tenemos que editar el archivo *.yaml* que hay en el paquete de navegación del ordenador del summit, ya que es ahí donde se ejecuta el nodo. Para esto hacemos en un terminal nuevo:

export ROS_MASTER_URI=http://summit:11311

ssh summit@summit

cd ~/summit_xl_common/summit_xl_navigation/config

sudo vim costmap_common_params.yaml

Esto nos muestra el interior del paquete, donde se declara el sensor láser. Para solucionar el problema tenemos que corregir la información para incluir el sick. La que viene por defecto es:

```
  obstacle_layer:
                      observation_sources: hokuyo_laser
                      hokuyo_laser: {sensor_frame: hokuyo_base_laser_link, data_type:
                    LaserScan, topic: hokyo_base/scan, marking:     true, clearing: true}
```

Tras probar varias combinaciones, la mejor forma de hacerlo es conservar la transformada del hokuyo_base, porque el lugar a donde transforma las coordenadas (véase, el lugar donde físicamente iría el hokuyo) es el mismo que donde está el sick. Cambiamos el nodo donde se hace la lectura a */scan*, que es donde sale la información del sick, y cambiamos el nombre para que a partir de ahora se haga referencia a él como "sick".

```
obstacle_range: 2.5
raytrace_range: 3.0

footprint: [[0.35, -0.3], [0.35, 0.3], [-0.35,0.3], [-0.35, -0.3]]

publish_frequency: 1.0

inflation_layer:
  inflation_radius: 0.5

obstacle_layer:
  observation_sources: sick
  sick: {sensor_frame: hokuyo_base_laser_link, data_type: LaserScan, topic: scan, marking: true, clearing: true}
```

Ilustración 15: Edición remota de la configuración de costmap_common_params.yaml

Tras este ajuste guardamos el archivo y nuevamente lanzamos el nodo:

ssh summit@summit

cd ~/summit_xl_common/summit_xl_navigation/config

roslaunch summit_xl_navigation move_base_nomap.launch

Comprobamos usando rosgraph que ahora efectivamente la lectura del láser llega a move_base:

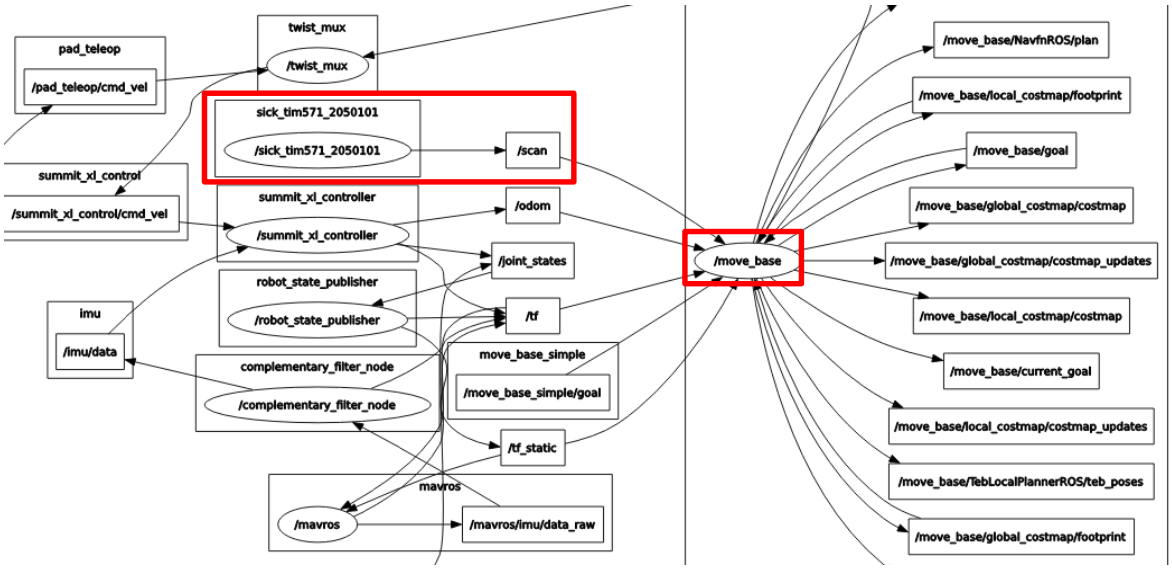


Ilustración 16: La lectura llega a move_base

Ahora la navegación muestra los costmaps y el robot navega hacia los objetivos sin colisionar y esquivando obstáculos. En Rviz tenemos que añadir:

- Map: suscrito a */map* para ver el mapa que estamos creando en tiempo real.
- Path: Suscribiendonos al local planner y al global planner (flechas naranjas).
- Map: Marcando como tipo Costmap, para ver lo que ven el global planner y el local planner (mapas de coste en rango de color amarillo hasta azul).
- 2D Nav Goal: marca el lugar objetivo al que se quiere navegar, incluyendo la orientación.

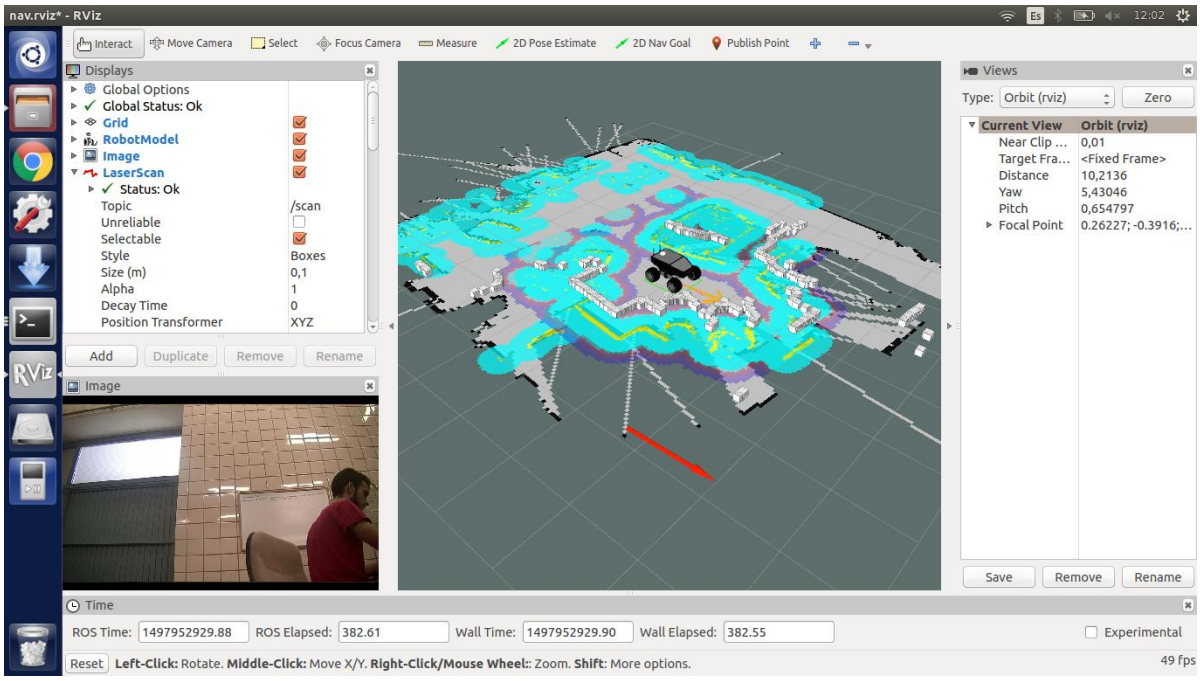


Ilustración 17: Navegación del robot real con mapas de coste

6 RESULTADOS Y CONCLUSIONES

Tras lograr hacer funcionar el sistema de navegación autónoma y mapeado, se hicieron varias pruebas, tanto en un entorno de interiores como en exteriores, y con obstáculos fijos y móviles. Esto fue revelador respecto a las bondades y las deficiencias del sistema, que se describirán a continuación.

6.1 Resultados

6.1.1 Navegación en interiores

En entornos de interiores el mapeado es rápido y preciso, y el sensor láser tiene un alcance real de unos 12 metros, por lo que si la sala es pequeña basta con moverlo entre los obstáculos para disponer del mapa. Cabe destacar que este puede tener alguna lectura errónea por ruido del sensor, que produce de forma perceptible zonas donde el mapa se extiende hasta donde no debería.

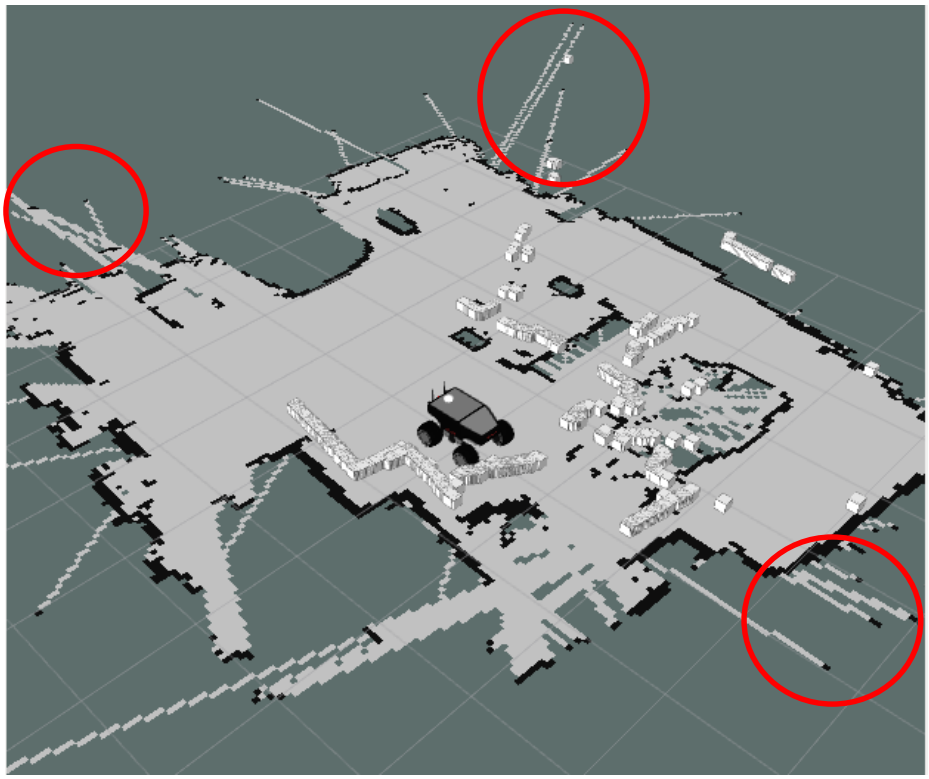


Ilustración 18: Mapa con lecturas espúreas

6.1.2 Navegación en exteriores

En exteriores, dado que el alcance del laser está restringido, el mapa tiene que hacerse barriendo completamente el territorio para evitar que queden espacios donde no se conoce la información de ocupación.

En caso de tener un mapa desconocido (que se esté haciendo en ese momento) resulta especialmente difícil predecir dónde se está poniendo la meta de navegación, porque estamos ubicando un punto en un espacio vacío.

Ilustración 19: Mapa de exteriores con zonas no cubiertas

6.1.3 Evitación de obstáculos fijos

El sistema de navegación supone un método de respuesta ante obstáculos que se ejecuta en tiempo real, utilizando las lecturas que el sensor recibe. Es importante discernir entre obstáculos fijos (están ahí permanentemente) y móviles (están en una ubicación pero desaparecerán). El sistema tiene esto en cuenta a la hora de hacer mapas de coste, y va realimentando los obstáculos que cree tener con las lecturas del láser, eliminando los que son móviles.

Una vez que el robot recibe las lecturas del láser hace los mapas de coste y los utiliza para calcular una ruta. Su comportamiento es lógico, aumenta la velocidad cuando está lejos de los obstáculos y la reduce al aproximarse. Por otra parte intenta circular siempre en zonas donde no haya peligro ninguno de colisión, es decir, intenta que el centro del robot no toque el mapa de coste de ningún obstáculo. En caso de que los mapas de coste de dos obstáculos solapen (ver ilustración 20), el robot navegará manteniendo su eje de simetría en la zona de coste más bajo (menor riesgo de colisión).

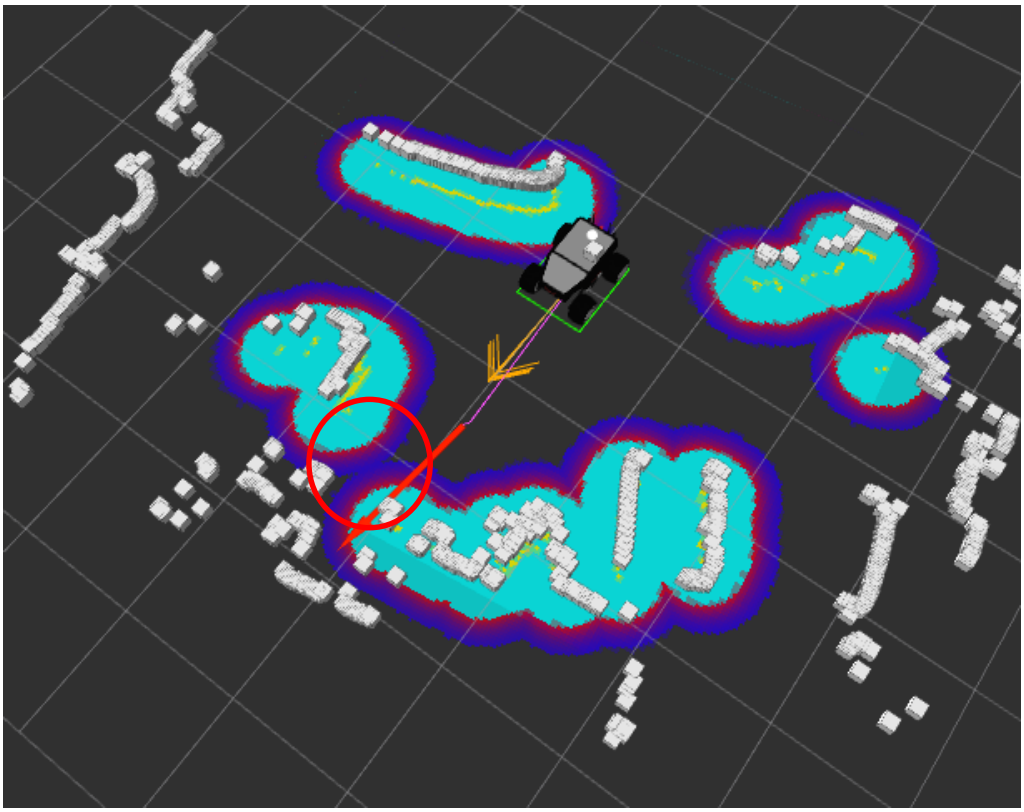


Ilustración 20: Navegación con mapas de coste solapados

Por otra parte si pedimos al robot que llegue a un sitio que es inalcanzable o está en el mapa de coste de un obstáculo, se aproximará al máximo posible, pero no colisiona en casi ninguna ocasión (ver ubicación del láser), y da por consola un mensaje que indica que el objetivo solicitado es inalcanzable.

6.1.4 Evitación de obstáculos móviles

Cuando ponemos un obstáculo móvil en la trayectoria del robot, éste lo esquiva segun se aproxime, es decir, recalcula su ruta teniendo en cuenta que hay un obstáculo nuevo. En el peor caso (el obstáculo está justo en frente) llega al punto de casi pararse, pero a continuación lo rodea lentamente.

El robot ha navegado autónomamente por la universidad por zonas con personas andando, en algunos casos muy cerca del robot, y en ninguna prueba de navegación ha colisionado con ninguna.

6.2 Conclusiones y mejoras propuestas

6.2.1 Relativo al mapeado

De las pruebas se extrae que siempre es mejor intentar evitar hacer giros con curvas cerradas, ya que inducen errores en la odometría y pueden provocar que el mapa y las lecturas del láser se descoordinen. Especialmente se hace evidente en espacios pequeños (interiores) dado que es más frecuente hacer giros significativos en lugares donde el espacio de maniobra es reducido.

Por otra parte, en caso de estar en un espacio amplio conviene hacer un barrido inicial usando control manual, o en su defecto poner metas de navegación próximas al robot. Con esto se pretente ir barriendo el territorio poco a poco, ya que sino estamos mandando al robot a una posición que no sabemos dónde está realmente.

6.2.2 Relativo a la navegación

Quizá el mayor problema que podría resolverse a continuación es que tiene el robot tiene el sensor láser barriendo un plano a una determinada altura. Esto supone que aunque funcione correctamente el sistema de navegación, el robot podrá chocar contra obstáculos que estén fuera del área de lectura del láser, véase, por debajo del plano que barre.

En la ilustración 21 se marca la zona de barrido del sensor en color amarillo, y la de posible colisión en naranja. Si algún obstáculo es lo suficientemente pequeño como para no sobresalir hasta la línea amarilla, o tiene un saliente que entre en la zona naranja, el robot “no lo verá” y puede chocar con él si está dentro de su ruta.

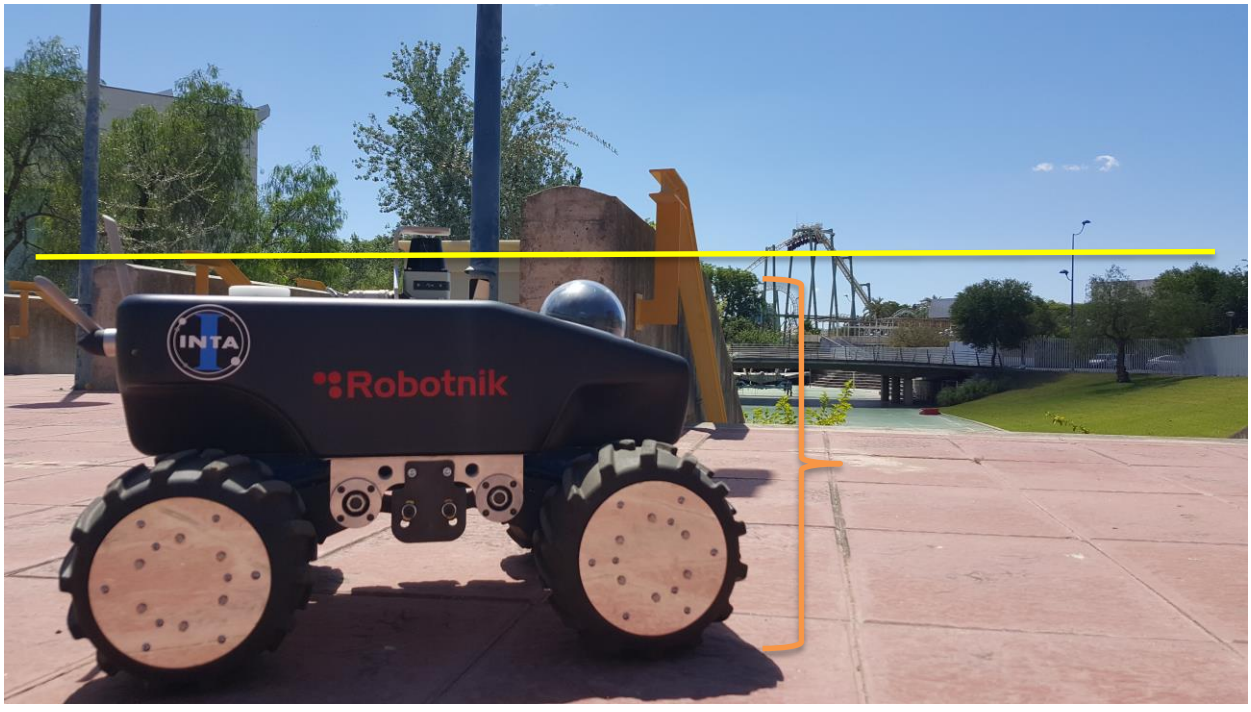


Ilustración 21: Zonas no cubiertas por sensor

Como solución a este problema se podrían instalar unos sensores de ultrasonidos en el frontal, de forma que en cualquier caso si éstos leen un objeto cercano el robot se detenga y se aleje, e intente abordar por otra parte la aproximación al objetivo.

Por otro lado encontramos que según la configuración del robot cada rueda tiene capacidad de moverse independientemente. En el caso de que esté en una zona con baches, huecos en el suelo, superficies resbaladizas… el robot puede intentar moverse con una o dos ruedas. Si estas quedasen flotantes por la forma del terreno, se quedará atascado sin poder moverse del sitio (como sucedió en las pruebas al entrar en una zona de césped).

A modo de solución propongo un sistema que detecte que el robot no se está moviendo, a pesar de estar moviendo las ruedas, ya sea mediante lecturas del láser (no funcionaría en espacios completamente abiertos) o por algún otro sistema de posicionamiento (como puede ser GPS). Al detectar esto, el robot podría entrar en una rutina de recuperación, como puede ser retroceder moviendo todas las ruedas simultáneamente, e intentar llegar a la meta por otra ruta (marcar esa zona como obstáculo en el mapa de coste).

7 REFERENCIAS Y BIBLIOGRAFÍA

[1] Página principal de ROS: http://www.ros.org/

[2] Wiki ROS: http://wiki.ros.org/

[3] ROS answers: http://answers.ros.org/

[4] Anis Koubaa: *Robot Operating System: The Complete Reference,* 2016

[5] Robotnik robotics: http://www.robotnik.es/

[6] Proyecto IUFCV: http://iufcv.com/

[7] GMapping: http://openslam.org/gmapping.html