Universitat de les Illes Balears

escola politècnica superior

Trabajo Fin de Grado

INGENIERÍA ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA

# Herramienta de ayuda a la teleoperación a distancia de robots móviles

MARTA BARQUIEL ROSSELLÓ

**Tutores**

Javier Antich Alberto Ortiz

Escola Politècnica Superior
Universitat de les Illes Balears
Palma, 17 de septiembre de 2018

Me gustaría agradecer a mis tutores Javier Antich y Alberto Ortiz la responsabilidad y la confianza depositada en mi para realizar este proyecto.

# ÍNDICE GENERAL

**Índice general** **iii**

**Índice de figuras** **v**

**Acrónimos** **vii**

**Resumen** **ix**

**1 Introducción** **1**
1.1 Motivación . . . 1
1.2 Objetivo . . . 1
1.3 Punto de partida . . . 2
1.4 Herramientas utilizadas . . . 3
1.4.1 ROS . . . 3
1.4.2 Android Studio . . . 7
1.5 Estructura del documento . . . 7

**2 Generación de un camino al objetivo** **9**
2.1 Comunicación entre la aplicación Android y el nodo *pr2_controller* . . . 9
2.2 Fijar el punto objetivo . . . 10
2.3 Paquete *Gmapping* . . . 11
2.4 Librería SBPL . . . 11
2.4.1 Creación del fichero del entorno . . . 12
2.4.2 Aplicación del algoritmo de planificación . . . 14
2.4.3 Lectura del fichero con la trayectoria generada . . . 16
2.5 Visualización en Rviz del punto objetivo y del camino a recorrer . . . 16

**3 Visualización del láser, del camino recorrido y de los gaps navegables** **19**
3.1 Información proporcionada por el sensor láser . . . 19
3.2 Información proporcionada por el sensor posición . . . 21
3.3 Algoritmo de navegación *Closest Gap* . . . 21
3.3.1 Clasificación de los algoritmos de navegación . . . 21
3.3.2 Introducción al algoritmo . . . 22
3.3.3 Análisis de *gaps* . . . 22
3.4 Visualización en Rviz . . . 25
3.4.1 Escáner láser del robot . . . 25
3.4.2 Trayectoria recorrida por el robot y el modelo del robot PR2 . . . 25

3.4.3 *Gaps* navegables . . . 26
3.4.4 Representación conjunta en Rviz . . . 27

**4 Estructura general del sistema** **29**
4.1 Componentes del sistema y su interconexión . . . 29
4.2 Herramienta de teleoperación propuesta . . . 30
4.3 Detalles del nodo *pr2_controller* . . . 31
4.3.1 Funciones de suscripción: *callbacks* . . . 33
4.3.2 Funciones implementadas directamente vinculadas con el algoritmo *Closest Gap* . . . 33
4.3.3 Funciones implementadas para generar un camino al objetivo . . . 34
4.3.4 Función implementada para la visualización en Rviz . . . 34

**5 Resultados experimentales** **37**
5.1 Sobre *Blender* . . . 37
5.2 Sobre MORSE . . . 37
5.3 Sobre Rviz . . . 38
5.4 Elementos visualizados en Rviz . . . 39
5.4.1 Mapa del entorno, posición y orientación del robot y el punto objetivo . . . 39
5.4.2 Sensor láser . . . 40
5.4.3 Camino recorrido por el robot . . . 40
5.4.4 *Gaps* navegables . . . 41
5.4.5 Ruta hacia el objetivo . . . 41
5.5 Entornos . . . 42
5.5.1 Entornos simples . . . 42
5.5.2 Entornos complejos . . . 42

**6 Conclusiones** **53**
6.1 Fase previa . . . 53
6.2 Implementación . . . 53
6.3 Resultados obtenidos . . . 54
6.4 Problemas encontrados . . . 54
6.4.1 Transformaciones . . . 55
6.4.2 Información proporcionada por el sensor láser . . . 55
6.4.3 Generación del camino al objetivo . . . 55
6.4.4 Documentación en la wikipedia de ROS . . . 56

**A Código *pr2_controller.cpp*** **57**

**B Código *morse_pr2.py*** **87**

**Bibliografía** **89**

# ÍNDICE DE FIGURAS


1.1 Menú principal de la aplicación Android previamente desarrollada. . . . . 2
1.2 Elementos que componían el sistema previamente desarrollado. . . . . 3
1.3 Elementos que componen el sistema desarrollado. . . . . 3
1.4 Simulación en MORSE del robot PR2. . . . . 5
1.5 Utilidad Rviz de ROS. . . . . 5
1.6 Ejemplo de mapa generado por *Gmapping*. . . . . 6
1.7 Ejemplo de trayectoria hasta un punto objetivo generada por la librería SBPL. 6

2.1 Pantalla de configuración de la aplicación Android. . . . . 10
2.2 Visualización en Rviz de un mapa generado por *Gmapping*. . . . . 11
2.3 Entorno en MORSE correspondiente a la figura 2.2. . . . . 12
2.4 Ejemplo de un fichero con un entorno 2D discretizado. . . . . 13
2.5 Proceso de discretización de un entorno con obstáculos. . . . . 15
2.6 Representación gráfica del proceso de un algoritmo basado en métodos de búsqueda de grafo. . . . . 16
2.7 Ejemplo de entorno en Rviz con el mapa, el punto objetivo y el camino a éste. 18

3.1 Parámetros del mensaje que publica el sensor láser. . . . . 20
3.2 Análisis de *gaps* con el algoritmo *Closest Gap*. . . . . 23
3.3 Esquema de los puntos de los marcadores *triangle list*. . . . . 27
3.4 Ejemplo de entorno en Rviz con el sensor láser, el camino recorrido por el robot, el modelo del robot y los *gaps* navegables. . . . . 28

4.1 Principales componentes del sistema. . . . . 30
4.2 Diagrama de flujo del proceso que lleva a cabo el sistema. . . . . 31
4.3 Herramienta de ayuda desarrollada. . . . . 32
4.4 Pseudocódigo del paso 1 del análisis de *gaps* del algoritmo CG. . . . . 35
4.5 Pseudocódigo del paso 2 del análisis de *gaps* del algoritmo CG. . . . . 36

5.1 Entorno con obstáculos diseñado en Blender. . . . . 38
5.2 Simulación en MORSE del entorno de la figura 5.1. . . . . 38
5.3 Aspecto general de la herramienta Rviz. . . . . 39
5.4 Representación en Rviz del mapa global del entorno, el modelo del robot y el punto objetivo. . . . . 40
5.5 Mapa local del entorno visualizado por el robot en Rviz. . . . . 41
5.6 Representación del camino recorrido por el robot en Rviz. . . . . 44
5.7 Detección de espacios navegables en Rviz. . . . . 45

5.8 Camino hacia el punto objetivo en Rviz. . . . . 46
5.10 Vista del entorno de la figuras 5.8 y 5.9 en MORSE. . . . . 47
5.11 Proceso de teleoperación del robot en el entorno 1. . . . . 48
5.12 Proceso de teleoperación del robot en el entorno 2. . . . . 49
5.13 Proceso de teleoperación del robot en el entorno 3. . . . . 50
5.14 Proceso de teleoperación del robot en el entorno 4. . . . . 51
5.15 Proceso de teleoperación del robot en el entorno 5. . . . . 52

# ACRÓNIMOS

**TFG** Trabajo Final de Grado

**ROS** Robot Operating System

**MORSE** Modular Open Robots Simulation Engine

**SLAM** Simultaneous Localization And Mapping

**SBPL** Search-Based Planning Library

**CG** Closest Gap

**ARA*** Anytime Repairing A*

**URDF** Unified Robot Description Format

**SBPL** Search-Based Planning Library

**AD*** Anytime D*

**ANA*** Anytime Non-parametric A*

**ATRV** All Terrain Robot Vehicle

# RESUMEN

Los robots terrestres forman parte de nuestro presente ofreciendo múltiples aplicaciones como la intervención en lugares peligrosos o desconocidos para las personas. Por este motivo, resulta interesante conseguir la exploración de un entorno sin la necesidad de tener a la vista el robot. Planteada esta necesidad, se propone continuar con el TFG realizado por David Arévalo Jiménez el curso académico 2016-17. En ese trabajo se desarrolló una aplicación sobre el sistema operativo Android que permitía la teleoperación de robots móviles desde un dispositivo móvil.

En este Trabajo Final de Grado se pretende desarrollar una herramienta que permita al usuario junto con la aplicación Android teleoperar a distancia un robot sin riesgo de colisión. Para hacer esto posible, esta nueva herramienta proporciona información como: los espacios entre obstáculos por los que el robot puede navegar, un mapa del entorno, la posición y orientación del robot dentro del mapa, etc.

CAPÍTULO 1

# INTRODUCCIÓN

En este capítulo se expondrán las motivaciones y objetivos de este trabajo. También se aborda el punto de partida y las herramientas utilizadas durante el desarrollo del proyecto.

## 1.1 Motivación

La robótica móvil está cada vez más presente en nuestro día a día, por ejemplo para realizar tareas complicadas o acceder a entornos inaccesibles para los humanos.

Un claro ejemplo es el caso de los robots de servicio, dedicados, entre otras posibilidades, a la intervención en ambientes peligrosos, como las aplicaciones submarinas y espaciales, o incluso para uso doméstico.

En línea con estos avances, en este TFG se pretende implementar un *software* que permita a un usuario teleoperar un robot y visualizar su entorno sin estar presente en ese espacio, evitando así esas situaciones peligrosas o desconocidas.

## 1.2 Objetivo

El objetivo principal de este trabajo es la teleoperación de un robot móvil a distancia a través de una consola, sin visualizar el entorno por el que se desplaza directamente. Esto se consigue gracias a los sensores que lleva incorporados el robot, que proporcionan información de su entorno inmediato en tiempo real.

Para llevar a cabo este objetivo, se ha implementado parcialmente el algoritmo de navegación llamado *Closest Gap* con la intención de poder visualizar los espacios navegables entre obstáculos. Por otro lado, se ha utilizado una herramienta que crea un mapa del entorno con el propósito de poder determinar un camino a un punto destino previamente establecido.

Más concretamente, en este trabajo se utiliza un visualizador 3D que muestra un mapeado en 2D de los obstáculos, los espacios navegables entre ellos, el recorrido que

lleva a cabo el robot y la trayectoria correspondiente a un camino que permite al robot alcanzar un determinado punto objetivo del entorno del robot (ver figura 1.3).

## 1.3 Punto de partida

El punto de partida para desarrollar este proyecto ha sido un trabajo final de grado que consistía en una aplicación (ver fig. 1.1) capaz de recolectar muestras de los sensores de navegación de un smartphone o tablet (Android), así como controlar un robot móvil genérico empleando la información mencionada [1]. O dicho de otro modo, esa aplicación permitía teleoperar a un robot según los movimientos que el usuario hacía con su smartphone o tablet (Android) (ver figura 1.2). Para ello, se utilizaban librerías y paquetes ROS *(Robot Operating System)* [2], para el fusionado de información sensorial y soporte a diversos cálculos de índole geométrica, necesarios para dar lugar a los movimientos adecuados del robot. La visualización del entorno se realizaba mediante el simulador 3D llamado MORSE [3].

Figura 1.1: Menú principal de la aplicación Android previamente desarrollada.

Este proyecto pretende eliminar la dependencia de la visualización del simulador a la hora de teleoperar el robot, que es equivalente a decir que el usuario tenga a la vista el robot. Creamos una nueva herramienta que el teleoperador podrá ejecutar en un PC y donde podrá visualizar el entorno que rodea a la plataforma robótica y, mediante un algoritmo de planificación de caminos, se determina un camino que permite llevar al robot hasta el punto objetivo deseado, el cual se muestra al usuario para facilitar la teleoperación (ver figura 1.3).

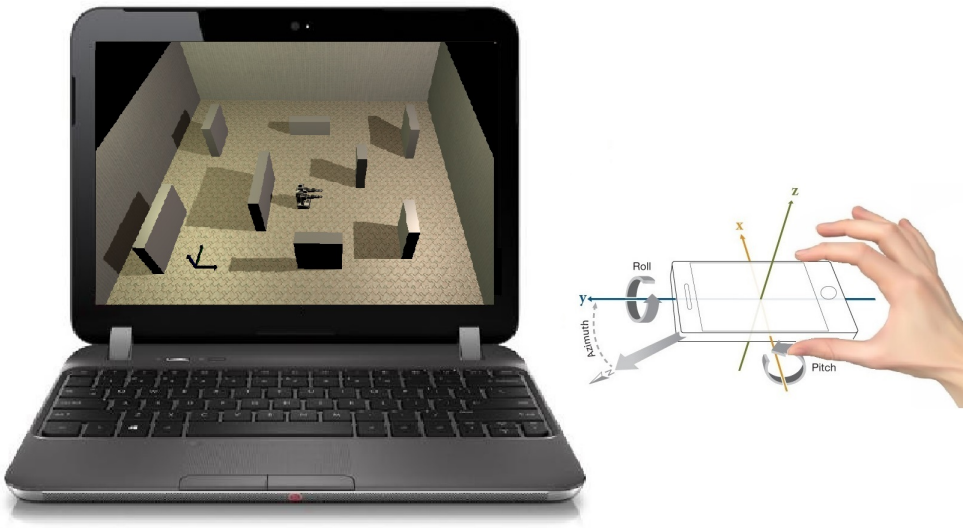



Figura 1.2: Elementos que componían el sistema previamente desarrollado.

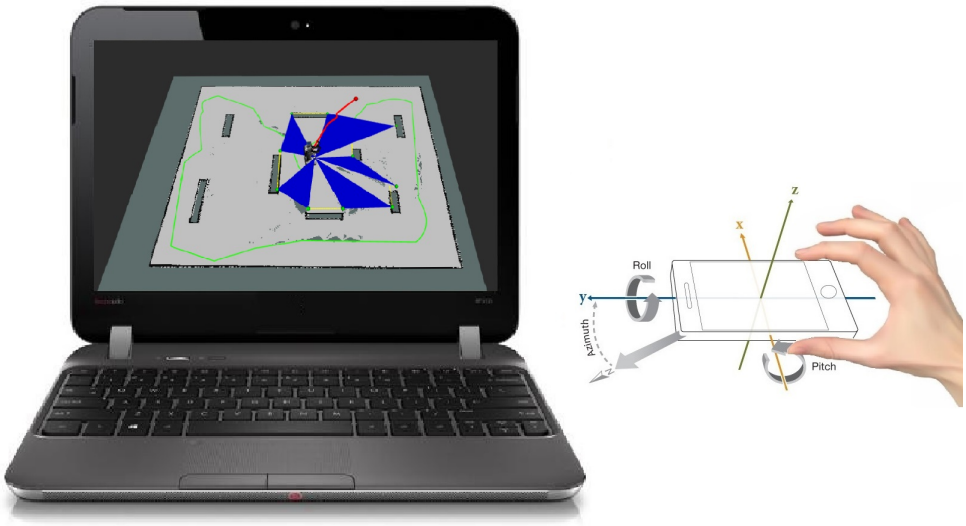



Figura 1.3: Elementos que componen el sistema desarrollado.

## 1.4 Herramientas utilizadas

Para realizar este trabajo se han utilizado dos herramientas: ROS [2] y Android Studio [4]. Estas herramientas se describen brevemente a continuación.

### 1.4.1 ROS

La parte del proyecto que no se ejecuta sobre el dispositivo móvil se ha desarrollado en ROS [2], plataforma que permite el desarrollo de software para el campo de la robótica. Este sistema operativo provee librerías y paquetes que facilitan la implementación de este

tipo de aplicaciones.

Los elementos principales de una aplicación ROS son los siguientes:

- Nodos, que son programas ejecutables que realizan una función concreta.
- Mensajes, que son un tipo de estructuras de datos utilizada para la comunicación entre nodos.
- Tópico, que es el "bus lógico" a través del cual los nodos intercambian mensajes. El proceso de comunicación que se lleva a cabo consiste en que un nodo publica un mensaje por un tópico determinado y uno o varios nodos se suscriben a dicho tópico para recibir la información.
- Nodo máster, que es el nodo principal de una aplicación ROS, el cual se encarga de registrar los nodos y los tópicos, permitiendo la conexión entre ellos.

El software de ROS está organizado en paquetes. Los paquetes de ROS utilizados para este proyecto son:

- **MORSE** (*Modular Open Robots Simulation Engine*) [3], que es un simulador de robots móviles para entornos 3D.

  Este simulador permite la visualización de escenas realistas a partir de scripts Python y Blender. Blender es un programa dedicado al modelado tridimensional de escenas que posteriormente se visualizan en MORSE.

  MORSE dispone de librerías que permiten emular sensores (p.e. escáneres láser, receptores GPS, odómetros, etc.), actuadores (p.e. controladores de velocidad de las ruedas) y robots móviles diversos (p.e. Pioneer 3DX, PR2, ATRV, etc.). De esta forma, este simulador permite la comunicación mediante tópicos que reciben/envían información desde los sensores/actuadores del robot. En este proyecto se utiliza un robot que hemos equipado con un actuador y dos sensores. Para definir estos elementos se realiza un fichero Python con los parámetros necesarios. El robot utilizado es el PR2, una imagen del cual se muestra en la figura 1.4. PR2 es un robot de forma humana con ruedas y dos brazos multiarticulados. Combina la movibilidad para navegar en entornos humanos y la destreza para manipular objetos en esos entornos. Cabe destacar que en este TFG no se han utilizado los brazos.

- **Rviz** [5], que es un visualizador de datos 3D.

  Mediante este visualizador se muestra la información sensorial que maneja el sistema, así como la información que se proporciona como asistencia a la teleoperación para navegar por el entorno desconocido. Se puede observar un ejemplo de uso de Rviz en la figura 1.5, donde se muestran figuras 3D para representar elementos deseados en un entorno definido.

- ***Gmapping*** [6], que es un paquete que permite crear mapas de entorno mediante un algoritmo SLAM *(Simultaneous Localization And Mapping)*.

  Este paquete proporciona un mapa en 2D del entorno a partir de la información suministrada por un sensor láser.

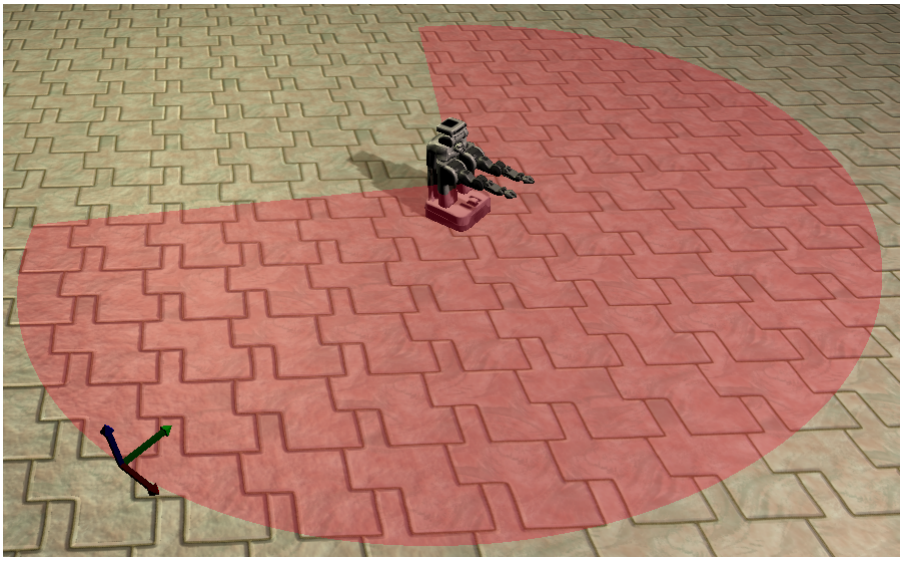


Figura 1.4: Simulación en MORSE del robot PR2 equipado con un sensor láser con cobertura de 270 grados en un entorno sin obstáculos.

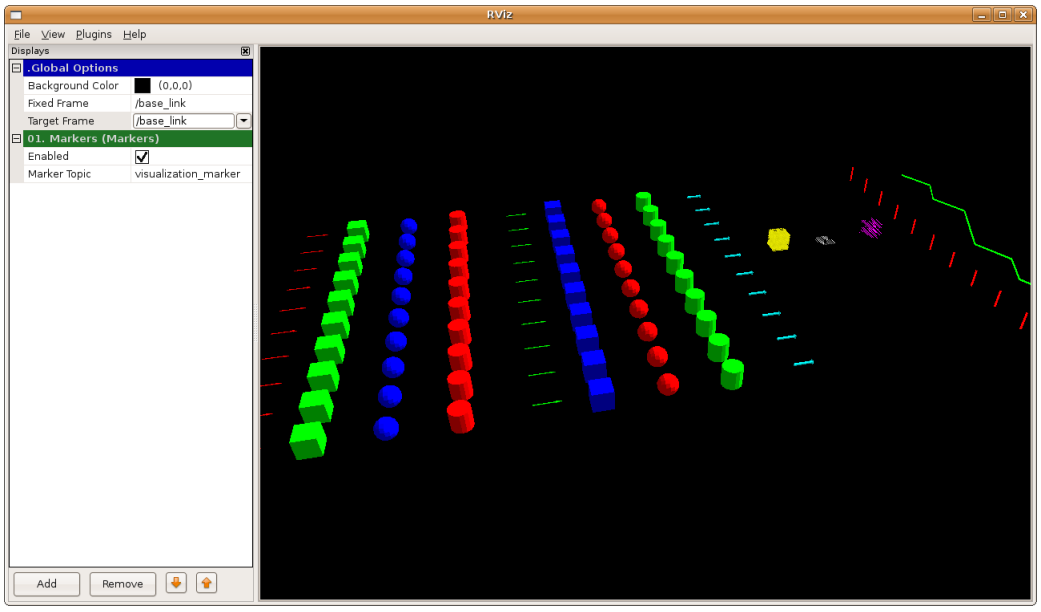



Figura 1.5: Utilidad Rviz de ROS.

El mapa es una representación matricial del entorno, donde cada celda almacena un valor relacionado con el coste que le supone al robot pasar por ella. En nuestro caso, las celdas con valor 100 representan un obstáculo, el valor 0 representa espacio libre y el valor 1 representa una zona no visitada (zona que el láser no ha alcanzado). La figura 1.6 representa gráficamente uno de estos mapas.

- **SBPL** (*Search-Based Planning Library*) [7], que es un paquete que incluye un conjunto de planificadores de caminos que generan una trayectoria hasta un punto

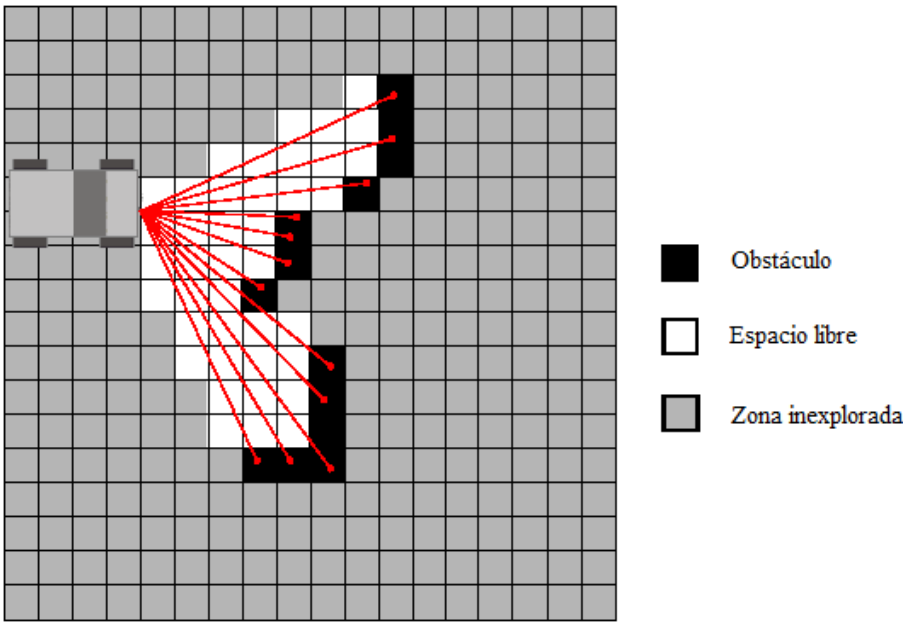



Figura 1.6: Ejemplo de mapa generado por *Gmapping*.

objetivo dado. En la figura 1.7 se representa una ruta de color azul desde la celda verde (posición del robot) hasta la celda roja (punto objetivo) proporcionada por esta librería sobre un mapa generado por el paquete *Gmapping*.

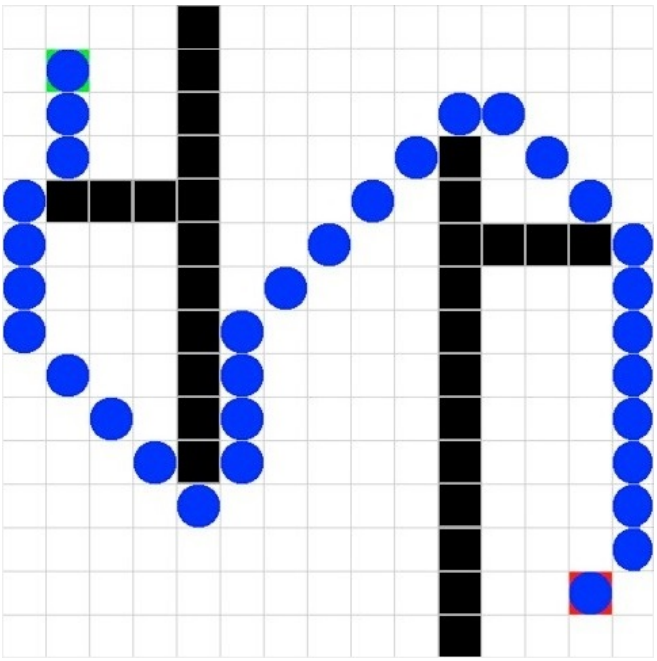


Figura 1.7: Ejemplo de trayectoria hasta un punto objetivo generada por la librería SBPL.

Todo nuestro código se ha escrito en lenguaje C++ y se ha incluido en un nuevo nodo ROS denominado *pr2_controller*. Este nodo se comunica con los paquetes/nodos MORSE,

Rviz, *Gmapping* y SBPL mediante la suscripción y publicación de tópicos que permiten obtener información del estado del robot así como también cambiar su estado.

### 1.4.2 Android Studio

Android Studio [4] es el entorno de desarrollo integrado oficial para la plataforma Android. Este programa se ha utilizado para poder modificar la aplicación Android de la figura 1.1. Más concretamente, a dicha aplicación se le ha añadido la funcionalidad de poder especificar el punto objetivo que se desea que alcance el robot.

## 1.5 Estructura del documento

El resto del documento se organiza en los siguientes capítulos:

- *Capítulo 2: Generación de un camino al objetivo*, en el que se detalla cómo se construye un mapa del entorno y cómo a partir de ese mapa se calcula el camino que permitiría al robot alcanzar el punto objetivo deseado.
- *Capítulo 3: Visualización del láser, del camino recorrido y de los gaps navegables*, el cual detalla la implementación realizada para conseguir visualizar en Rviz varios elementos adicionales de ayuda a la navegación tal como son: la nube de puntos de detección del sensor láser, el camino recorrido hasta el momento por el robot, y los espacios entre obstáculos que son navegables.
- *Capítulo 4: Estructura general del sistema*, en el que se describen todos los componentes que forman parte del sistema de teleoperación propuesto y cómo se interconectan entre sí. Además, se especifican los detalles de implementación del nodo ROS denominado *pr2_controller*.
- *Capítulo 5: Resultados experimentales*, donde se presentan y analizan los resultados obtenidos de las pruebas realizadas en diversos entornos.
- *Capítulo 6: Conclusiones*, el cual realiza una valoración general del trabajo realizado y los diferentes problemas encontrados.

CAPÍTULO 2

# GENERACIÓN DE UN CAMINO AL OBJETIVO

En el sistema de teleoperación propuesto, para poder generar y visualizar un camino a un punto objetivo se deben llevar a cabo una serie de pasos:

- Establecer el punto objetivo a alcanzar mediante la aplicación Android.
- Crear un fichero que contenga el mapa del entorno por el que se está moviendo el robot.
- Determinar el camino a seguir hasta el punto objetivo aplicando un algoritmo de planificación.
- Visualizar en Rviz el punto objetivo y el camino hacia éste.

A continuación se detalla cada una de estas fases y la conexión establecida entre el dispositivo Android y el nodo implementado.

## 2.1 Comunicación entre la aplicación Android y el nodo *pr2_controller*

La aplicación Android y el nodo *pr2_controller* se comunican entre sí siguiendo el modelo cliente-servidor; la aplicación Android actúa como cliente mientras que el nodo *pr2_controller* actúa como servidor. La decisión de utilizar este modelo proviene ya de [1].

La comunicación cliente-servidor se basa en el concepto de *sockets*. Un *socket* establece una conexión entre dos dispositivos que les permite intercambiar datos. La clase *Socket* incorpora todos los métodos necesarios para definir este tipo de conexiones. En nuestro caso, el servidor se ha escrito en lenguaje C++ y el cliente se ha programado en Java.

Por último, cabe destacar que el servidor es *multithread*, permitiendo así que varios clientes se puedan conectar simultáneamente al servidor.

## 2.2 Fijar el punto objetivo

Para establecer el punto objetivo al que el teleoperador quiere mover el robot móvil, se ha decidido utilizar la aplicación Android. De esta forma, el usuario podrá establecer de una forma sencilla las coordenadas a las que el robot debe dirigirse.

Para poder dar valor a esas coordenadas, se ha utilizado el apartado de ajustes ya existente en el Trabajo Final de Grado [1]. Así se podrán seleccionar las coordenadas del punto objetivo en el mismo lugar donde ya residen las demás opciones de configuración (ver figura 2.1). Simplemente se han añadido dos opciones más al diseño (*layout*) de la pantalla de configuración denominadas *Coordenada X* y *Coordenada Y* que permiten insertar un valor que será luego leído como una cadena de caracteres (*string*). A esta pantalla se entra pulsando el botón superior derecho del menú principal (ver figura 1.1).

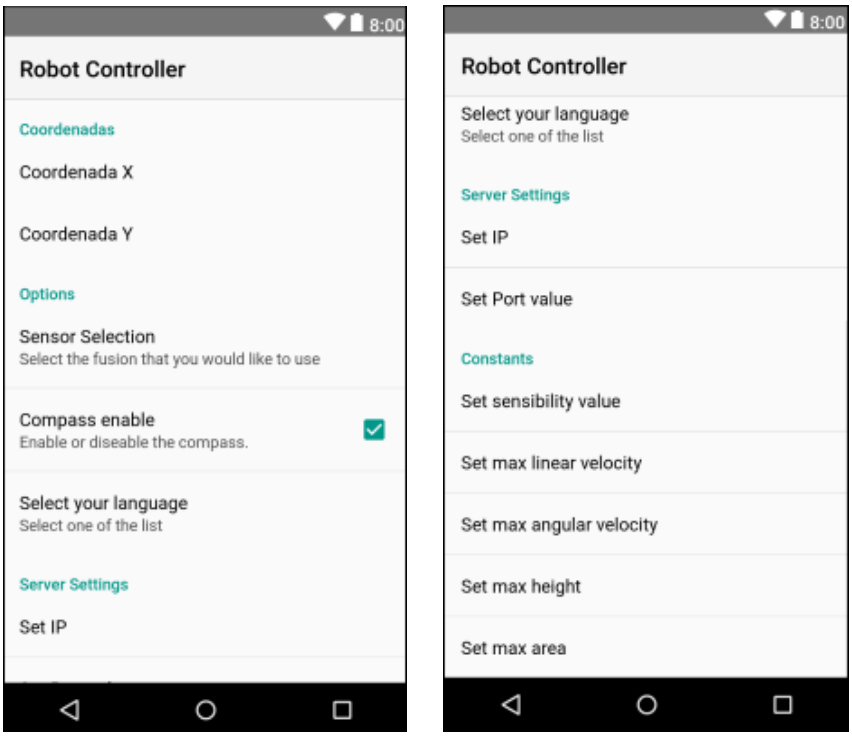



Figura 2.1: Pantalla de configuración de la aplicación Android.

Gracias a las clases *settings* y *OrientationVisualisationFragment* también ya existentes, se puede detectar cuando se ha cambiado el valor de una opción de configuración. Cuando esto ocurre, el nuevo valor se guarda.

En [1], la aplicación Android enviaba mensajes al nodo *pr2_controller* haciendo uso de la clase *Client*, la cual definía internamente un *socket*. Nosotros hemos modificado esa clase para incorporar la información de las coordenadas del punto objetivo dentro de los mensajes que se envían.

Finalmente, cuando el nodo *pr2_controller* recibe el mensaje de la aplicación Android, se extrae la información correspondiente a las coordenadas del punto objetivo para poder utilizarlas a lo largo del código.

## 2.3 Paquete *Gmapping*

Este paquete, como se ha mencionado en el capítulo 1, se encarga de crear un mapa 2D del entorno por el que navega a partir de la información recogida por el sensor láser. Esta información se publica sobre un tópico llamado *base_scan*, que contiene los datos necesarios para identificar los obstáculos que rodean al robot según la cobertura que el sensor láser tenga.

El nodo *Gmapping* se encarga de generar un mapa global del entorno, que publica sobre el tópico *map* en base a un mapa local de ocupación (*base_scan*).

En la figura 2.2 se puede observar la visualización en Rviz de un mapa del entorno creado por el paquete *Gmapping* una vez navegado en su totalidad. Este mapa corresponde con el escenario en MORSE de la figura 2.3. Como se puede observar, las celdas negras son los obstáculos, las grises son las celadas inexploradas y las blancas son espacio libre. Cabe destacar que las celdas se marcan como libres sólo cuando el haz del sensor láser detecta un obstáculo.

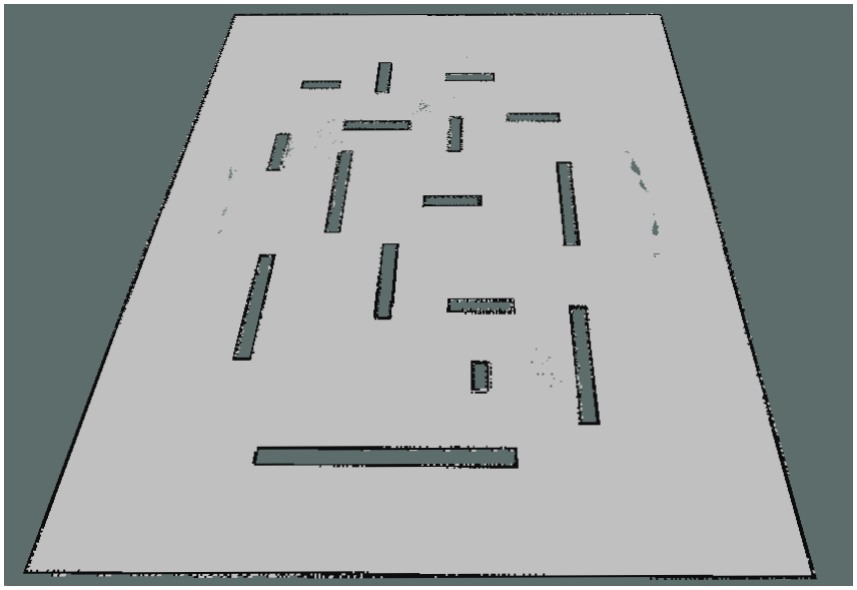


Figura 2.2: Visualización en Rviz de un mapa generado por *Gmapping*.

## 2.4 Librería SBPL

La librería SBPL (*Search-Based Planning Library*), desarrollada por el profesor Maxim Likhachev [8], proporciona un conjunto de algoritmos basados en búsqueda para la planificación de caminos. Las algoritmos de planificación se dividen en dos categorías:

- Búsqueda de grafo, algoritmos que buscan una ruta óptima sobre un gráfico que representa todo el espacio de trabajo. Este gráfico se representa mediante un grafo, un conjunto de nodos unidos entre sí.
- Muestreo probabilístico, algoritmos que generan puntos de muestreo aleatorios en el espacio de trabajo y se unen si existe un camino entre ellos.

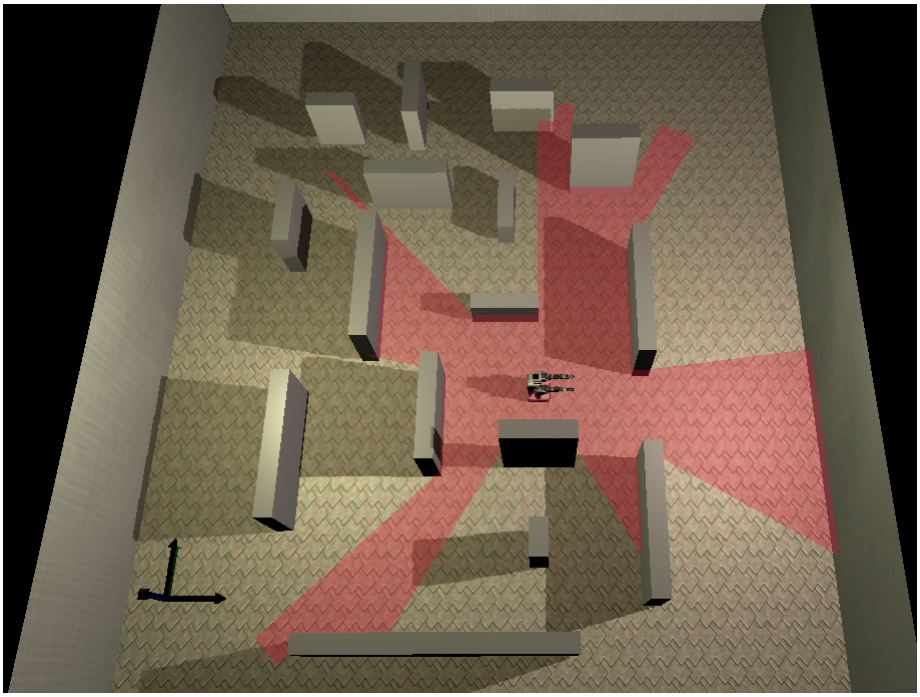


Figura 2.3: Entorno en MORSE correspondiente a la figura 2.2.

Todos los algoritmos de la librería SBPL se basan en la técnica de búsqueda de grafo.

Estos algoritmos generan caminos considerando el modelo cinemático del robot. De esta manera, se garantiza que los caminos generados son realizables por el robot.

Para poder aplicar cualquier algoritmo de planificación de la librería SBPL se necesita un fichero que contenga una representación discreta del entorno del robot.

A continuación, se detallan las características de este fichero, el proceso de aplicación del algoritmo y la posterior lectura del camino generado por el algoritmo de planificación.

### 2.4.1 Creación del fichero del entorno

A partir de la información del entorno proporcionada por el paquete *Gmapping*, se debe crear un fichero con extensión .cfg de aspecto similar a aquel que aparece en la figura 2.4.

El fichero se compone de los siguientes elementos:

- Número total de celdas (*discretization(cells)*): el entorno se define como una matriz de tamaño MxN. El primer valor corresponde con el número de columnas de la matriz (marcado en la figura 2.4 de color naranja) y el segundo con el número de filas (marcado en la figura 2.4 de color azul).

- Coste (*obsthresh*) a partir del cual una celda es considerada como ocupada por un obstáculo. En el ejemplo de la figura 2.4, este coste es 1 (marcado de color marrón).

- Las coordenadas de la celda inicial (*start(cells)*, representa la localización actual del robot en color verde en la figura 2.4). El primer valor de estas coordenadas corresponde con la columna de la celda y el segundo con la fila.

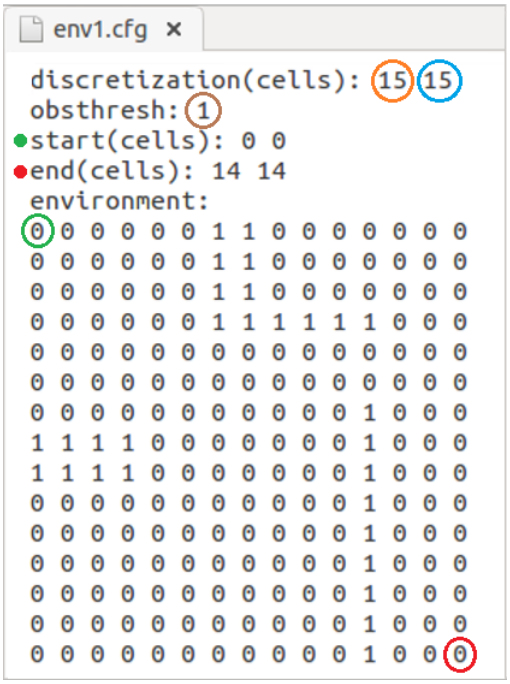



Figura 2.4: Ejemplo de un fichero con un entorno 2D discretizado.

- Las coordenadas de la celda objetivo (*end(cells)*, representa el punto objetivo que quiere alcanzar el robot y aparece marcado en color rojo en la figura 2.4). Estas son las coordenadas establecidas mediante la aplicación Android y recogidas por nuestro nodo *pr2_controller*. Al igual que antes, el primer valor de estas coordenadas corresponde con la columna de la celda y el segundo con la fila.

- El entorno como una matriz de celdas (*environment*). Para el ejemplo de la figura 2.4, el valor 0 representa espacio libre y el valor 1 significa espacio ocupado por un obstáculo, tal y como indica el parámetro *obsthresh*.

Se ha creado un *callback* (*gridCallback*) para que, cada vez que el mapa del entorno cambie, se actualice el fichero .cfg y se recalcule el camino que debe recorrer el robot hasta el punto objetivo.

Para poder garantizar que el robot podrá recorrer el camino generado por el algoritmo de planificación sin que se produzcan colisiones, es necesario tener en cuenta las dimensiones del robot. Para ello, se deben agrandar los obstáculos del entorno como mínimo el radio del robot.

Este proceso se lleva a cabo guardando en un *array* las posiciones de las celdas ocupadas. Para cada celda ocupada se expanden tantas celdas a su alrededor como sean necesarias para añadir el radio del robot.

Para saber cuántas se deben expandir se ha seguido la ecuación 2.1, donde *ceil* significa redondeo hacia arriba del valor obtenido, $R$ es el radio del robot y $r$ es la resolución de las celdas.

$$n = ceil\left(\frac{R}{r}\right) \tag{2.1}$$

En la figura 2.5 se puede observar un ejemplo de este proceso teniendo en cuenta que las celdas tienen una resolución de 0,1m y el radio del robot es de 0,3m. En la figura 2.5b, las celdas de color rojo representan los obstáculos que han sido detectados y las celdas de color naranja representan la expansión realizada sobre cada obstáculo. Una vez aplicada la ecuación 2.1, se obtiene un número total de celdas a expandir de 3; es por eso que los dos obstáculos están rodeados por tres capas de color naranja. Donde anteriormente solo correspondían con obstáculo las celdas de color rojo, ahora modificaremos las celdas desocupadas de color naranja a celdas ocupadas. En la figura 2.5c se puede observar el entorno discretizado con un valor de *obsthresh* de 100.

### 2.4.2 Aplicación del algoritmo de planificación

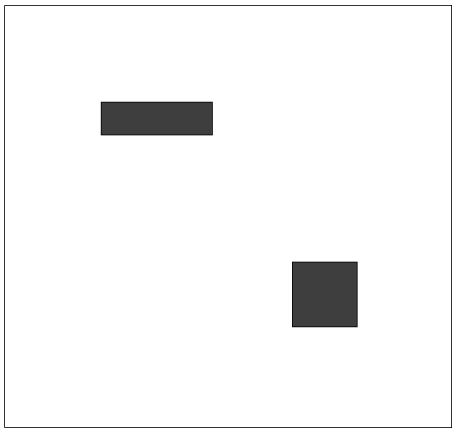
El siguiente paso a seguir consiste en aplicar un algoritmo de planificación sobre el fichero creado anteriormente para obtener un camino hasta el punto objetivo seleccionado. La librería SBPL incluye los siguientes algoritmos basados en métodos de búsqueda de grafos:

- ARA*, es una variante del algoritmo A* [9] con un mejor tiempo de respuesta. Esto se debe a que busca un camino válido entre un nodo origen y uno objetivo para entornos complejos. Incluso si se interrumpe la búsqueda antes de terminar el proceso, genera una solución rápida, no óptima, antes de optimizarla progresivamente.
- ANA*, es una variante del algoritmo A* [9] que generalmente encuentra una solución más rápida que el algoritmo ARA*. Esto se debe a que utiliza menos tiempo para mejorar el camino al nodo objetivo.
- AD*, es una variante del algoritmo D* [10] que se encarga de mejorar la trayectoria al nodo objetivo a medida que actualiza la información del entorno.
- R* es una versión aleatorizada del algoritmo A* [9] que se encarga de ejecutar una sucesión de búsquedas A* hacia nodos elegidos al azar hasta encontrar un camino al nodo objetivo.

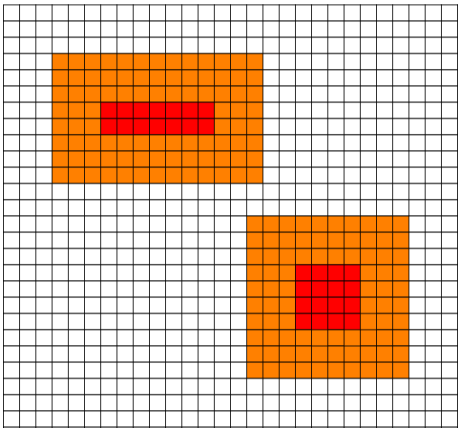
Debido a que en este TFG el entorno por el que el robot navega es parcialmente o totalmente desconocido, se ha utilizado el algoritmo AD*. A continuación, se entrará en detalle con el proceso que realiza este algoritmo.

En la figura 2.6 se puede observar una representación gráfica de la función de coste (ver ecuación 2.2) que aplica este algoritmo. En dicha figura, las celdas negras son obstáculos, las blancas representan espacio libre, las amarillas el camino calculado desde el nodo origen (*q start*) hasta *x*, que representa un nodo cualquiera del grafo, y la verde y roja son el nodo origen y el nodo destino (*q goal*), respectivamente. Las funciones y variables que componen la ecuación 2.2 son las siguientes:

- G(x), esta función devuelve el coste para alcanzar el nodo x desde el nodo *q start* (se representa en la figura 2.6 con las celdas de color naranja).
- H(x), es la función heurística de coste. Proporciona una medida aproximada de lo cerca que estás del punto objetivo. Más concretamente, esta función devuelve el coste estimado para alcanzar el nodo *q goal* desde el nodo x (se representa en la figura 2.6 con la flecha rosa).



(a) Entorno real con dos obstáculos.



(b) Mismo entorno que en 2.5a pero con los obstáculos agrandados.

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 100 100 100 100 100 100 100 100 100 100 100 100 100 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 100 100 100 100 100 100 100 100 100 100 100 100 100 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 100 100 100 100 100 100 100 100 100 100 100 100 100 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 100 100 100 100 100 100 100 100 100 100 100 100 100 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 100 100 100 100 100 100 100 100 100 100 100 100 100 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 100 100 100 100 100 100 100 100 100 100 100 100 100 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 100 100 100 100 100 100 100 100 100 100 100 100 100 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 100 100 100 100 100 100 100 100 100 100 100 100 100 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 100 100 100 100 100 100 100 100 100 100 100 100 100 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 100 100 100 100 100 100 100 100 100 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 100 100 100 100 100 100 100 100 100 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 100 100 100 100 100 100 100 100 100 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 100 100 100 100 100 100 100 100 100 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 100 100 100 100 100 100 100 100 100 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 100 100 100 100 100 100 100 100 100 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 100 100 100 100 100 100 100 100 100 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 100 100 100 100 100 100 100 100 100 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 100 100 100 100 100 100 100 100 100 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 100 100 100 100 100 100 100 100 100 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

(c) Contenido final del fichero .cfg correspondiente al entorno 2.5b.

Figura 2.5: Proceso de discretización de un entorno con obstáculos.

- $\varepsilon$, es un factor de inflado que se aplica sobre la función heurística H(x), y que tiene influencia sobre el coste del camino encontrado por el algoritmo de planificación. En concreto, esa influencia se puede describir como sigue: el coste del camino que el planificador encuentre entre *q start* y *q goal* nunca sería superior a $\varepsilon$ veces el coste del mejor camino que existe entre *q start* y *q goal*.
- F(x), es el coste total del camino desde el nodo *q start* hasta el nodo *q goal*.

El algoritmo AD* realiza la búsqueda del camino que une *q start* y *q goal* construyendo

un grafo sobre el entorno definido en el fichero .cfg. La búsqueda empieza siempre desde el nodo *q start*. Posteriormente, se expanden todos los nodos que sean vecinos de *q start* y se elige aquel cuyo coste (F(x)) sea menor. Este proceso se realiza hasta alcanzar el nodo *q goal*.

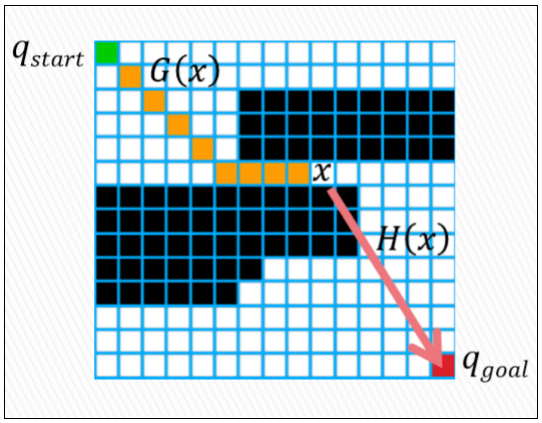



Figura 2.6: Representación gráfica del proceso que lleva a cabo un algoritmo basado en métodos de búsqueda de grafo.

$$F(x) = G(x) + \varepsilon \cdot H(x) \tag{2.2}$$

### 2.4.3 Lectura del fichero con la trayectoria generada

Cualquier algoritmo de la librería SBPL, y en particular el planificador AD*, escribe en un fichero denominado *sol.txt* la secuencia de celdas a recorrer del camino que ha calculado entre *q start* y *q goal*. Cada línea de este fichero codifica la columna y la fila de una celda. Por tanto, el fichero tendrá tantas líneas como celdas compongan el camino encontrado.

Si al aplicar el algoritmo de planificación AD*, no se encuentra ningún camino factible entre *q start* y *q goal* el fichero permanecerá vacío.

El siguiente paso es copiar en dos *arrays* la columna y la fila de cada celda guardada en el fichero *sol.txt*. Para ello, debemos realizar una lectura de este fichero.

Cabe destacar que para la lectura del fichero se utiliza la función *atoi()* que convierte un *string* a un número entero. El fichero contiene números con una cantidad de dígitos variables en formato *string*. Por lo tanto, esta función nos permite guardar la información como números enteros para tratarla posteriormente en la visualización.

## 2.5 Visualización en Rviz del punto objetivo y del camino a recorrer

Para eliminar la dependencia de la visualización del simulador (MORSE) se utiliza un visualizador 3D (Rviz) para asistir la teleoperación del robot.

Se ha implementado una función llamada *visualization_rviz* que contiene toda la información que se publica y posteriormente se añade en el *software* Rviz. En este apartado solo se aborda la visualización del punto objetivo y del camino generado por el algoritmo de búsqueda AD*. En el capítulo 3 se detalla la información restante.

Para mostrar los elementos que componen la asistencia a la teleoperación del robot se utilizan unos marcadores (*Display Markers* [11]) que permiten añadir en la pantalla de visualización formas primitivas en 3D.

Los marcadores utilizados para mostrar esta información son los siguientes:

- Esfera, para representar el punto objetivo. Se visualiza una pequeña esfera de color rojo que indica la localización del destino que quiere alcanzar el robot (ver figura 2.7).

- Conjunto de líneas enlazadas (*line strip*), para representar la ruta hacia el objetivo proporcionada por el algoritmo de planificación. Se realiza una lectura de los dos *arrays* que contienen las coordenadas de las celdas por las que debe pasar el robot y se representa un conjunto de líneas de color rojo que enlazan cada celda a recorrer con la consecutiva (ver figura 2.7).

El proceso que se lleva a cabo para publicar estos marcadores y que el *software* Rviz se suscriba y los represente, es publicar un mensaje de tipo *visualization_msgs/Marker* en un tópico que identifique cada elemento. Los parámetros que definen este mensaje se encuentran en la documentación de ROS [12].

El tópico que identifica el punto objetivo es *robot_goal* y el de la ruta hacia este objetivo es *line_strip*. En la figura 2.7 se puede observar un ejemplo de entorno en Rviz con esta información representada. El tópico *line_strip* es el conjunto de líneas rojas enlazadas hasta el destino y el tópico *robot_goal* es la esfera roja que indica el objetivo que el robot quiere alcanzar. Por otro lado, se puede observar el mapa generado por *Gmapping* en el tópico *map*. Este tópico se visualiza mediante: las líneas negras que representan los obstáculos detectados por el sensor láser, el entorno de color blanco que es el espacio libre y el de color gris que es el entorno inexplorado.

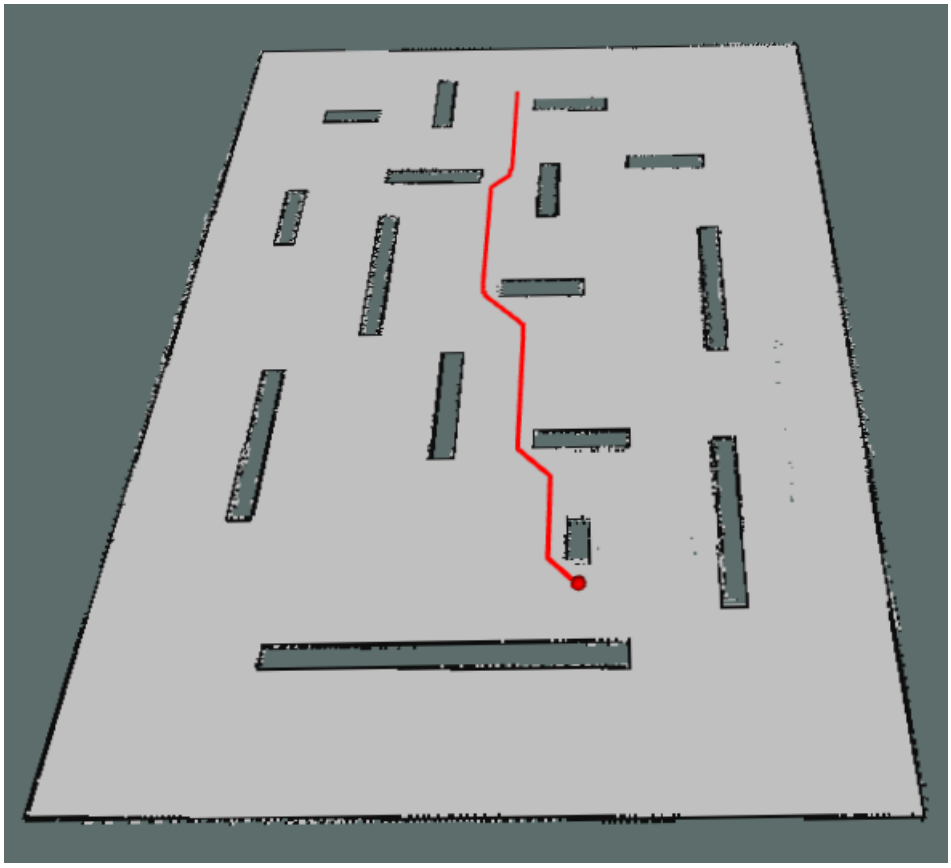


Figura 2.7: Ejemplo de entorno en Rviz con la representación del mapa generado por *Gmapping*, el punto objetivo establecido con el dispositivo Android y el camino al punto objetivo generado por el algoritmo de planificación AD*.

CAPÍTULO 3

# VISUALIZACIÓN DEL LÁSER, DEL CAMINO RECORRIDO Y DE LOS GAPS NAVEGABLES

Este capítulo se centra en explicar cómo hemos conseguido visualizar en el *software* Rviz la información que proporciona el sensor láser y el sensor de posicionamiento del robot. Además, también se detalla el algoritmo de navegación *Closest Gap* parcialmente implementado. Este algoritmo nos proporciona información sobre los espacios navegables entre obstáculos para, posteriormente, añadir a la visualización de Rviz y facilitar la teleoperación del robot por los lugares que permitan su paso.

## 3.1 Información proporcionada por el sensor láser

MORSE dispone de emuladores de sensores láser, como son los sensores SICK y Hokuyo, que permiten captar información del entorno en el que se encuentra el robot.

El sensor utilizado en este proyecto es el láser Hokuyo. Este sensor se encarga de generar rayos en direcciones predefinidas y de calcular la distancia a la que se encuentran los objetos que rodean al robot. El rango de detección, la resolución, la frecuencia de muestreo y el ángulo de cobertura del láser son parámetros configurables a través del fichero Python (*morse_pr2.py*).

MORSE proporciona la información recogida por el sensor láser emulado sobre el robot PR2 en el tópico llamado *base_scan*. Para obtener esta información en nuestro nodo, se debe crear un *callback* (*laserCallback*) vinculado con este tópico, que asigne a una variable la información del sensor láser cada vez que se actualice. Esta variable tiene que ser de tipo *sensor_msgs/LaserScan*. En ella se guardan todos los parámetros que caracterizan al sensor láser [13]. Seguidamente, se explican los más relevantes:

- Ángulo mínimo (*angle_min*), ángulo en radianes del primer rayo.
- Ángulo máximo (*angle_max*), ángulo en radianes del último rayo.

- Resolución (*angle_increment*), distancia angular (en radianes) entre medidas tomadas por el sensor láser.

- Rango máximo (*range_max*), distancia máxima que alcanza el láser del robot.

- Rangos (*ranges*), estructura de datos con las medidas tomadas por el sensor láser. Proporciona un array de tamaño número total de puntos de escaneo captados desde el ángulo mínimo hasta el máximo. Si no encuentra obstáculo proporciona el rango máximo y si encuentra obstáculo devuelve la distancia desde el centro del robot hasta el obstáculo detectado.

En la figura 3.1 se puede observar un ejemplo de la información proporcionada por el sensor láser con un ángulo mínimo de -2,35619 radianes (-135 grados), un ángulo máximo de 2,35619 radianes (135 grados), una resolución de 0,017453 radianes (1 grado) y un rango máximo de 5 metros. Esto implica que el láser abarca un total de 270 grados, de -135 grados a 135 grados. Además, significa que hay un total de 270 rayos debido a que los rayos que genera se separan los unos de los otros según la resolución del láser; como el láser tiene una resolución de 1 grado y un ángulo de cobertura de 270 grados, generará un total de 270 rayos. Del *array* que proporciona el parámetro rangos, el primer elemento corresponde con el ángulo -135 grados (rayo inicial) y el último con el de 135 grados (rayo final).


```
header:
  seq: 4039
  stamp:
    secs: 1535021513
    nsecs: 132817745
  frame_id: "/base_laser_link"
angle_min: -2.35619449615
angle_max: 2.35619449615
angle_increment: 0.0174532923847
time_increment: 9.25925924093e-05
scan_time: 1.0
range_min: 0.300000011921
range_max: 5.0
ranges: [5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 4.441563606262207, 4.382474899291992, 4.32623815536499, 4.272711277008057,
 4.221762657165527, 4.173271656036377, 4.245929718017578, 4.3686089515686035, 4.499999523162842, 4.640996932983398, 4.
792622089385986, 4.956051349639893, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.
0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.
5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 4.917801856994629,
 4.794478893280029, 4.678579330444336, 4.569510459899902, 4.466740131378174, 4.36979341506958, 4.278240203857422, 4.19
16961669921875, 4.1098103523254395, 4.0322675704956055, 3.958780527114868, 3.889087200164795, 3.8229498863220215, 3.76
0150194168091, 3.7361912727355957, 3.8106327056884766, 3.8893096446990967, 3.972539186477661, 4.060673236846924, 5.0,
5.0, 5.0, 5.0, 5.0, 5.0, 4.958191871643066, 4.907477378845215, 4.859254837036133, 3.114567756652832, 3.086397171020508
, 3.059655189514416, 3.0342893600463867, 3.0102500915527344, 2.9874913692474365, 2.965970516204834, 2.945648670196533,
2.9264886379241943, 2.908456802368164, 2.8915212154388428, 2.875652313232422, 2.860823392868042, 2.8470094203948975, 2
.8341875076293945, 2.822336196899414, 2.811436653137207, 2.8014707565307617, 2.7924234867095947, 2.7842791080474854, 2
.7770259380340576, 2.7706520557403564, 2.7651476860046387, 2.7605044841766357, 2.7567155361175537, 2.7537736892700195,
 2.7516763210296663, 2.7504189014434814, 2.75, 2.7504189014434814, 2.7516763210296663, 2.7537736892700195, 2.75671553611
75537, 2.7605044841766357, 2.7651476860046387, 2.7706520557403564, 2.7770259380340576, 2.7842791080474854, 2.792423486
7095947, 2.8014707565307617, 2.811436653137207, 2.822336196899414, 2.8341875076293945, 2.8470094203948975, 2.860823392
868042, 2.875652313232422, 2.8915212154388428, 2.908456802368164, 2.9264886379241943, 2.945648670196533, 2.96597051620
4834, 2.9874913692474365, 3.0102500915527344, 3.0342893600463867, 3.059655189514416, 3.086397171020508, 3.1145677566528
32, 3.1442236690032959, 3.1754262447357178, 3.20824170111260986, 3.2427406311035156, 3.2789993286132812, 3.3170995712280
273, 3.3571300506591797, 3.399186611175537, 3.443373203277588, 3.489800453186035, 3.538588523864746, 3.589870214462280
3, 3.643785238265991, 3.70048999078637695, 3.760150194168091, 3.8229503631591797, 3.889087200164795, 3.958780527114868,
 4.0322675704956055, 4.1098103523254395, 4.1916961669921875, 4.27824068069458, 4.36979341506958, 4.505014896392822, 4.
445082187652588, 4.3880414962768555, 4.333749294281006, 4.282073497772217, 4.232889652252197, 4.18608331680297785, 4.14
15486335754395, 4.09918737411499, 4.0589070320129395, 4.020623683929443, 3.9842584133148193, 3.9497368335723877, 3.916
9914722442627, 3.885958671569824, 3.856579303741455, 3.8287980556488037, 3.8025643825531006, 3.7778310775756836, 3.754
5535564422607, 3.8994622230529785, 4.121466159820557, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0,
 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.
0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 4.131175994873047, 4.023656368255615, 3.922755241394043
, 3.8279287815093994, 3.7386903762817383, 3.6546056270599365, 3.5752854347229004, 3.500378131866455, 3.429569482803344
7, 3.362572431564331, 3.299128532409668, 3.239001750946045, 3.181980609893799]
intensities: []
---
```


Figura 3.1: Parámetros del mensaje que publica el sensor láser.

## 3.2 Información proporcionada por el sensor posición

Para conocer la posición y orientación del robot en el entorno de navegación, MORSE dispone de diversos emuladores. Para este proyecto se ha utilizado la odometría, que estima la posición y orientación relativa del robot móvil usando la información sobre la rotación de las ruedas. En los ejes de las ruedas del robot disponemos de encoders, sensores que miden las revoluciones de un eje. Éstos nos proporcionan las vueltas que ha dado un eje en un intervalo de tiempo y por lo tanto, nos permite localizar el robot de forma relativa a su punto de partida en el entorno.

El tópico *odom* publicado por MORSE devuelve la posición y orientación del robot mediante odometría en el entorno de navegación. Se ha creado un *callback* (*poseCallback*) asignado a este tópico de tipo *nav_msgs/Odometry* para que actualice una variable cada vez que la posición y/u orientación del robot varíe. Este mensaje especifica las coordenadas de la posición en un vector (x, y, z) y la orientación en forma de cuaternión (x, y, z, w). Además, proporciona la velocidad lineal y angular del robot con vectores (x, y, z) que definen el desplazamiento y la rotación sobre los ejes en m/s y rad/s, respectivamente.

## 3.3 Algoritmo de navegación *Closest Gap*

Para facilitar la teleoperación del robot se ha implementado parcialmente el algoritmo de navegación *Closest Gap*. Este algoritmo se presentó en el artículo *Closest Gap Based (CG) Reactive Obstacle Avoidance Navigation for Highly Cluttered Environments* y fue propuesto por Muhannad Mujahad, Dirk Fischer, Bärbel Mertsching y Hussein Jaddu [14].

Antes de entrar en detalle con la explicación del algoritmo *Closest Gap*, se va a clasificar este algoritmo dentro de las diferentes arquitecturas de navegación existentes. Posteriormente, se intoducirán los principales parámetros que caracterizan al mencionado algoritmo.

### 3.3.1 Clasificación de los algoritmos de navegación

Para comprender las propiedades del algoritmo *Closest Gap*, se van a describir las tres arquitecturas de control de robots que predominan en la actualidad:

- Arquitecturas deliberativas, basadas en el paradigma *Sense-Plan-Act* (SPA) utilizan un mapa discretizado del entorno para planificar la ruta que debe seguir el robot.

- Arquitecturas reactivas, reaccionan a la información proporcionada por los sensores de forma inmediata, eliminando la fase de planificación ya que no disponen de un mapa del entorno. Se trata del paradigma *Sense-Act* (SA).

- Arquitecturas híbridas, son una combinación de los dos paradigmas anteriores. Más concretamente, se forma una estructura basada en capas de forma que se combina la capacidad de planificar una ruta a seguir con la capacidad de reacción ante situaciones inesperadas. Su paradigma es *Plan, Sense-Act*.

Dentro de esta clasificación, el algoritmo *Closest Gap* es un algoritmo de navegación reactivo con una capacidad de respuesta rápida e ideal para entornos repletos de obstáculos.

### 3.3.2 Introducción al algoritmo

El algoritmo *Closest Gap* se desarrolló con intenciones de crear una nueva estrategia de navegación para robots móviles en entornos desordenados y complejos al contener espacios estrechos. Su principal contribución recae en la utilización de un nuevo método para analizar los espacios libres de obstáculos que se encuentran próximos al robot, generando rutas seguras y aplicando desviaciones estrictas contra obstáculos.

Este algoritmo se divide en las tres siguientes fases:

- Análisis de *gaps*, se centra en buscar *gaps* alrededor del robot. Los *gaps* son los espacios libres de obstáculos que tienen un tamaño suficientemente grande como para que el robot se mueva por ellos.
- Determinar la dirección del movimiento del robot.
- Aplicar un método de navegación reactiva que haga que el robot se mueva en la dirección calculada en la fase anterior de forma autónoma.

Cabe decir que no se ha implementado por completo el algoritmo *CG*, debido a que el objetivo de este TFG no era el de obtener un robot autónomo, sino el de facilitar la tarea de teleoperaciónde un robot al reconocer los espacios navegables entre los obstáculos que el robot se va encontrando. Por lo tanto, nos centraremos en la fase implementada, análisis de *gaps*.

### 3.3.3 Análisis de *gaps*

El objetivo principal de esta fase es el de encontrar una lista definitiva de los *gaps* navegables a partir de la información proporcionada por el sensor láser.

Antes de explicar el procedimiento que se lleva a cabo el análisis de *gaps*, es necesario dar una serie de definiciones:

- Las medidas tomadas por el sensor láser se guardan en una estructura de datos llamada *scan*. La medida correspondiente al primer/último rayo generado por el sensor láser queda registrada en la posición 0/n-1 de esta estructura de datos. Por *scan i* se entenderá la información contenida en la posicion *i* de la estructura de datos *scan*, con $0 \leq i \leq n-1$.
- *L* es la lista de obstáculos detectados por el sensor láser.
- El rango máximo de detección del sensor láser es $d_{max}$.
- d(*A, B*) es la función que devuelve la distancia entre los puntos A y B.
- $dist_c(\theta_i, \theta_j)$ es la distancia angular entre el ángulo del obstáculo *i* y el ángulo del obstáculo *j*.

Por otro lado, la aplicación de la fase de análisis de *gaps* requiere:

- La localización ($x_{robot}$) y el radio ($R$ ) del robot.

- El rango máximo de detección del sensor láser ($d_{max}$).
- Una lista ($L$) de puntos de obstáculo dónde un obstáculo es $O_i^L$ (es importante recalcar que $O_i^L$ representa la localización del obstáculo detectado por el *scan i*).

Para encontrar los *gaps* navegables se recorren todas las posiciones de la estructura de datos *scan*. De ahí, se buscan discontinuidades entre posiciones adyacentes, *scan* i y *scan* j. Existen dos tipos de discontinuidades:

- Discontinuidad de tipo 1: ocurre cuando la distancia entre los obstáculos detectados por el *scan i* y el *scan j* es mayor al diámetro del robot (ver ecuación 3.1).

$$|d(x_{robot}, O_j^L) - d(x_{robot}, O_i^L)| > 2R \tag{3.1}$$

- Discontinuidad de tipo 2: ocurre cuando uno de los dos *scans* adyacentes devuelve el rango máximo del sensor (ver ecuación 3.2).

$$d(x_{robot}, O_j^L) = d_{max} \; AND \; d(x_{robot}, O_i^L) < d_{max} \tag{3.2}$$

Si $j > i$, se produce una discontinuidad ascendente en el *scan i*. Por contraposición, si $i > j$, se produce una discontinuidad descendente en el *scan j*. Las discontinuidades de tipo 1 tienen prioridad sobre las de tipo 2.

El análisis de *gaps* consiste en dos pasos: el primero implica encontrar la lista de todos los *gaps*, y el segundo elimina los *gaps* innecesarios. Para entender los pasos que sigue el algoritmo *Closest Gap*, se utiliza la figura 3.2 del artículo original [14]. A continuación, se detallan las fases de cada paso:

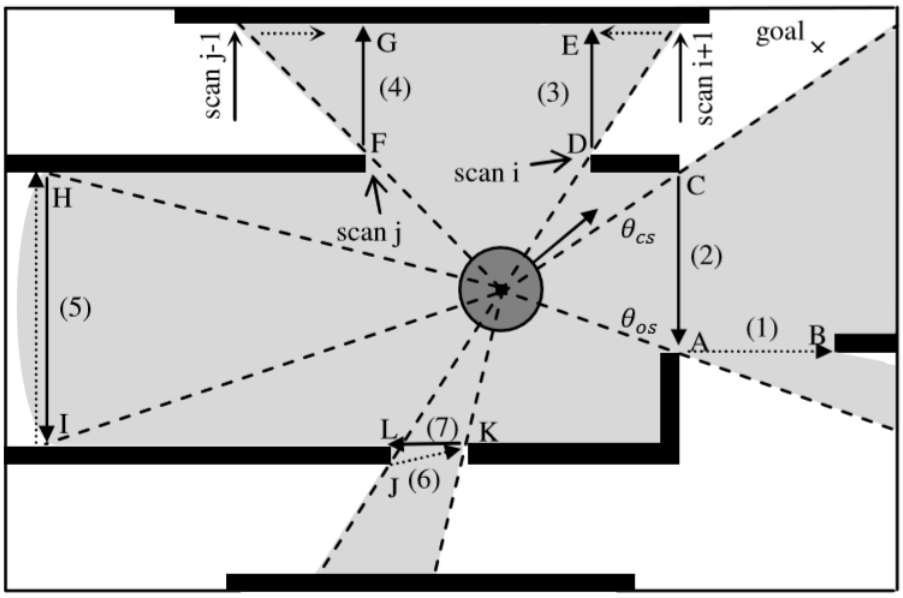



Figura 3.2: Análisis de *gaps* con el algoritmo *Closest Gap*. En el primer paso se detectan los *gaps* etiquetados del 1 al 7. En el segundo paso se eliminan los *gaps* 1 y 6.

- Paso 1: recorremos las *n* posiciones de la estructura de datos *scan* dos eces. En primer lugar, se recorre desde el *scan* 0 al *scan* n-1 buscando discontinuidades ascendentes (búsqueda *forward*). En segundo lugar, se recorre desde el *scan* n-1 al *scan* 0 buscando discontinuidades descendentes (búsqueda *backward*). Seguidamente, se explican con más detalle ambas búsquedas.

  – Búsqueda *forward*: supongamos que se encuentra una discontinuidad ascendente de tipo 1 o 2 en el *scan i*. Este *scan* es el que determina el principio del *gap* (punto D y H en la figura 3.2). Para determinar el segundo extremo de ese *gap*, se debe considerar el tipo de la discontinuidad encontrada:

    * Para una discontinuidad de tipo 1 se define que el siguiente extremo del *gap* (*scan j*) se encuentra en el espacio $S^+ = \{i+1, ..., n-1\}$. El punto *j* tiene que satisfacer que sea la distancia más corta al primer lado del *gap* y que la distancia angular recorrida sea inferior a $\pi$ (ver ecuación 3.3). El punto E de la figura 3.2 se ha encontrado al detectar este tipo de discontinuidad.

      $$d(O_i^L, O_j^L) \leq d(O_i^L, O_k^L) \; AND \; dist_c(\theta_i, \theta_j) \leq \pi \tag{3.3}$$

      donde *k* es cualquier *scan* dentro del espacio $S^+$.

    * Para una discontinuidad de tipo 2 se continúa recorriendo la estructura de datos *scan* desde la posición $i+1$ hasta encontrar una posición cuya distancia sea inferior al rango máximo de detección del sensor láser. El punto I de la figura 3.2 se ha encontrado al detectar este tipo de discontinuidad.

    Una vez encontrado el segundo extremo del *gap* se continuará la búsqueda a partir del *scan j+1*. La búsqueda *forward* se encuentran los *gaps* 1, 3, 5 y 6 (los *gaps* 3 y 6 se encuentran en discontinuidades de tipo 1, mientras que los *gaps* 1 y 5 en discontinuidades de tipo 2).

  – Búsqueda *backward*: se lleva a cabo el mismo proceso que en la búsqueda *forward* pero el recorrido al revés, desde el *scan* n-1 al *scan* 0. Supongamos ahora que se encuentra una discontinuidad descendente de tipo 1 o 2 en el *scan j*. De nuevo, el segundo extremo del *gap* se define según el tipo de discontinuidad encontrada:

    * Para una discontinuidad de tipo 1 se define que el siguiente extremo del *gap* (*scan i*) se encuentra en el espacio $S^- = \{j-1, ..., 0\}$. El punto *i* tiene que satisfacer que sea la distancia más corta al primer extremo del *gap* y que la distancia angular recorrida sea inferior a $\pi$ (ver ecuación 3.4). El punto G de la figura 3.2 se ha encontrado al detectar este tipo de discontinuidad.

      $$d(O_i^L, O_{j+1}^L) \leq d(O_k^L, O_{j+1}^L) \; AND \; dist_c(\theta_{j+1}, \theta_i) \leq \pi \tag{3.4}$$

      donde *k* es cualquier *scan* dentro del espacio $S^-$.

    - Para una discontinuidad de tipo 2 encontrado el *scan j* se escanean los puntos anteriores hasta encontrar un *scan* con una distancia inferior al rango máximo del sensor. El punto H de la figura 3.2 se ha encontrado al detectar este tipo de discontinuidad.

  Una vez encontrado el segundo extremo de un *gap* se continuará la búsqueda a partir del *scan j+1*. En el ejemplo de la figura 3.2, con la búsqueda *backward* se encuentran los *gaps* 7, 4 y 2 en discontinuidades de tipo 1 y el *gap* 5 se vuelve a encontrar en discontinuidad de tipo 2.

- Paso 2: al completar las dos búsquedas (*forward* y *backward*) tenemos una lista de *gaps* encontrados. De estos *gaps*, debemos eliminar los que se encuentren dentro de otros (en la figura 3.2, los *gaps* 1 y 6 se eliminan por encontrarse dentro de los *gaps* 2 y 7, respectivamente; el *gap* 5 solo se guarda una veza pesar de ser encontrado tanto en la búsqueda *forward* como en la búsqueda *backward*), y los que tengan un ancho inferior al diámetro del robot (*gap* 7 de la figura 3.2).

  Una vez eliminados los *gaps* que cumplan estas condiciones, se obtiene una lista definitiva de *gaps* navegables. En el ejemplo de la figura, corresponden con los *gaps* 2, 3, 4 y 5.

## 3.4 Visualización en Rviz

Recogida la información proporcionada por el sensor láser y de posición del robot y encontrados los *gaps* navegables, podemos enviar al *software* Rviz estos datos para que sean visualizados.

A continuación, se detalla el procedimiento realizado para visualizar el escáner láser, la trayectoria recorrida por el robot, el modelo del robot PR2 y los *gaps* navegables. Finalmente, se muestra un ejemplo de entorno en Rviz con la visualización conjunta de estos tres elementos.

### 3.4.1 Escáner láser del robot

Para proyectar la información proporcionada por el sensor láser en Rviz se ha utilizado la clase *LaserProjection* recogida en el paquete *laser_geometry* [15]. Esta clase transforma los mensajes de tipo *sensor_msgs/LaserScan* a *sensor_msgs/PointCloud*. Esto significa que proyecta cada rayo del láser a lo largo del ángulo apropiado tomado como centro el lugar donde se encuentra el robot.

La llamada a esta clase se realiza en el propio *callbak* del láser, *laserCallback*. Una vez recogidos los datos del sensor, se proyecta esta información sobre la variable *cloud_laser_scan*. Finalmente, se publica esta nube de puntos, que posteriormente añadiremos al *software* Rviz.

### 3.4.2 Trayectoria recorrida por el robot y el modelo del robot PR2

Para saber el recorrido que ha llevado a cabo nuestro robot, se ha creado una variable que guarda la posición actual del robot. Esta variable es *pose_aux*, encargada de actualizar la posición del robot tras cada ejecución del *callback* de posición, *poseCallback*.

Se ha creado esta variable auxiliar debido a que debe contener un mensaje de tipo *geometry_msgs/PoseStamped* [16] para poder tratarlo posteriormente en Rviz.

Una vez recogida la posición, en la función *visualization_rviz*, en caso de que sea distinta a la anterior, se añade esta nueva posición a un vector que contiene todas las posiciones recorridas por el robot desde su punto de partida. Para ello, se utiliza la función *push_back(pose_aux)*. Finalmente, se publica este vector llamado *global_path* para añadir en el entorno de Rviz la trayectoria recorrida por el robot. El recorrido se visualiza mediante un conjunto de líneas enlazadas de color verde.

Por otro lado, se visualiza el robot PR2 mediante el paquete *pr2_description* para conocer la orientación del robot. Este paquete contiene la descripción del robot PR2 en formato URDF. Se trata de un formato XML con especificaciones para representar el modelo del robot en Rviz.

### 3.4.3 *Gaps* navegables

Para representar los *gaps* navegables en Rviz, se han utilizado los marcadores de esferas y los *triangle list*, para mostrar los extremos del *gap* y el espacio que abarca cada *gap* desde el centro del robot, respectivamente.

Para ello, en la función *visualization_rviz* se consulta la información guardada en la variable *cloud_laser_scan*. Como se ha comentado en la sección 3.4.1, esta variable es un vector que contiene las medidas de distancia proporcionadas por el sensor láser, por lo que debemos recorrer todas las posiciones de este vector para encontrar los índices de los *gaps* navegables.

Cada tipo de marcador se ha tratado de forma diferente:

- Marcador esfera, debemos crear dos esferas que marquen los extremos de cada *gap* navegable. Los índices de cada *gap* navegable los hemos guardado en los *arrays* *index_gaps_i* e *index_gaps_j*, por lo tanto, debemos consultar dentro de la nube de puntos (*cloud_laser_scan*) estos índices. Las esferas tienen una dimensión (x, y, z) de (0.2m , 0.2m, 0.2 m) y se representan de color verde. Finalmente, publicamos esta información recopilada en la variable *marker_gaps* que añadiremos en Rviz.

- Marcador *triangle list*, este marcador junta cada conjunto de tres puntos y los trata como un triángulo, con los índices 0-1-2, 3-4-5, etc. Debemos guardar en tres *arrays* distintos las tres coordenadas (x, y, z) del conjunto de puntos que forman el triángulo que marcará el espacio que abarca cada *gap* navegable. Los tres puntos que forman cada triángulo son el *scan i*, el *scan j* y el centro del robot. En la figura 3.3 se puede observar un esquema de cómo se guardan estos *arrays*. En este ejemplo, hay seis *gaps* navegables, tres encontrados en la búsqueda *forward* y tres encontrados en la búsqueda *backward*. Seguidamente, estos *arrays* se copian a una variable (*point_triangle*) con tipo de mensaje apropiado para visualizar en Rviz (geometry_msgs/Point) mediante la función *push_back(point_triangle)*. Finalmente, se publica *point_triangle* y lo añadimos al *software* Rviz. Los triángulos se representan de color azul.

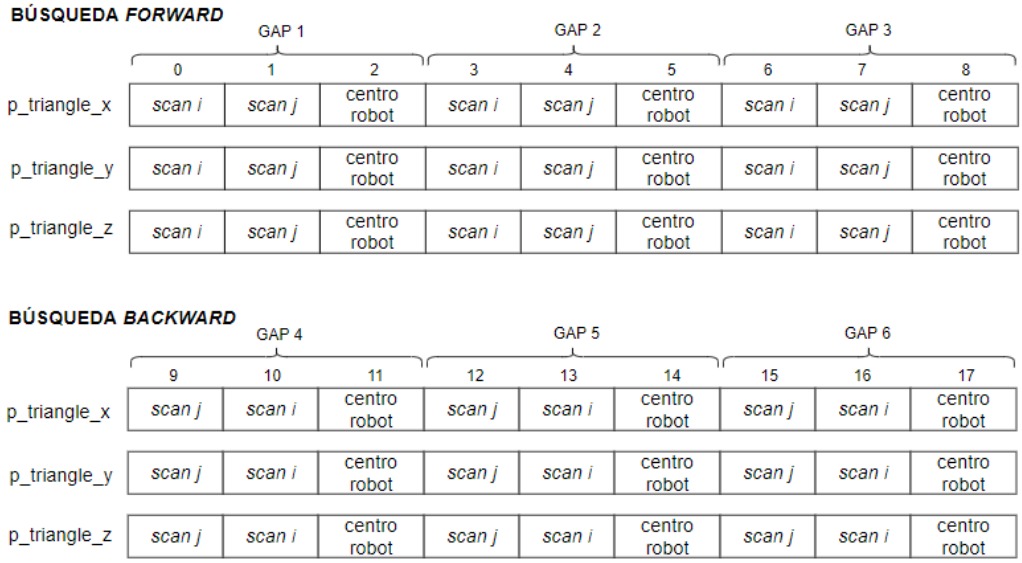



Figura 3.3: Este esquema muestra cómo se guardan los puntos que darán lugar a los marcadores *triangle list*.

### 3.4.4 Representación conjunta en Rviz

En este apartado, se muestra un ejemplo (ver figura 3.4) de la visualización en Rviz de los tres elementos detallados previamente en un entorno con obstáculos. Los puntos amarillos representan las medidas tomadas por el sensor láser; la línea verde es el camino recorrido por el robot desde el instante inicial de la simulación; y para representar los *gaps* navegables tenemos dos marcadores: los puntos verdes son los extremos de cada *gap* (*scan i* y *scan j*) y los triángulos azules representan el espacio que abarca cada *gap* hasta el centro del robot.

Este ejemplo de entorno es una replica de aquel mostrado en la figura 3.2. Como se puede observar marca exactamente los 4 *gaps* navegables descritos en la sección 3.3.3.

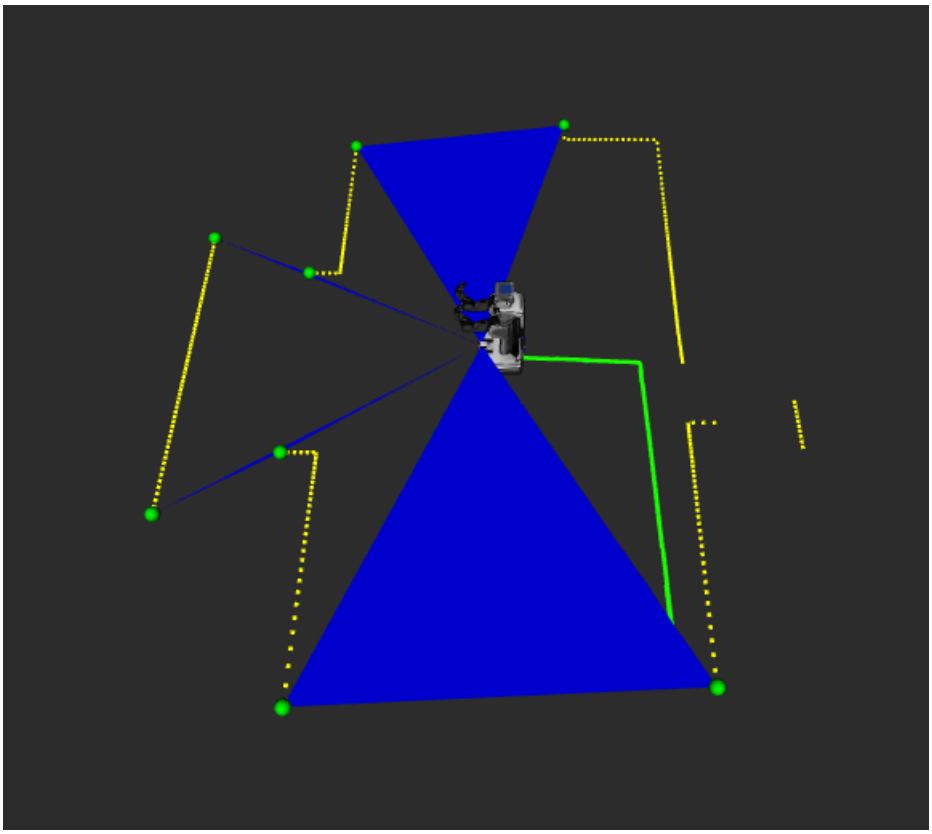


Figura 3.4: Ejemplo de entorno en Rviz con la representación de la información proporcionada por el sensor láser, el camino recorrido por el robot, el modelo del robot y los *gaps* navegables.

CAPÍTULO 4

# ESTRUCTURA GENERAL DEL SISTEMA

La estructura general del sistema de teleoperación propuesto sigue aquella adoptada por el Trabajo Final de Grado del alumno David Arévalo Jiménez [1], tanto en lo que respecta al nodo creado en ROS como a la aplicación móvil creada sobre Android. Ese nodo ROS y esa aplicación Android han sido modificados para poder añadir todas las funcionalidades que son objeto del presente proyecto.

En este capítulo se describen cuáles son los componentes del sistema y cómo se interconectan entre sí. Finalmente, se muestra un ejemplo de la herramienta de ayuda a la teleoperación a distancia de robots móviles desarrollada y se detallan las funciones implementadas sobre el nodo ROS denominado *pr2_controller*.

## 4.1 Componentes del sistema y su interconexión

Como se ha mencionado en el capítulo 1, el sistema de teleoperación propuesto está formado por diversos nodos ROS y un dispositivo móvil Android. En la figura 4.1, se pueden observar todos los componentes que forman parte de este sistema.

Para que todos los nodos ROS se comuniquen mediante publicaciones y suscripciones de mensajes, debemos tener un nodo máster que controle esta transferencia (las líneas de color naranja de la figura 4.1 son las que representan este enlace).

Cada nodo se representa con un círculo y se enlazan mediante flechas con la información publicada. El origen de la flecha significa el nodo encargado de publicar el mensaje y hacia donde apunta es el nodo que se suscribe a dicho mensaje.

Por otro lado, el dispositivo Android se representa con un rectángulo (ver figura 4.1) y envía a través del *socket* el punto objetivo al nodo *pr2_controller* y los comandos de teleoperación.

Para poder poner en funcionamiento el sistema implementado de forma ordenada debe seguir los siguientes pasos:

- Poner en marcha el nodo máster.

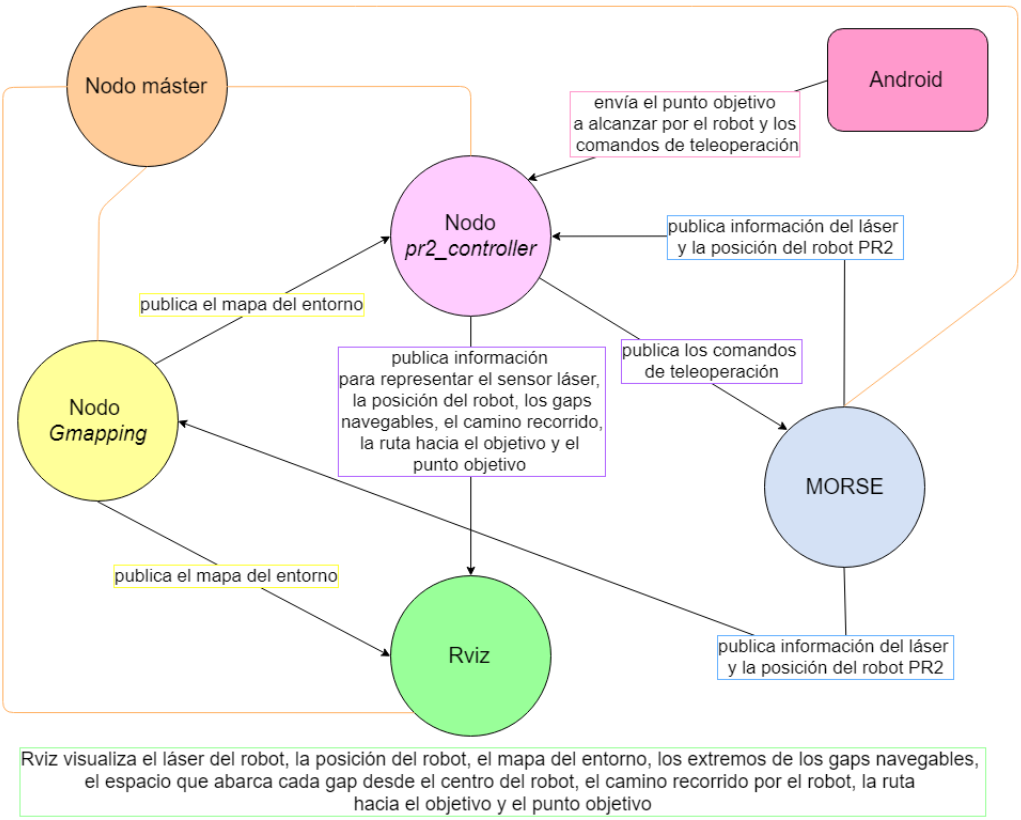



Figura 4.1: Principales componentes del sistema.

- Poner en marcha el simulador MORSE, encargado de publicar información sobre el sensor láser y la posición del robot.
- Poner en marcha nuestro nodo *pr2_controller*.
- Poner en marcha el paquete *Gmapping*, encargado de publicar el mapa del entorno por el que se mueve el robot.
- Poner en marcha la herramienta Rviz y añadir los tópicos que se desean visualizar.

El funcionamiento global del sistema queda descrito por el diagrama de flujo de la figura 4.2.

## 4.2 Herramienta de teleoperación propuesta

La herramienta de ayuda a la teleoperación a distancia de robots móviles se compone por todos los elementos anteriormente detallados (en los capítulos 2 y 3) que se visualizan en el *software* Rviz. En la figura 4.3 se muestra un ejemplo de un entorno representado con todos los componentes que forman el sistema.

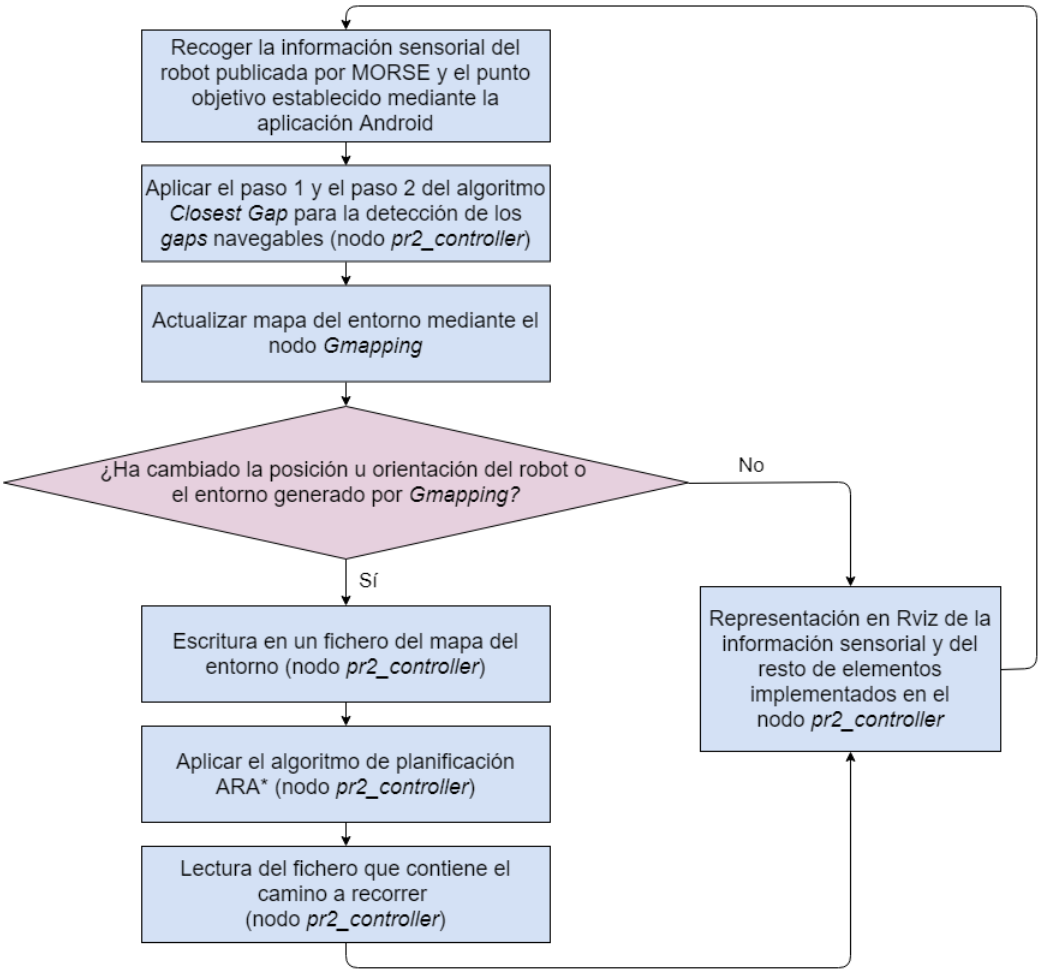



Figura 4.2: Diagrama de flujo del proceso que lleva a cabo el sistema.

## 4.3 Detalles del nodo *pr2_controller*

El Trabajo Final de Grado previo [1] se estructuraba de forma que se ejecutaba continuamente una función llamada *Server()*. Esta función era la encargada de recibir los comandos de teleoperación del cliente (aplicación Android) y de convertir esos comandos en órdenes de movimiento dirigidas al robot.

Al nodo ROS original propuesto en [1] (*pr2_controller*) se le han añadido varias funciones: el análisis de *gaps* del algoritmo *Closest Gap*, permitir al teleoperador establecer un punto objetivo y encontrar una ruta hacia ese punto mediante un algortimo de planificación, mapear el entorno mediante el paquete *Gmapping* y, por último, visualizar en Rviz todos estos elementos para facilitar la teleoperación y no depender del simulador MORSE (ver apéndice A).

Como se ha explicado en el capítulo 1, la comunicación en ROS se basa en publicadores y suscriptores de tópicos. Para ello, el nodo *pr2_controller* debe inicializar en el *main* todos los *publishers* y *subscribers*, asociándolos al tópico correspondiente. El nodo *pr2_controller* se compone por los siguientes:

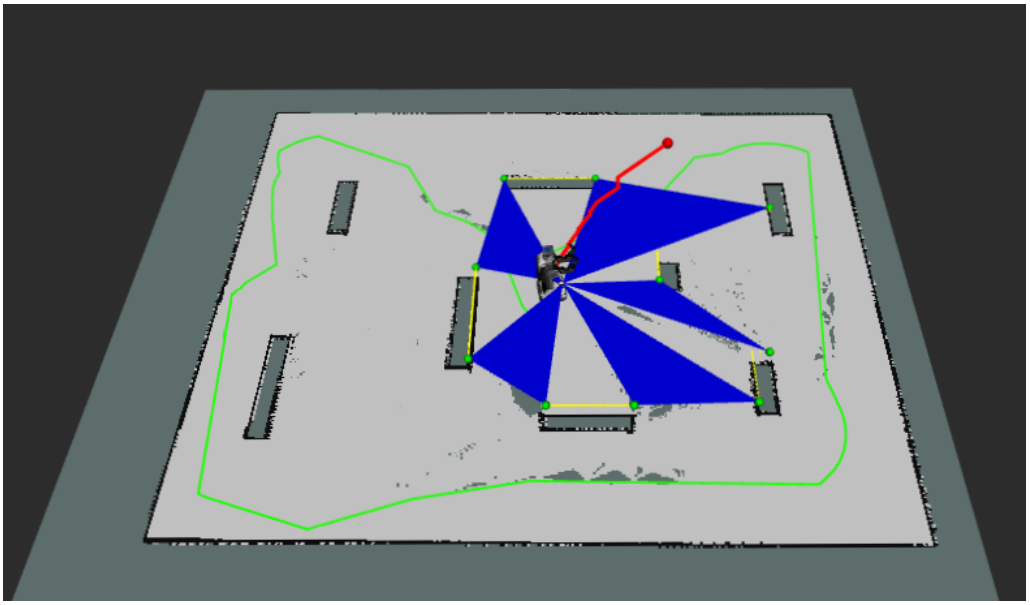


Figura 4.3: Herramienta de ayuda desarrollada.

- *Publishers*:
  - *velocity_publisher*, publica la velocidad para que el robot rote y se desplace. Este publicador se creó en [1].
  - *triangle_publisher*, publica el espacio que abarcan los *gaps* hasta el centro del robot mediante un marcador para, posteriormente, visualizarlo en Rviz. Se publica mediante el tópico */triangles*.
  - *gaps_publisher*, publica las esferas que representan los extremos de los *gaps* navegables mediante un marcador para, posteriormente, visualizarlos en Rviz. Se publica mediante el tópico */gaps*.
  - *path_publisher*, publica el camino recorrido por el robot desde su localización inicial. Se publica mediante el tópico */path*.
  - *cloud_laser_scan_publisher*, publica la proyección de la información proporcionada por el sensor láser. Se publica mediante el tópico */laser_scan*.
  - *goal_publisher*, publica el punto objetivo a alcanzar mediante un marcador. Se publica mediante el tópico */robot_goal*.
  - *line_strip_publisher*, publica el camino generado por el algoritmo de planificación AD* para, posteriormente, representarlo en Rviz. Se publica mediante el tópico */line_strip*.
- *Subscribers*:
  - *pose_subscriber*, se suscribe al tópico */odom* generado por MORSE que contiene la posición del robot.
  - *laser_subscriber*, se suscribe al tópico */base_scan* generado por MORSE que contiene las medidas de distancia tomadas por el sensor láser.

– *grid_subscriber*, se suscribe al tópico */map* generado por el nodo *Gmapping* que contiene el mapa del entorno por el que navega el robot.

Para el caso de los suscriptores, debemos crear un *callback* que asigne los datos publicados en el tópico correspondiente a una variable y se actualice.

A continuación, se listan las funciones implementados en el nodo *pr2_controller*.

### 4.3.1 Funciones de suscripción: *callbacks*

En ROS, para llamar a los *callbacks* se invoca la función *ros::spinOnce()*. Esta función se encarga de consultar y actualizar los valores del sensor láser, de la posición y del mapa por el que navega el robot. En concreto, se han definido los tres siguientes *callbacks*:

- *LaserCallback*, como se ha detallado en la sección 3.1, este *callback* se encarga de actualizar y guardar los datos proporcionados por el sensor láser en la variable *laser*. Además, mediante la clase *LaserProjection* se proyecta cada rayo del láser en un vector que, posteriormente, se utilizará para su visualización (explicado en la sección 3.4.1). Se trata del *callback* del suscriptor *laser_subscriber*.

- *PoseCallback*, *callback* del suscriptor *pose_subscriber*, se encarga de actualizar y guardar en la variable *car_pose* la información sobre la posición del robot obtenido mediante odometría. Esta información se guarda también en una variable auxiliar tal y como se ha explicado en la sección 3.4.2.

- *GridCallback*, *callback* del suscriptor *grid_subscriber*, actualiza y guarda en la variable *grid* el mapa construido por el nodo *Gmapping* del entorno que rodea al robot.

### 4.3.2 Funciones implementadas directamente vinculadas con el algortimo *Closest Gap*

Para poder aplicar el análisis de *gaps* del algoritmo CG, como se ha mencionado en la sección 3.3, se llevan a cabo dos pasos. Estos dos pasos se han implementado en dos funciones diferentes: *struct infoScanner* ***scanner_step1****(typeScanner scan)* y *struct infoScanner* ***scanner_step2*** *(struct infoScanner scanner_forward, struct infoScanner scanner_backward)*.

El paso 1 proporciona al paso 2 la información necesaria para poder eliminar aquellos *gaps* que se encuentren dentro de otros. Esa información se facilita a través de una estructura de datos llamada *infoScanner*, la cual contiene la siguiente información:

- Dos *arrays* llamados *points_i* y *points_j* con los índices en los que se encuentran los extremos de cada *gap* con respecto a la estructura de datos *scan*.

- Dos *arrays* llamados *dist_i* y *dist_j* que contienen las distancias a las que se encuentra el primer y segundo extremo de cada *gap* al centro del robot.

- Un *array* llamado *dist_gap* que contiene el ancho de los *gaps*.

- Un entero llamado *num_gap* que se corresponde con el número total de *gaps* encontrados.

- Un entero llamado *n_scans* que se corresponde con el número total de medidas tomadas por el sensor láser.
- Dos *arrays* llamados *index_gaps_i* e *index_gaps_j* que contienen los índices en los que se encuentran los extremos de cada *gap* con respecto de la estructura de datos *scan* proyectada. La diferencia entre los *arrays index_gaps_i*/*index_gaps_j* y *points_i*/*points_j* es que los primeros únicamente consideran las medidas del sensor láser en las que se ha detectado algún obstáculo. Estos índices se necesitan debido a que la proyección de la información del sensor láser solo incluye los rayos que detectan obstáculo.

Por otro lado, debido a que realizamos dos búsquedas, búsqueda *forward* y búsqueda *backward*, debemos saber de qué búsqueda se trata. Se ha creado un tipo enumerado llamado *typeScanner* que identifica en qué tipo de búsqueda nos encontramos.

La función del paso 1 tiene como parámetro de entrada el tipo de búsqueda que debe realizar (*forward* o *backward*) y devuelve una estructura de datos con los *gaps* encontrados en dicha búsqueda realizada. De esta forma, se llamará dos veces a esta función, una para la búsqueda *forward* y otra para la búsqueda *backward*.

La función del paso 2 tiene dos parámetros de entrada: en el primero se guardan los *gaps* encontrados en la búsqueda *forward* y en el segundo los *gaps* encontrados en la búsqueda *backward*. Esta función devuelve también una estructura de datos que corresponde con la lista definitiva de gaps navegables después de eliminar aquellos que eran innecesarios.

Para poder entender mejor el procedimiento seguido en los pasos 1 y 2 del análisis de *gaps*, las figuras 4.4 y 4.5 describen en pseudocódigo dichos pasos.

#### Funciones auxiliares

Para poder convertir ángulos de grados a radianes y viceversa, se han creado dos funciones:

- *double* ***degrees2radians****(double angle_in_degrees)*, convierte el ángulo de entrada de grados a radianes y lo devuelve en formato *double*.
- *double* ***radians2degrees****(double angle_in_radians)*, convierte el ángulo de entrada de radianes a grados y lo devuelve en formato *double*.

### 4.3.3 Funciones implementadas para generar un camino al objetivo

Para generar el camino a un punto objetivo se han implementado dos funciones: *void* ***save_environment****()* y *void* ***sbpl_path****()*. La primera de ellas se encarga de crear un fichero, llamado *environment.cfg*, con el mapa del entorno por el que navega el robot (para más información, véase la sección 2.4.1), y la segunda aplica el algoritmo de planificación AD* y lee la trayectoria generada por este algortimo desde un fichero llamado *sol.txt* (como se ha explicado en las secciones 2.4.2 y 2.4.3).

### 4.3.4 Función implementada para la visualización en Rviz

Para visualizar en la herramienta Rviz los elementos deseados, se deben publicar mensajes que contengan la información a mostrar. Estos mensajes se publican mediante los *publishers* en los tópicos anteriormente mencionados.

**PASO 1 CG**

Declarar e inicializar variables locales

Inicializar el índice para que apunte a la primera medida proporcionada por el sensor láser y el signo de giro en función del tipo de búsqueda, *forward* o *backward*

```
for (número total de scans) {
    calcular angle: cantidad de medidas del sensor láser que quedan por procesar
    if (discontinuidad de tipo 1) {
        for (angle) {
            buscar la distancia más corta al primer extremo del gap encontrado, corresponde
            con el segundo extremo del gap
            guardar gap encontrado
        }
    } else if (discontinuidad de tipo 2) {
        for (angle) {
            buscar primer punto con rango de detección máximo, corresponde con el segundo
            extremo del gap
            guardar gap encontrado
        }
    }
}
```

Guardar dentro de la estructura de datos *infoScanner* la información del *gap* en función del tipo de búsqueda

Figura 4.4: Pseudocódigo del paso 1 del análisis de *gaps* del algoritmo CG.

Se ha creado una función llamada *void* ***visualization_rviz*** *(struct infoScanner scanner_gaps)* que recibe como entrada la estructura de datos con la información de los *gaps* navegables. Esta función se encarga de publicar los marcadores que se quieren visualizar en Rviz. Una vez publicados, en Rviz debemos añadir los tópicos que corresponden a estos marcadores. Los detalles de cada elemento añadido ya se han detallado en las secciones 2.5 y 3.4.

Para determinar las características de cada marcador se deben especificar los parámetros que lo componen como: la dimensión, el color, la posición del marcador en coordenadas (x, y, z), la acción que se desea hacer (añadir, eliminar o modificar el marcador), etc.

Cabe destacar que, como primer paso antes de actualizar la información de marcadores, la función *visualization_rviz* elimina los marcadores antiguos para evitar que se muestren elementos que ya no existen en nuestro entorno.

***PASO 2 CG***

Declarar e inicializar variables locales

for (número de *gaps* encontrados en la búsqueda *forward*) {

    guardar el *gap* si su anchura es superior al diámetro del robot

}

for (número de *gaps* encontrados en la búsqueda *backward*) {

    guardar el *gap* si su anchura es superior al diámetro del robot

}

for (número total de *gaps* con anchura superior al diámetro del robot) {

    eliminar aquellos *gaps* que tengan algún extremo que coincida con el de otro *gap* ya encontrado

}

Guardar dentro de la estructura de datos *infoScanner* todos los *gaps* navegables

Figura 4.5: Pseudocódigo del paso 2 del análisis de *gaps* del algoritmo CG.

CAPÍTULO 5

# RESULTADOS EXPERIMENTALES

Los resultados experimentales se han dividido entre los entornos que sirven para justificar cada elemento visualizado en Rviz y los entornos en los que se lleva a cabo una navegación completa. No obstante, en primer lugar, se va a introducir el *software Blender* utilizado para el diseño de los entornos 3D simulados. También se explicará el simulador MORSE y, concretamente, la forma como este simulador nos permite definir los sensores y actuadores del robot utilizado. Finalmente, se hablará de la herramienta Rviz.

## 5.1 Sobre *Blender*

Este *software* es una herramienta clave para modelar entornos tridimensionales. El diseño se ha llevado a cabo añadiendo cubos 3D y modificando sus dimensiones. Además, se han añadido luces y 5 planos para delimitar el espacio por el que se puede mover el robot (el terreno y las paredes que delimitan ese terreno). En la figura 5.1 se puede observar un entorno con varios cubos, luces y planos (esos elementos están definidos en la segunda columna empezando por la derecha).

## 5.2 Sobre MORSE

MORSE es una plataforma de simulación robótica que permite la visualización realista de entornos diseñados previamente en *Blender*. Las escenas de simulación se generan a partir de *scripts* en Python (ver apéndice B) donde se definen el entorno *Blender* a visualizar, el robot utilizado y los sensores y actuadores que componen el robot. Además, se ha añadido el control del robot mediante el teclado. En la figura 5.2 se puede observar la simulación en MORSE del entorno de la figura 5.1.

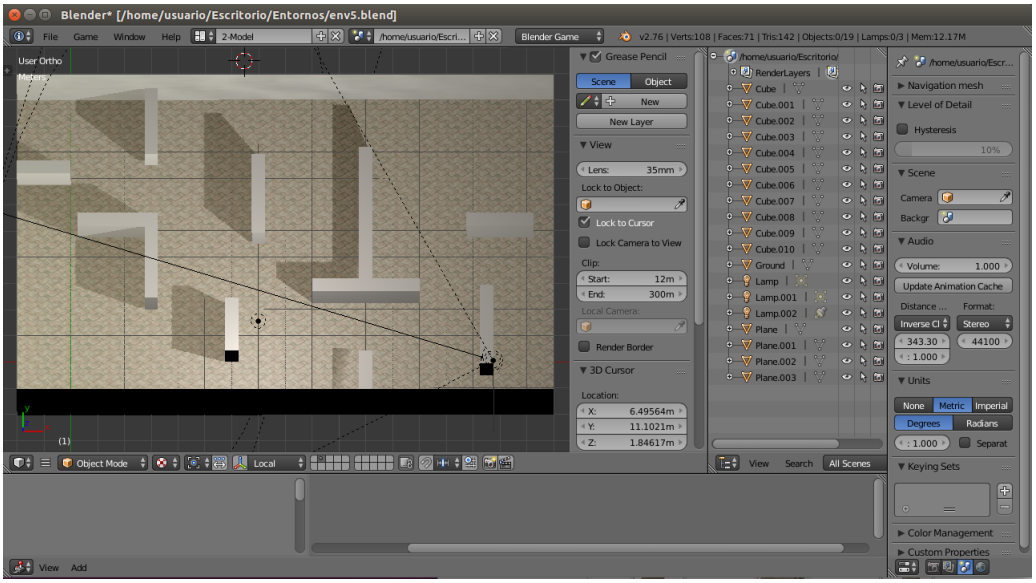



Figura 5.1: Entorno con obstáculos diseñado en Blender.

Figura 5.2: Simulación en MORSE del entorno de la figura 5.1.

## 5.3 Sobre Rviz

La herramienta de visualización 3D utilizada para ayudar a la teleoperación a distancia de robots móviles es Rviz.

Cuando se utiliza Rviz por primera vez, se visualiza una pantalla negra que corresponde con la vista 3D vacía porque aún no se ha añadido ningún elemento. Para que Rviz visualice la información de un tópico, se debe hacer clic en el botón *Add* de la parte inferior izquierda de la pantalla (ver figura 5.3). Los tópicos añadidos se listan en la columna de la izquierda y se puede elegir visualizarlo o no pulsando sobre el *checkbox* que tiene a su derecha.

Cabe destacar que también se pueden representar los sistemas de coordenadas del robot con las transformaciones pertinentes al sistema de coordenadas mundo. No obstante, se ha decidido no visualizar ningún sistema de coordenadas ya que, a veces, tapaban otros elementos cuya visualización era más relevante desde el punto de vista de la navegación.

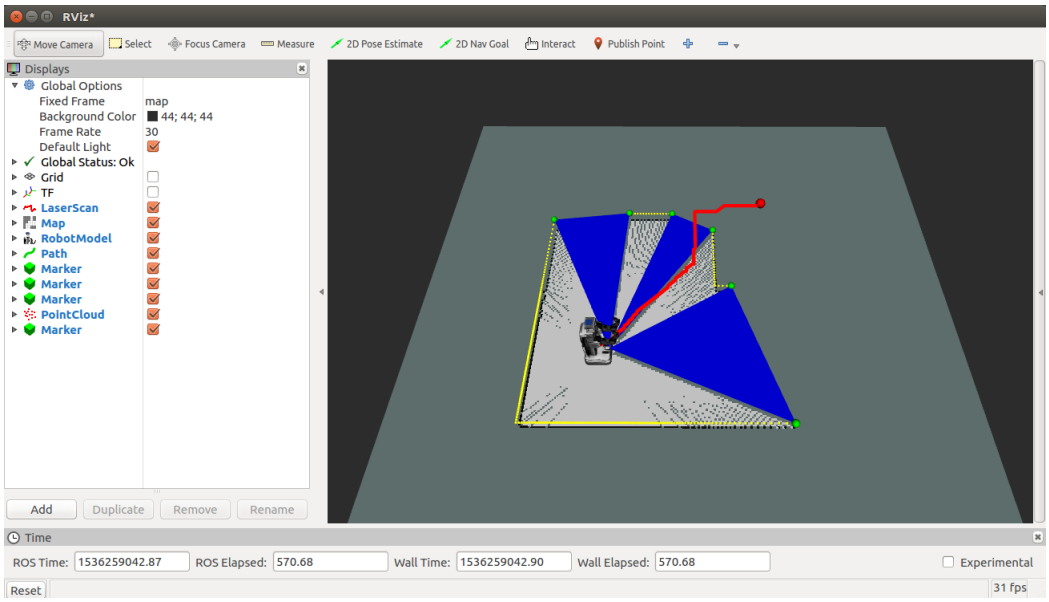



Figura 5.3: Aspecto general de la herramienta Rviz. En la parte izquierda de la pantalla se muestran los tópicos añadidos, y en la parte derecha tenemos la visualización 3D de la información publicada en dichos tópicos.

## 5.4 Elementos visualizados en Rviz

A continuación, se muestra la utilidad que puede tener cada uno de los elementos añadidos al *software* Rviz. Es importante recordar que la visualización de estos elementos tiene por objetivo eliminar la necesidad de mirar el simulador MORSE o el propio entorno del robot a la hora de teleoperar.

### 5.4.1 Mapa del entorno, posición y orientación del robot y el punto objetivo

Tres elementos claves para poder controlar a distancia el robot móvil dentro de un entorno son: la creación de un mapa a medida que el robot se desplaza, saber en qué posición y orientación está el robot, y conocer el punto que desea alcanzar. Por esta razón, se ha considerado imprescindible incorporar en Rviz los siguientes elementos:

- Un mapa global del entorno, generado por el nodo *Gmapping*, para poder conocer los obstáculos que rodean al robot y así poder navegar sin colisiones.

- El modelado del robot, para conocer la posición y orientación del robot, así como las dimensiones de éste.

- El punto objetivo, para saber el destino que quiere alcanzar el robot.

En la figura 5.4 se puede observar una representación de estos tres elementos en el *software* Rviz.

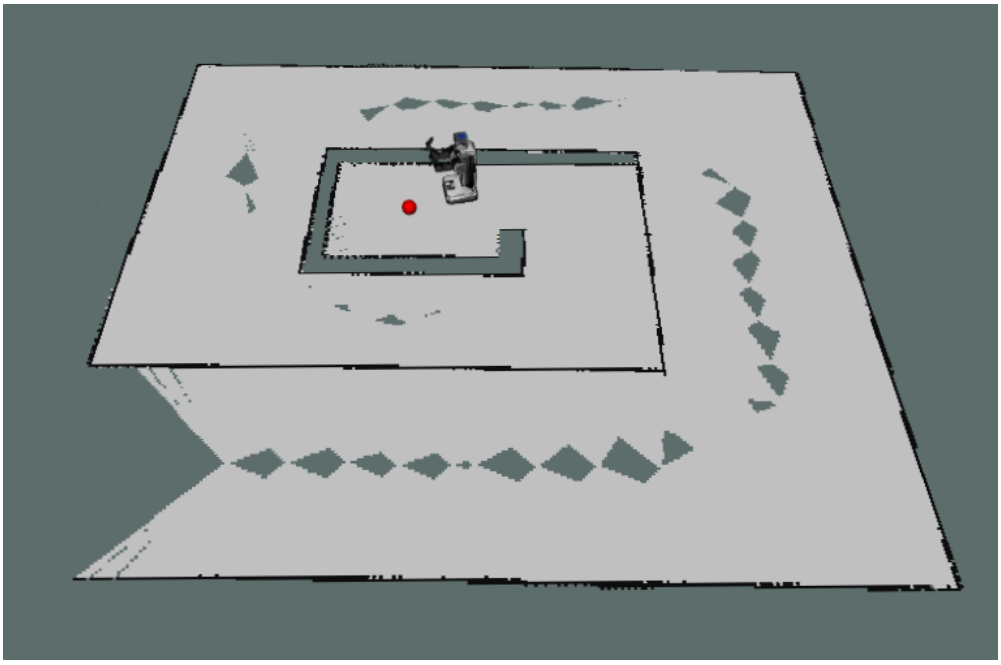


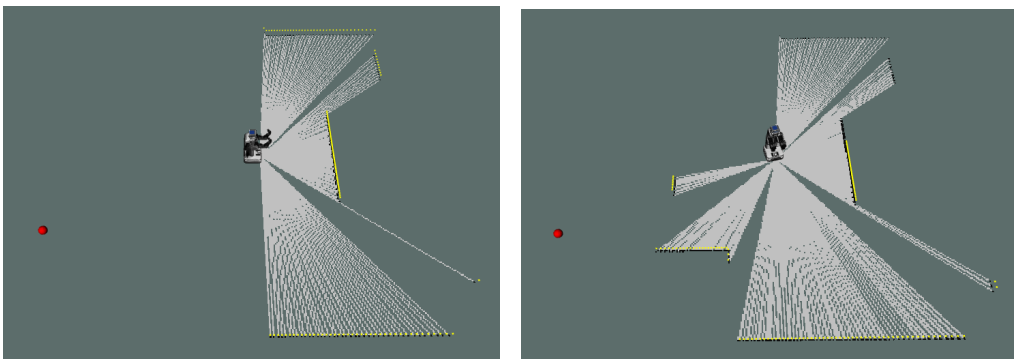
Figura 5.4: Representación en Rviz del mapa global del entorno, el modelo del robot y el punto objetivo.

### 5.4.2 Sensor láser

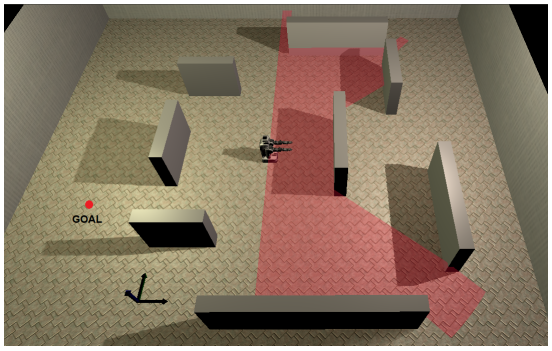
Además de visualizar el entorno global, tener constancia de lo que observa el robot en cada momento permite al teleoperador conocer los grados que debe girar al robot para visualizar cierto punto del espacio, como por ejemplo el punto objetivo. Esto permite crear un mapeado local mediante la información que proporciona el sensor láser en el parámetro rangos. En la figura 5.5 podemos observar que necesitamos girar al robot aproximadamente 90 grados (figura 5.5b) para que el láser detecte si hay camino directo o no al punto objetivo.

### 5.4.3 Camino recorrido por el robot

Conocer el camino que ha recorrido el robot desde el punto de partida puede parecer información irrelevante, pero si nos fijamos en la figura 5.6 se puede observar su importancia. Esto se debe a que inicialmente (ver figura 5.6a) parece que disponemos de tres caminos libres de obstáculos hacia el punto objetivo, por lo que escogemos aquel camino que parece que nos permitirá alcanzar el punto objetivo de forma más directa. sin embargo, resulta que ese camino nos devuelve a la posición inicial (figura 5.6d) eliminando el otro espacio que parecía disponible. Gracias a esta información, el usuario no volverá a dirigirse a



(a) (b)




(c) Escenario en MORSE.

Figura 5.5: Mapa local del entorno visualizado por el robot sobre un escenario con obstáculos en el *software* Rviz.

ninguno de esos dos caminos y no quedará en bucle, por lo que solo dispone de la otra alternativa.

### 5.4.4 *Gaps* navegables

Es cierto que disponer de un mapa global permite al usuario reconocer los espacios por los que el robot puede moverse. Si nos fijamos en la figura 5.7a, parece lógico que la mejor opción para alcanzar el punto objetivo será ir directamente por el espacio que se encuentra junto en frente del punto objetivo. Pero el hecho de aplicar el análisis de *gaps* del algortimo *Closest Gap*, nos permite reconocer antes de realizar ningún movimiento que este espacio no será navegable para nuestro robot, como muestra la figura 5.7b. Ahora solo nos quedará decidir cual de los dos espacios navegables elegir para alcanzar el punto objetivo.

### 5.4.5 Ruta hacia el objetivo

Otra duda que genera el movimiento por entornos con muchas rutas hasta un objetivo, es decidir el camino a tomar. Esto se debe a que en escenarios con obstáculos equivalentes a ambos lados, es prácticamente imposible que el ojo humano detecte a simple vista el camino óptimo a seguir (ver figura 5.10). Por esta razón, se ha aplicado un algoritmo

de búsqueda que genera un camino inmediatamente óptimo que deberíamos recorrer si deseamos alcanzar dicho objetivo con mayor rapidez. Si observamos la figura 5.10, resulta complicado escoger entre dirigirse hacia la derecha o hacia la izquierda, mientras que al aplicar este algoritmo, él se encarga de calcular cuál es el más corto (ver figura 5.8). Si nos dirigiésemos hacia la izquierda, se puede observar en la figura 5.9 que el recorrido sería ligeramente más largo debido a que tiene que esquivar el objeto que se encuentra al desviarse hacia abajo.

## 5.5 Entornos

A continuación, se muestran los resultados obtenidos en Rviz tras teleoperar al robot en varios entornos simulados en MORSE.

### 5.5.1 Entornos simples

Por entorno simple se consideran aquellos entornos con pocos obstáculos y con una distancia a recorrer hasta el punto objetivo corta.

#### Entorno 1

En este primer entorno (ver figura 5.11), el robot dispone de un sensor láser con un rango máximo de detección de 6 metros y una cobertura de 360 grados. En el instante mostrado en la figura 5.11a resultaría complicado escoger uno de los dos espacios navegables. Sin embargo, gracias al uso de la librería SBPL que nos proporciona el camino más corto a seguir hasta el punto objetivo, era esa decisión se convierte en algo trivial.

#### Entorno 2

Para este segundo entorno (ver figura 5.12), el sensor láser tiene un rango máximo de detección de 6 metros y 270 grados de cobertura. Una vez recorrido todo el espacio navegable, volvemos a la posición inicial. El algoritmo de planificación AD* permite determinar que no hay camino hasta el punto objetivo debido a que éste se encuentra dentro de un obstáculo. En la figura 5.12a, se puede observar que el camino generado por el algoritmo de planificación empieza con un movimiento oscilante. Esto se debe a que las celdas inexploradas suponen un coste superior a las celdas libres de obstáculo. En consecuencia, la ruta al punto objetivo evita en la medida de lo posible las celdas grises moviéndose por las blancas.

### 5.5.2 Entornos complejos

Por entorno complejo se consideran aquellos entornos con mayor número de obstáculos, con mayor distancia a recorrer hasta el punto objetivo y con escenarios que obligan al algoritmo de planificación la reconducción por otros caminos.

#### Entorno 3

En el primer entorno complejo (ver figura 5.13), el robot debe recorrer una distancia de más de 20 metros para poder alcanzar el punto objetivo que se encuentra en el otro extremo.

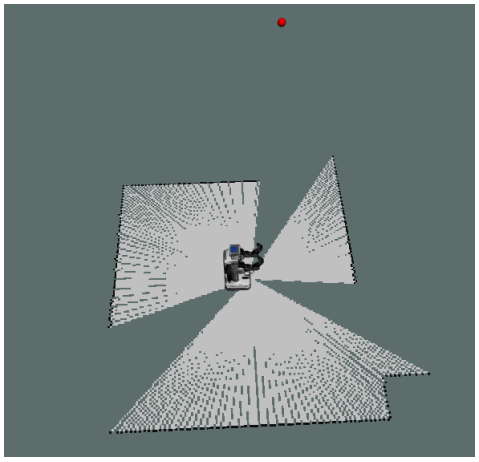
Además, a lo largo del camino, el robot deberá sortear múltiples obstáculos/paredes. En esta ocasión, el robot posee un sensor láser con un rango máximo de detección de 5 metros y una cobertura de 270 grados.

### Entorno 4

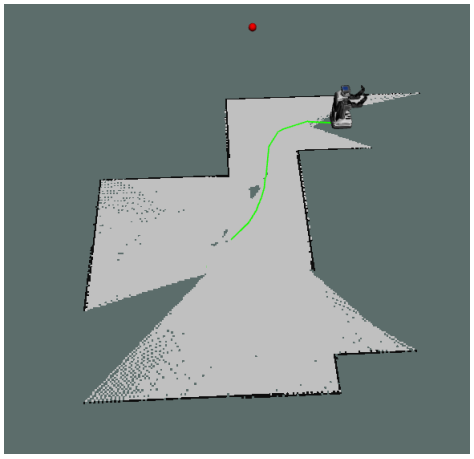
Este entorno (ver figura 5.14) es similar al anterior con la diferencia de que ahora hay obstáculos/paredes con una forma más compleja (tipo zig-zag) y una longitud mayor. Eso hace que el robot tarde en detectar completamente ciertos obstáculos y, por tanto, provoca que se produzcan cambios importantes del camino a recorrer (línea roja) en intervalos cortos de tiempo. Uno de esos cambios ocurre entro los instantes e) y f) (ver figuras 5.14e y 5.14f). En esta ocasión, el robot posee un sensor láser con un rango máximo de detección de 5 metros y una cobertura de 270 grados.

### Entorno 5

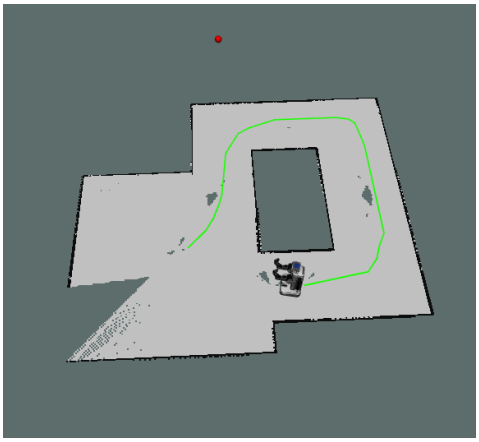
Para acabar, este entorno (ver figura 5.15) pretende mover al robot por un laberinto de obstáculos. No es hasta la figura 5.15d que el robot detecta que el pasillo más próximo al punto objetivo no tiene salida, por lo que debe continuar por el otro extremo navegable. Esta vez sí, por el otro camino, podrá alcanzar el punto objetivo. Si se compara la figura 5.15g con la figura 5.15h se puede observar que el camino recorrido por el robot se ha desplazado ligeramente, pareciendo así que el robot ha colisionado con el obstáculo. Esto se debe a que el sistema de localización utilizado no es perfecto, por lo que acumula errores produciendo un desplazamiento de la posición del robot y en consecuencia, todo lo que está referenciado a él. En este entorno, el robot posee un sensor láser con un rango máximo de detección de 4 metros y una cobertura de 270 grados.



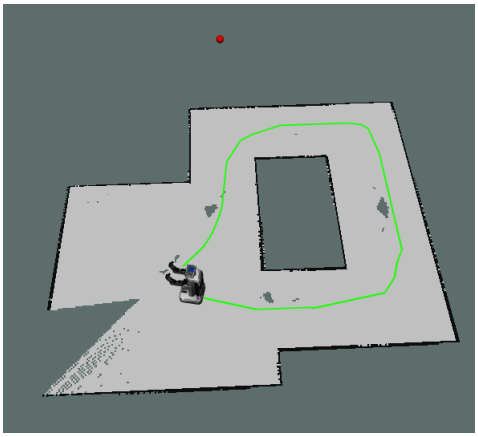
(a)



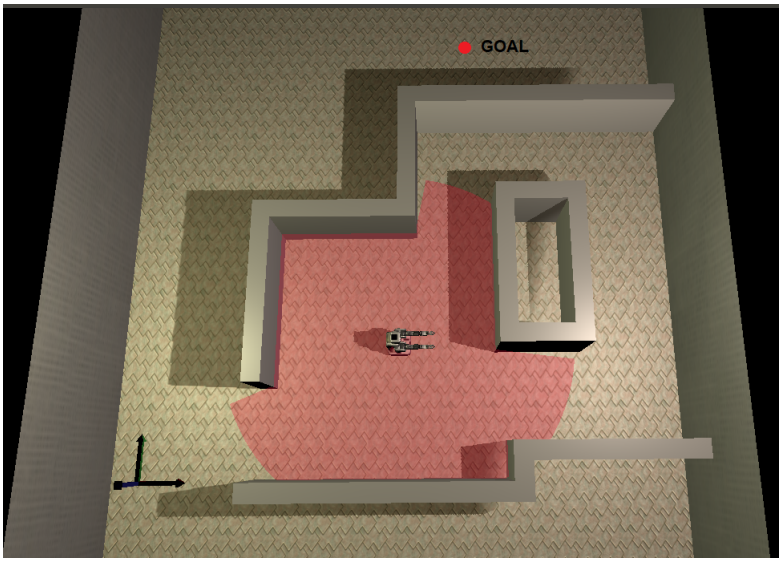
(b)



(c)



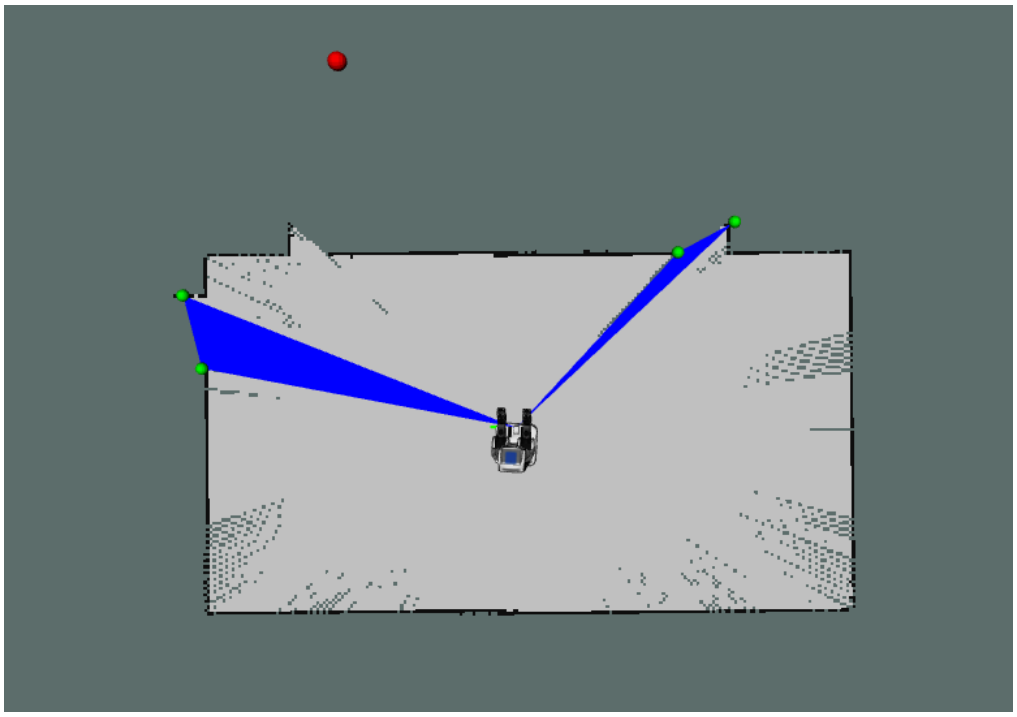
(d)




(e) Escenario en MORSE.

Figura 5.6: Representación del camino recorrido por el robot en Rviz.

(a) Entorno en MORSE.



(b) Entorno en Rviz.

Figura 5.7: Detección de espacios navegables al implementar el análisis de gaps del algoritmo *CG* en Rviz.

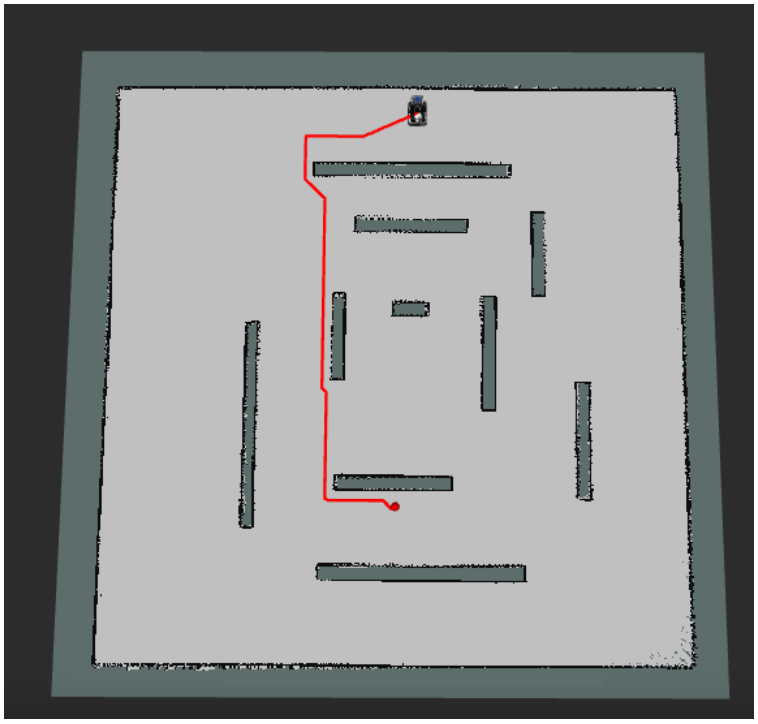


Figura 5.8: Camino hacia el punto objetivo del entorno 5.10 generado por el algoritmo de búsqueda AD* en Rviz.

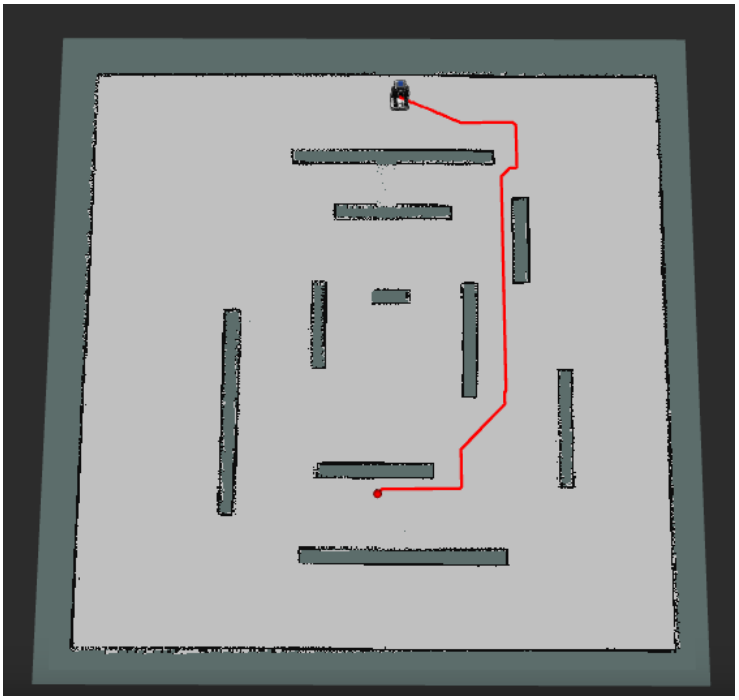


Figura 5.9: La otra ruta al punto objetivo del entorno 5.10 ligeramente más larga en Rviz.

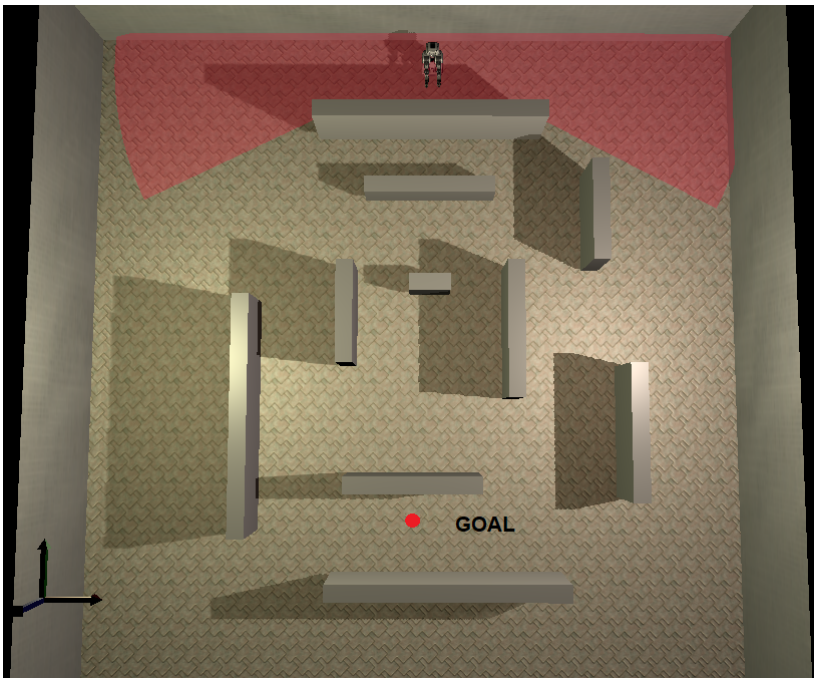
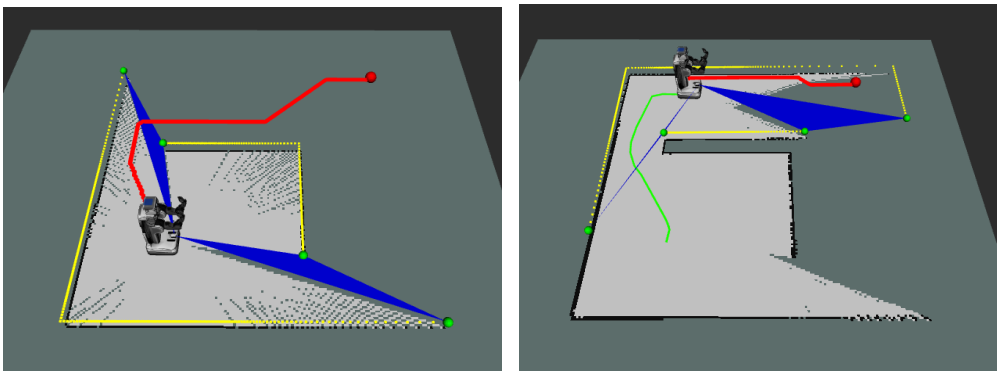
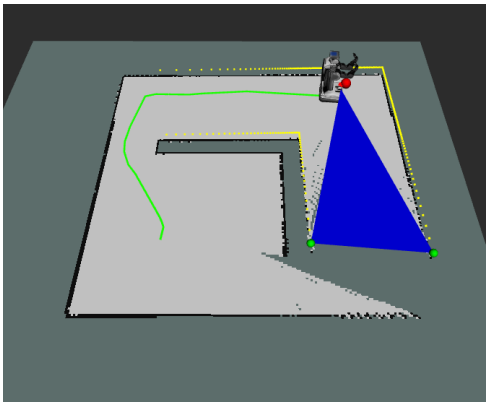



Figura 5.10: Vista del entorno de la figuras 5.8 y 5.9 en MORSE.



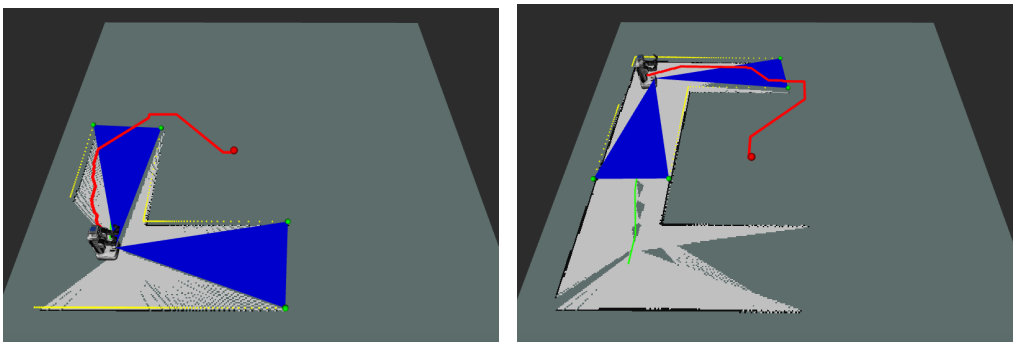
(a) (b)



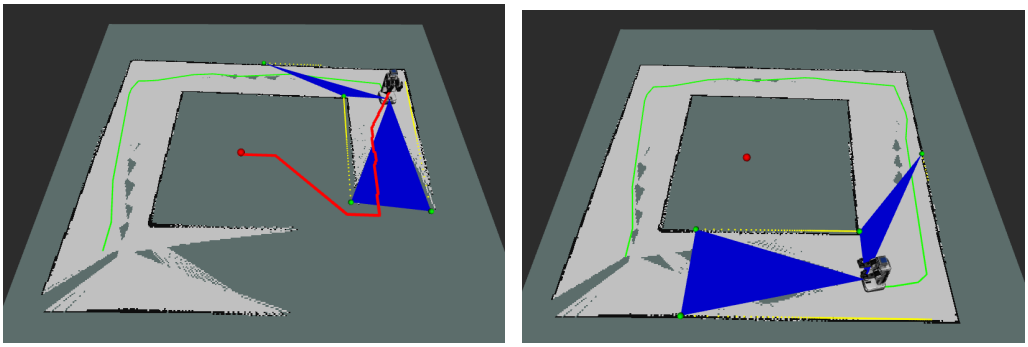
(c)

(d)

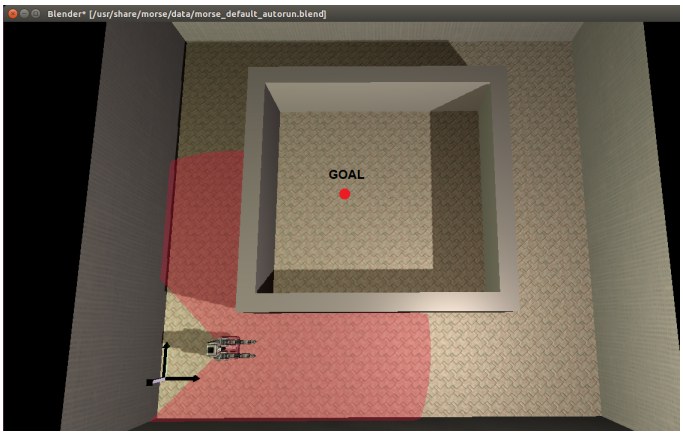
Figura 5.11: Proceso de teleoperación del robot en el entorno 1. (a, b, c) progreso del robot visualizado en Rviz. (d) vista del entorno tomada desde MORSE.



(a) (b)

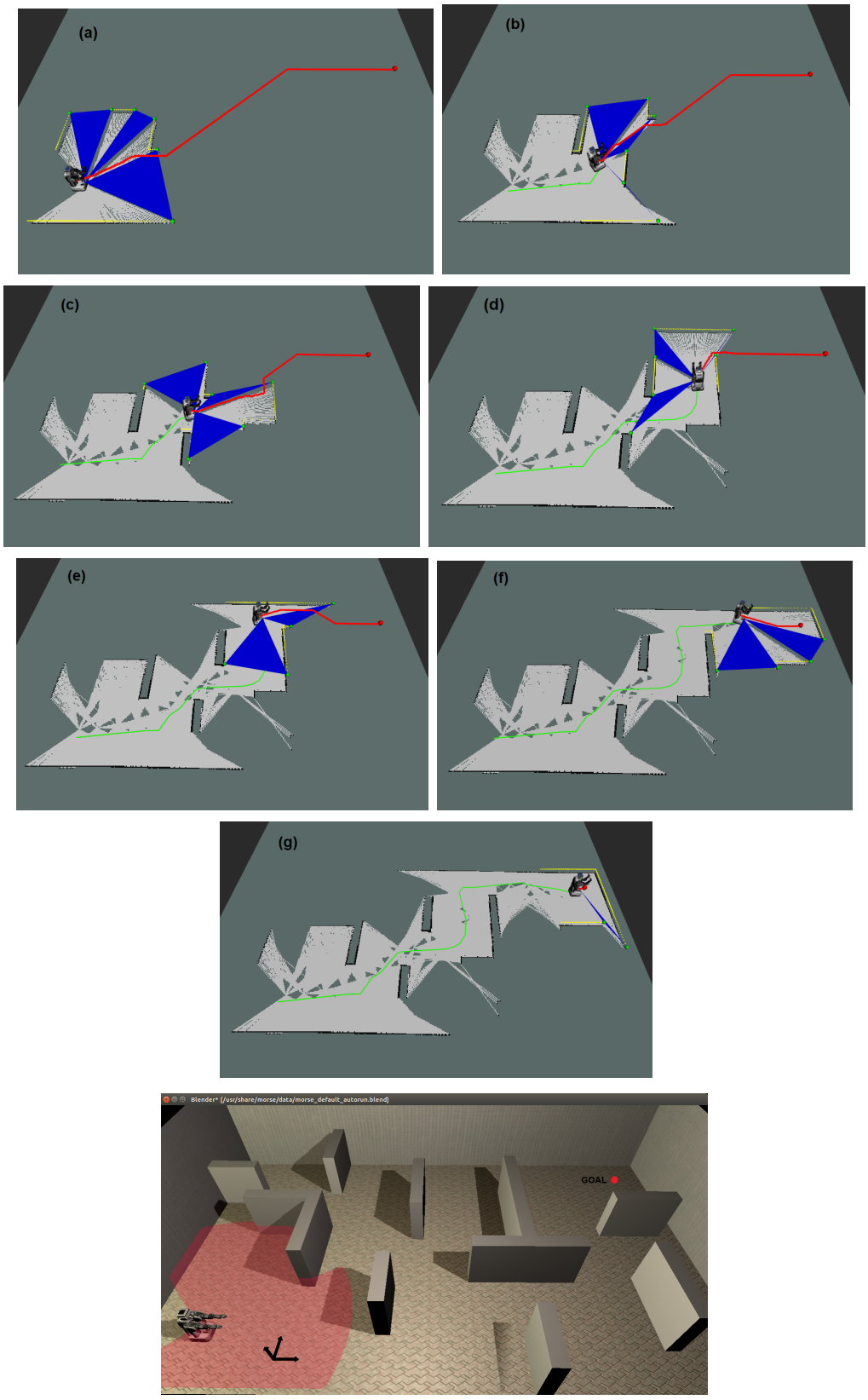


(c) (d)



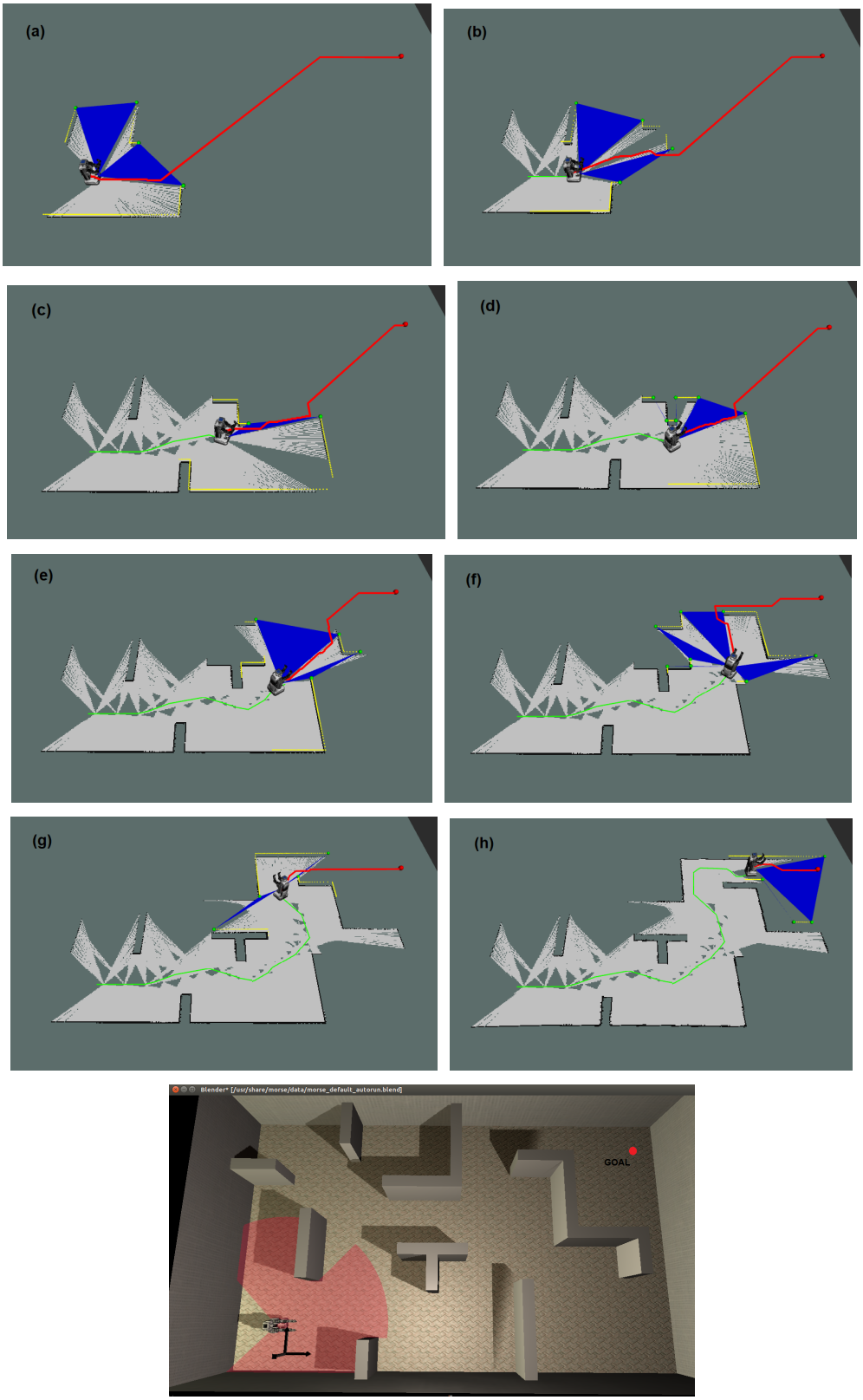

(e)

Figura 5.12: Proceso de teleoperación del robot en el entorno 2. (a, b, c, d) progreso del robot visualizado en Rviz. (e) vista del entorno tomada desde MORSE.



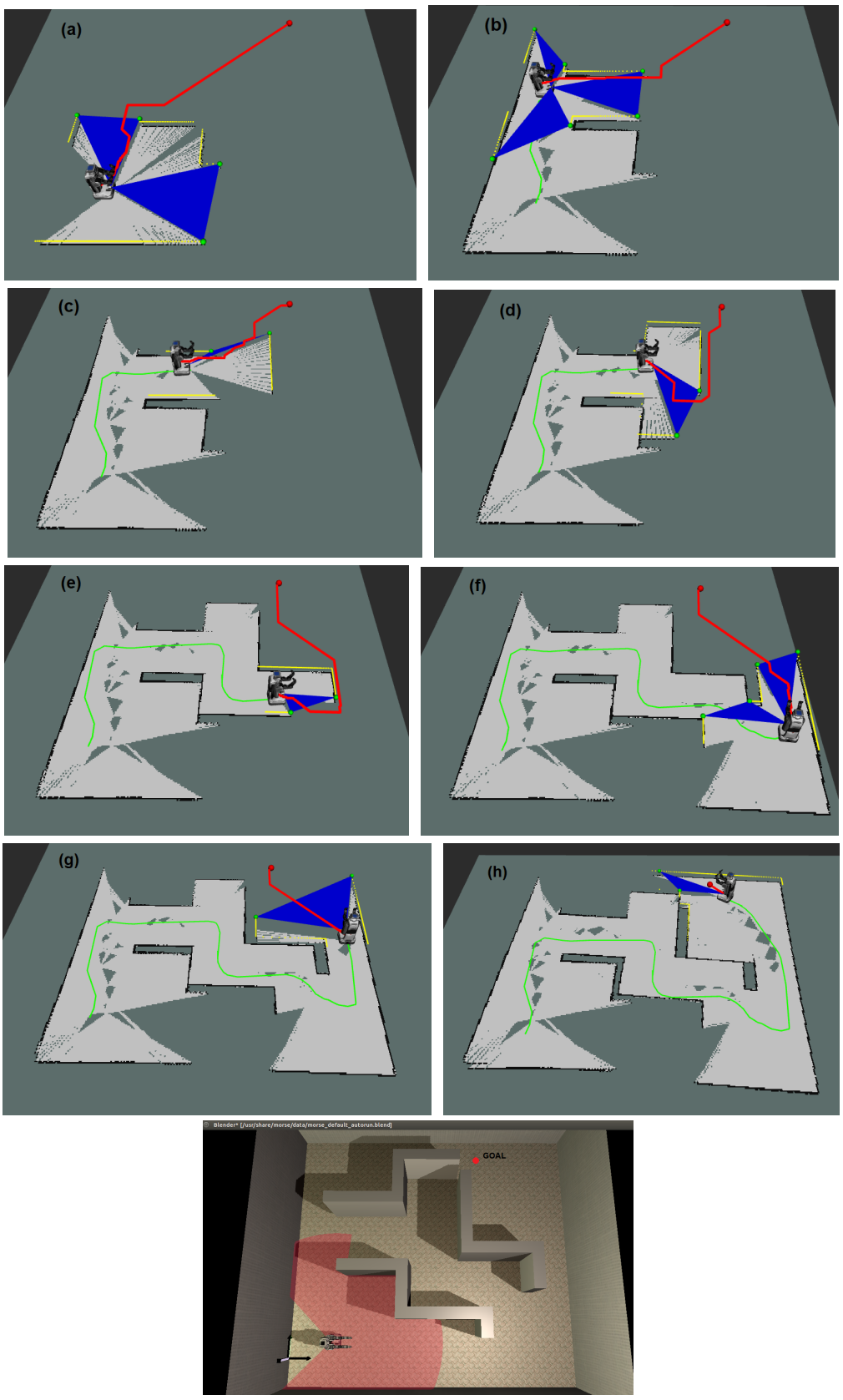

(h)

Figura 5.13: Proceso de teleoperación del robot en el entorno 3. (a, b, c, d, e, f, g) progreso del robot visualizado en Rviz. (h) vista del entorno tomada en MORSE.




(i)

Figura 5.14: Proceso de teleoperación del robot en el entorno 4. (a, b, c, d, e, f, g, h) progreso del robot visualizado en Rviz. (i) vista del entorno tomada en MORSE.




(i)

Figura 5.15: Proceso de teleoperación del robot en el entorno 5. (a, b, c, d, f, g, h) progreso del robot visualizado en Rviz. (i) vista del entorno tomada en MORSE.

CAPÍTULO 6

# CONCLUSIONES

Para concluir esta memoria se analizará la fase previa al desarrollo del TFG, la implementación realizada y se hará un análisis general de los resultados obtenidos. Finalmente, se enumeran los principales problemas encontrados en el transcurso de este proyecto.

## 6.1 Fase previa

Debido a la necesidad de utilizar un nuevo sistema operativo (Ubuntu), se tuvo que realizar una partición del disco duro para así alojar Ubuntu de forma paralela a Windows. Posteriormente, se llevó a cabo la instalación del sistema operativo ROS en el interior de Ubuntu. Pese a ser un sistema operativo, la instalación llevada a cabo es realmente sencilla gracias a los comandos que se pueden introducir a través de la consola. Por otro lado, Ubuntu exigió aprender y comprender todos los conceptos relacionados con ROS, Android y *sockets*.

También cabe mencionar que como primer simulador se utilizó el nodo *turtlesim*, el cual permite desarrollar simulaciones simples en entornos bidimensionales. Superada esa etapa, se estudió la aplicación Android realizada en el trabajo previo [1] y el nodo ya creado, con el objetivo de continuar ese proyecto y ampliarlo para proporcionarle funcionalidades nuevas.

## 6.2 Implementación

Esta fase comenzó con la implementación parcial del algoritmo *Closest Gap*. Para comprobar su correcto funcionamiento, se realizaron las primeras pruebas sobre el *software* Rviz. En este punto se tuvo que proyectar la información proporcionada por el sensor láser, la cual nos permitía identificar los extremos de cada *gap*. Además, pasamos de usar el robot ATRV a usar el robot PR2, por lo que muchos de los parámetros previos se tuvieron que modificar debido a que todo estaba configurado para el robot ATRV.

A medida que nos familiarizábamos con el *software* Rviz al visualizar la información del sensor láser y de los espacios navegables, se añadió el mapa global del entorno generado con el paquete *Gmapping*. Este paso parecía sencillo, pero, conseguir la visualización del mapa, requirió más tiempo del esperado.

El siguiente paso fue añadir la opción a la aplicación Android de poder fijar el punto objetivo. Es cierto que fue sencillo permitir que el usuario introdujera dos coordenadas en la pantalla de configuración, pero lograr entender todo el proceso que se debía seguir para enviar esta información a través de *sockets* y que, posteriormente, nuestro nodo ROS extrajera esta información, no fue tarea fácil.

Más tarde, se visualizó en Rviz el camino al punto objetivo que calculamos aplicando el algoritmo AD* de la librería SBPL. En esta fase, la tarea más complicada fue la escritura del fichero. Esto se debía a que el fichero debía ajustarse a un formato y unas dimensiones exactas (columnas y filas del entorno generado por el paquete *Gmappping*) o el programa dejaba de funcionar. Resultó costoso encontrar los fallos debido a la cantidad de datos que se acumulaban en ese fichero. En este punto se realizó también el proceso de agrandar los obstáculos.

Finalmente, añadir el modelo del robot PR2 en Rviz supuso muchos problemas. Primeramente, había problemas con las transformaciones. Arreglado este fallo, el modelo del robot PR2 sólo se visualizaba parcialmente. La solución fue añadir dentro de un archivo *launch* todos los nodos encargados de publicar el estado del robot y también la puesta en marcha de la herramienta Rviz. De esta forma conseguimos que se visualizara completamente el modelo del robot PR2.

## 6.3 Resultados obtenidos

Los resultados experimentales corroboran el buen funcionamiento del sistema implementado, consiguiendo así teleoperar un robot a distancia eliminando el visualizador de entorno MORSE. Por otro lado, también se ha conseguido implementar satisfactoriamente la fase de análisis de *gaps* del algoritmo *Closest Gap*. Con las diversas pruebas realizadas se muestran los espacios navegables ajustándose a la teoría del propio algoritmo.

Cabe decir que el algoritmo de planificación utilizado proporciona una ayuda indiscutible a la hora de escoger entre diversos caminos para recorrer. Esto permite reducir el tiempo que conlleva alcanzar un punto dentro de un entorno. Es cierto que no siempre es así porque no disponemos de un mapa previo del entorno. En cambio, si recorriéramos todo el entorno y eligiéramos un punto dentro de éste, podemos garantizar que la ruta generada por el algoritmo de planificación será la óptima.

Además, como ya se ha demostrado en el capítulo 5, el mapa generado por el nodo *Gmapping* es un elemento indispensable para teleoperar el robot y conocer el entorno que le rodea. Por otra parte, el modelo del robot y el camino recorrido por éste permiten al usuario conocer la posición y orientación del robot, así como la trayectoria recorrida desde el instante inicial.

## 6.4 Problemas encontrados

Inicialmente, el reto fue familiarizarse con el sistema operativo ROS debido a que no es fácil de manejar de antemano. Por esta razón, el primer paso fue realizar los tutoriales

introductorios de ROS [17]. Por otra parte, ampliar un trabajo ya realizado conlleva un estudio exhaustivo de todos los pasos realizados por otra persona. Una vez superado todo esto, se pudo empezar con el desarrollo de este TFG. A continuación, se mencionan los principales problemas que se han encontrado y se han ido solucionando durante el transcurso de este proyecto.

### 6.4.1 Transformaciones

Para trabajar con una herramienta como Rviz es necesario conocer la jerarquía de sistemas de coordenadas que toma como referencia. Cuando quieres visualizar un lemento en Rviz, debes indicar el sistema de coordenadas sobre el que dicho elemento se debe representar. Cualquier robot posee múltiples sistemas de coordenadas. Por ejemplo, el sistema de coordenadas de la base del robot, el sistema de coordenadas de cada uno de los sensores y actuadores del robot, etc. Se debe conocer la relación que existe entre estos sistemas de coordenadas del robot y un sistema de coordenadas externo y fijo denominado sistema de coordenadas mundo. O dicho con otras palabras, se debe poder conocer la transformación que nos permite pasar de cualquier sistema de coordenadas del robot al sistema de coordenadas mundo.

Inicialmente, se se simuló el robot terrestre ATRV, pero a medida que se avanzó en el proyecto y se introdujo el *software* Rviz empezaron a surgir una serie de problemas. Esto se debía a que el robot ATRV no proporcionaba las transformaciones de sus sistemas de coordenadas al sistema de coordenadas mundo, y resultó muy complicado establecer cada una de esas transformaciones de forma manual mediante el paquete tf [18]. Por eso, se decidió cambiar al robot terrestre PR2, que proporciona automáticamente las transformaciones de todos los sistemas de coordenadas que componen el robot al sistema de coordenadas mundo.

### 6.4.2 Información proporcionada por el sensor láser

El simulador MORSE publica un mensaje en donde envía el conjunto de medidas de distancia tomadas por el sensor láser. Para poder representar esta información en Rviz, nosotros transformábamos manualmente dicho conjunto de medidas desde el sistema de coordenadas del robot al sistema de coordenadas mundo. Sin embargo, esa transformación no se realizaba correctamente. Este problema se solucionó cuando utilizamos la clase *LaserProjection* para llevar a cabo la mencionada transformación (para más detalles, ver la sección 4.3).

### 6.4.3 Generación del camino al objetivo

Cuando aplicamos el algoritmo de planificación AD* para obtener un camino hacia el punto objetivo veíamos como el camino generado aparentemente atravesaba los obstáculos del entorno dirigiéndose hacia el destino de forma directa. Esto se debía a que nosotros suponíamos que el robot era un punto sin masa ni dimensiones. La solución consistió en agrandar los obstáculos como mínimo el radio del robot para así tener en cuenta sus dimensiones, como se detalla en la sección 2.4.1.

### 6.4.4 Documentación en la wikipedia de ROS

Para solucionar muchos de los problemas encontrados en el transcurso de este proyecto se ha tenido que acudir a foros, blogs, etc. Esto se ha debido a que en ROS falta documentación que explique ciertos detalles de funcionamiento de la mayoría de sus herramientas y paquetes. En consecuencia, se ha invertido más tiempo del esperado debido a que no todas las soluciones propuestas en estos foros arreglaban los problemas que iban surgiendo.

APÉNDICE A

# CÓDIGO *pr2_controller.cpp*

```
/*
* Author 1: David Arévalo Jiménez
* Year: 2017
* Trebajo Final de Grado EEIA
* University: Universitat de les Illes Balears (UIB)
*
*/

/*
* Author 2: Marta Barquiel Rossello
* Year: 2018
* Trebajo Final de Grado EEIA
* University: Universitat de les Illes Balears (UIB)
*
*/

#include "ros/ros.h"
#include "geometry_msgs/Twist.h"
#include "nav_msgs/Odometry.h"
#include "nav_msgs/OccupancyGrid.h"
#include "nav_msgs/Path.h"
#include "visualization_msgs/Marker.h"
#include "sensor_msgs/LaserScan.h"

#include "tf/transform_listener.h"
#include "tf/transform_broadcaster.h"
#include "sensor_msgs/PointCloud.h"
#include "tf/message_filter.h"
#include "message_filters/subscriber.h"
#include "laser_geometry/laser_geometry.h"
```

```
#include <iostream>
#include <vector>
#include <fstream>

#include <sstream>
#include <stdio.h>
#include <stdlib.h>

#include <sys/socket.h>
#include <sys/types.h>
#include <arpa/inet.h> //inet_addr
#include <unistd.h>    //write
#include <pthread.h> //for threading , link with lpthread
#include <cmath>
#include "/home/usuario/catkin_ws/src/sbpl/src/include/sbpl/
    headers.h" //Librería SBPL

using namespace std;

class MyEnvironment : public EnvironmentNAV2D{ //Modificación de
     la clase EnvironmentNAV2D de la librería SBPL
  public:

  void PrintState(int stateID, FILE* fOut, bool bVerbose)
  {
    EnvNAV2DHashEntry_t* HashEntry = EnvNAV2D.StateID2CoordTable
        [stateID];

    if (bVerbose) printf("X=%d Y=%d\n", HashEntry->X, HashEntry
        ->Y);

    fprintf(fOut, "%d %d\n", HashEntry->X, HashEntry->Y);
  }

};

//Global variables
ros::Publisher velocity_publisher; //Publisher velocidad del
    robot
ros::Subscriber pose_subscriber; //Subscriber poseCallback
nav_msgs::Odometry car_pose;
geometry_msgs::Twist vel_msg;
geometry_msgs::Twist vel_msg_stop;

nav_msgs::Path global_path; //Topic camino recorrido
geometry_msgs::PoseStamped pose_aux; //Topic duplicado de pose
sensor_msgs::LaserScan laser; //Laser callback
nav_msgs::OccupancyGrid grid; //Grid callback

sensor_msgs::PointCloud cloud_laser_scan; //PointCloud de la
    proyección de LaserScan
visualization_msgs::Marker marker_triangles; //Triángulos que
    unen los gaps navegables
```

```
visualization_msgs::Marker marker_gaps; //Esferas en los
   extremos de los gaps navegables
visualization_msgs::Marker marker_goal; //Esfera del punto
   objetivo
visualization_msgs::Marker marker_line_strip; //Line strip del
   camino óptimo que debería seguir el robot para llegar al
   objetivo


ros::Subscriber gaps_subscriber; //Subscriber laserCallback
ros::Subscriber grid_subscriber; //Subscriber gridCallback
ros::Publisher triangle_publisher; //Publisher triángulos gaps
   navegables
ros::Publisher path_publisher; //Publisher camino recorrido por
   el robot
ros::Publisher cloud_laser_scan_publisher; //Publisher duplicado
    LaserScan
ros::Publisher gaps_publisher; //Publisher esferas gaps
   navegables
ros::Publisher goal_publisher; //Publisher punto objetivo
ros::Publisher line_strip_publisher; //Publisher triángulos gaps
    navegables


double angulos;
double velocidades;

const double PI = 3.14159265359;
const double radio_PR2 = 0.5; //Radio del robot PR2
const int quantity_scans = 271;
int num_punts_ant=0;
int num_punts_ant2=0;
int mode=0,velMax=50,angularMax=50;

int num_obstacles; //Número de obstáculos detectados por el
   sensor láser
int f; //Rayo inicial del sensor láser

float x_goal; //Coordenada x del punto objetivo
float y_goal; //Coordenada y del punto objetivo

int num_points_line_strip = 0; //Número de celdas a seguir hasta
    el punto objetivo
int array_path_x[1000]; //Array con las coordenadas x del camino
    recorrido por el robot
int array_path_y[1000]; //Array con las coordenadas y del camino
    recorrido por el robot
int cont_path = 0; //Número de puntos recorridos por el robot
float x_ant = 0.0; //Posición anterior de la coordenada x del
   robot
float y_ant = 0.0; //Posición anterior de la coordenada y del
   robot
```

```
int field; //Ángulo que abarca el sensor láser
int number_scans; //Número de rayos totales del sensor láser

bool semaphore = false;
bool first_time = true;

enum typeScanner {forward, backward};

//Estructura
struct infoScanner {
  int points_i[quantity_scans];
  int points_j[quantity_scans];
  float dist_i[quantity_scans];
  float dist_j[quantity_scans];
  float dist_gap[quantity_scans];
  int num_gap;
  int n_scans;
  int index_gaps_i[quantity_scans];
  int index_gaps_j[quantity_scans];
};

//Funciones

void poseCallback(const nav_msgs::Odometry::ConstPtr &
    pose_message);
void laserCallback(const sensor_msgs::LaserScan::ConstPtr &
    laser_message);
void gridCallback(const nav_msgs::OccupancyGrid::ConstPtr &
    grid_message);
struct infoScanner scanner_step1(typeScanner scan);
struct infoScanner scanner_step2 (struct infoScanner
    scanner_forward, struct infoScanner scanner_backward);
void visualization_rviz (struct infoScanner scanner_gaps);
double degrees2radians(double angle_in_degrees);
double radians2degrees(double angle_in_radians);
void save_environment();
void sbpl_path();
void rotateREL (double relative_angle, double linear_speed,
    double maxRotVel);
void rotateABS (double angular_speed);
int Server();
void *connection_handler(void *);


int main(int argc, char **argv){
  // Initiate ROS node
  ros::init(argc, argv, "pr2_controller");
  ros::NodeHandle n;

  //Subscribers and publishers
  pose_subscriber = n.subscribe("/odom", 1, poseCallback);
  gaps_subscriber = n.subscribe("/base_scan", 1, laserCallback);
  grid_subscriber = n.subscribe("/map", 1, gridCallback);
```

```
  velocity_publisher = n.advertise<geometry_msgs::Twist>("/pr2/
      motion", 10);
  triangle_publisher = n.advertise<visualization_msgs::Marker>("
      /triangles", 10);
  line_strip_publisher = n.advertise<visualization_msgs::Marker
      >("/line_strip", 12);
  cloud_laser_scan_publisher = n.advertise<sensor_msgs::
      PointCloud>("/laser_scan",10);
  gaps_publisher = n.advertise<visualization_msgs::Marker>("/
      gaps",10);
  path_publisher = n.advertise<nav_msgs::Path>("/path", 10);
  goal_publisher = n.advertise<visualization_msgs::Marker>("/
      robot_goal", 10);

  double speed, angular_speed, angle;

  Server();

  return 0;
}

int Server(){

  int socket_desc , client_sock , c , *new_sock;
  struct sockaddr_in server , client;
  char client_message[2000];

  //Create socket
  socket_desc = socket(AF_INET , SOCK_STREAM , 0);
  if (socket_desc == -1){
    printf("Could not create socket");
  }
  puts("Socket created");

  //Prepare the sockaddr_in structure
  server.sin_family = AF_INET;
  server.sin_addr.s_addr = inet_addr("192.168.43.112");
  server.sin_port = htons( 8080 );
  int count = 0;

  //Bind
  if( bind(socket_desc,(struct sockaddr *)&server , sizeof(
      server)) < 0){
    //print the error message
    perror("bind failed. Error");
    return 1;
  }
  puts("bind done");

  printf("IP address is: %s\n", inet_ntoa(server.sin_addr));
  printf("port is: %d\n", (int) ntohs(server.sin_port));
```

```
  //Listen
  listen(socket_desc , 3);

  //Accept and incoming connection
  puts("Waiting for incoming connections...");
  c = sizeof(struct sockaddr_in);


  while( (client_sock = accept(socket_desc, (struct sockaddr *)&
      client, (socklen_t*)&c)) ){
    puts("Connection accepted");
    count++;
    printf("Thread #%i \n",count);
    printf("Client IP address: %s\n", inet_ntoa(client.sin_addr)
        );
    printf("port is: %d\n", (int) ntohs(client.sin_port));

    pthread_t sniffer_thread;
    new_sock = (int *)malloc(1);
    *new_sock = client_sock;


    //Cuando da -1 es que ha ocurrido un error
    if(pthread_create( &sniffer_thread , NULL ,
        connection_handler , (void*) new_sock) < 0){
      perror("could not create thread");
      return 1;
    }

    //Now join the thread , so that we dont terminate before the
        thread
    puts("Handler assigned");
  }

  if (client_sock < 0){
    perror("accept failed");
    return 1;
  }
  return 0;
}

void *connection_handler(void *socket_desc){
  //Get the socket descriptor
  int sock = *(int*)socket_desc;
  int read_size;
  char *message , client_message[2000],angulo_message[20],
      longitud_message[20],mode_message[1],velMax_message[5],
      angularMax_message[5];
  int numbytes;
  int i=0,e=0;
  float angulo=0,longitud=0;

  char x_goal_message[10]; //Coordenada x del punto objetivo
```

```
char y_goal_message[10]; //Coordenada y del punto objetivo


memset(client_message, 0, strlen(client_message));
ros::spinOnce();

field = radians2degrees(laser.angle_max - laser.angle_min); //
   Ángulo que abarca el sensor láser
number_scans = field/radians2degrees(laser.angle_increment);
   //Número de rayos totales del sensor láser

//Búsqueda forward del paso 1 del algoritmo Closest Gap
struct infoScanner scanner_forward = scanner_step1(forward);

//Búsqueda backward del paso 1 del algoritmo Closest Gap
struct infoScanner scanner_backward = scanner_step1(backward);

//Paso 2 del algoritmo Closest Gap
struct infoScanner scanner_gaps = scanner_step2 (
   scanner_forward, scanner_backward);

//Visualizar en Rviz los elementos de la herramienta
   desarrollada
visualization_rviz (scanner_gaps);

while( (read_size = recv(sock , client_message , 2000 , 0)) >
   0 ){
  //Send the message back to client

  // Guardar la coordenada x del punto objetivo
  while(client_message[i]!=' '){

    x_goal_message[e] = x_goal_message[e] + client_message[i];
    i++;
    e++;

  }
  x_goal = strtof(x_goal_message, NULL);

  //Guardar la coordenada y del punto objetivo
  i++;
  e=0;

  while(client_message[i]!=' '){

    y_goal_message[e] = y_goal_message[e] + client_message[i];
    i++;
    e++;

  }

  y_goal = strtof(y_goal_message, NULL);
```

```
    //ángulo
    i++;
    e=0;

    while(client_message[i]!=' '){
      angulo_message[i]=client_message[i];
      i++;
    }
    angulo=angulo_message[0] - '0';
    angulo=angulo+0.1*(angulo_message[2] - '0');
    angulo=angulo+0.01*(angulo_message[3] - '0');
    angulo=angulo+0.001*(angulo_message[4] - '0');
    angulo=angulo+0.0001*(angulo_message[5] - '0');
    angulo=angulo+0.00001*(angulo_message[6] - '0');
    angulo=angulo+0.000001*(angulo_message[7] - '0');

    //longitud
    i++;
    e=0;
    while(client_message[i]!=' '){
      longitud_message[e]=client_message[i];
      i++;
      e++;
    }
    i++;
    mode_message[0]=client_message[i];

    e=0;

    i++;
    i++;

    while(client_message[i]!=' '){
      velMax_message[e]=client_message[i];
      i++;
      e++;
    }

    longitud=longitud_message[0] - '0';
    longitud=longitud+0.1*(longitud_message[2] - '0');
    longitud=longitud+0.01*(longitud_message[3] - '0');
    longitud=longitud+0.001*(longitud_message[4] - '0');
    longitud=longitud+0.0001*(longitud_message[5] - '0');
    longitud=longitud+0.00001*(longitud_message[6] - '0');
    longitud=longitud+0.000001*(longitud_message[7] - '0');

    //mode
    mode=mode_message[0] - '0';

    //max velocity
    if(e>3){
      e=3;
    }
```

```
if (e==3){
  velMax=100*(velMax_message[0] - '0');
  velMax=velMax+10*(velMax_message[1] - '0');
  velMax=velMax+1*(velMax_message[2] - '0');
} else if (e==2){
  velMax=10*(velMax_message[0] - '0');
  velMax=velMax+(velMax_message[1] - '0');
} else {
  velMax=(velMax_message[0] - '0');
}

//max angle
i++;
e=0;

while(client_message[i]!=' '){
  angularMax_message[e]=client_message[i];
  i++;
  e++;
}
if(e>3){
  e=3;
}
if (e==3){
  angularMax=100*(angularMax_message[0] - '0');
  angularMax=angularMax+10*(angularMax_message[1] - '0');
  angularMax=angularMax+1*(angularMax_message[2] - '0');
} else if (e==2){
  angularMax=10*(angularMax_message[0] - '0');
  angularMax=angularMax+(angularMax_message[1] - '0');
} else {
  angularMax=(angularMax_message[0] - '0');
}

i=0;
e=0;

memset(client_message, 0, strlen(client_message));
memset(angulo_message, 0, strlen(angulo_message));
memset(longitud_message, 0, strlen(longitud_message));
memset(mode_message, 0, strlen(mode_message));
memset(velMax_message, 0, strlen(velMax_message));

angulos=(double)angulo;
velocidades=(double)longitud*5/2*velMax/100;
switch (mode){
  case 0:
  rotateABS (degrees2radians(200)*angularMax/100);
  break;
  case 1:
  rotateREL (angulos, velocidades,degrees2radians(200)*
     angularMax/100);
  break;
```

```
      default:
      break;
    }
  }

  if(read_size == 0){
    //memset(client_message, 0, sizeof client_message);
    puts("Client disconnected");
    fflush(stdout);
  }else if(read_size == -1){
    perror("recv failed");
  }

  //Free the socket pointer
  free(socket_desc);
  close(sock);
  pthread_exit(NULL);
  return 0;

}

void poseCallback(const nav_msgs::Odometry::ConstPtr &
   pose_message){

  car_pose.header.seq = pose_message-> header.seq;
  car_pose.header.stamp = pose_message-> header.stamp;
  car_pose.header.frame_id = pose_message-> header.frame_id;
  car_pose.pose.pose.orientation.x = pose_message->pose.pose.
     orientation.x;
  car_pose.pose.pose.orientation.y = pose_message->pose.pose.
     orientation.y;
  car_pose.pose.pose.orientation.z = pose_message->pose.pose.
     orientation.z;
  car_pose.pose.pose.orientation.w = pose_message->pose.pose.
     orientation.w;
  car_pose.pose.pose.position.x = pose_message->pose.pose.
     position.x;
  car_pose.pose.pose.position.y = pose_message->pose.pose.
     position.y;
  car_pose.pose.pose.position.z = pose_message->pose.pose.
     position.z;

  //Variable auxiliar a la posición del robot
  pose_aux.header = pose_message-> header;
  pose_aux.pose.position.x = pose_message->pose.pose.position.x;
  pose_aux.pose.position.y = pose_message->pose.pose.position.y;
  pose_aux.pose.position.z = 0.0;

  double t3= 2.0 * (car_pose.pose.pose.orientation.w*car_pose.
     pose.pose.orientation.z+car_pose.pose.pose.orientation.x*
     car_pose.pose.pose.orientation.y);
  double t4= 1.0-2.0 *((car_pose.pose.pose.orientation.y*
     car_pose.pose.pose.orientation.y)+car_pose.pose.pose.
```

```
      orientation.z*car_pose.pose.pose.orientation.z);

  car_pose.pose.pose.orientation.w=std::atan2(t3, t4);

  if(semaphore == true){ //Si ya se ha actualizado el entorno,
     se puede calcular el nuevo camino al punto objetivo

    sbpl_path();
    semaphore = false;
  }

}

void laserCallback(const sensor_msgs::LaserScan::ConstPtr &
    laser_message){

  laser.angle_min = laser_message->angle_min;
  laser.angle_max = laser_message->angle_max;
  laser.angle_increment = laser_message->angle_increment;
  laser.range_max = laser_message->range_max;
  laser.ranges = laser_message->ranges;

  laser_geometry::LaserProjection projector_;

  projector_.projectLaser(*laser_message, cloud_laser_scan);

  cloud_laser_scan_publisher.publish(cloud_laser_scan);

}


void gridCallback(const nav_msgs::OccupancyGrid::ConstPtr &
    grid_message){

  if(semaphore == false){

    grid.info.resolution = grid_message->info.resolution;
    grid.info.width = grid_message->info.width;
    grid.info.height = grid_message->info.height;
    grid.info.origin.position.x = grid_message->info.origin.
       position.x;
    grid.info.origin.position.y = grid_message->info.origin.
       position.y;
    grid.info.origin.position.z = grid_message->info.origin.
       position.z;
    grid.data = grid_message->data;

    save_environment();

    if (first_time == true){ //La primera vez siempre se tiene
       que calcular el camino al punto objetivo

      sbpl_path();
```

```
      first_time = false;
    }

    semaphore = true;
  }
}

struct infoScanner scanner_step1 (typeScanner scan){

  int i_ant, j_ant;
  float d, d_i, d_j;
  float d_ant = 10000;

  int angle = (number_scans*180)/field;
  int k = 0;
  int s, x;
  int z = 0;
  bool candau = false;
  int n_obstacles = 0;

  struct infoScanner step1;
  step1.n_scans = number_scans;

  if(scan == forward){
    x = 0;
    s = 1;
  }
  if(scan == backward){
    x = number_scans;
    s = -1;
  }

  for(int i = 0; i <= number_scans; i++){

    if(i > number_scans-angle){
      angle = angle - 1;
    }else{
      angle = (number_scans*180)/field;
    }

    if((laser.ranges[x+s*i+s*1] - laser.ranges[x+s*i]) > 2*
       radio_PR2){ //Discontinuidad de tipo 1
      for(int j = 1; j <= angle; j++){
        if(laser.ranges[x+s*i+s*j] < laser.range_max){
          d=sqrt((laser.ranges[x+s*i]*laser.ranges[x+s*i]) + (
              laser.ranges[x+s*i+s*j]*laser.ranges[x+s*i+s*j]) -
              (2*laser.ranges[x+s*i]*laser.ranges[x+s*i+s*j]*cos(
              j*laser.angle_increment)));

          if(d <= d_ant){ //Se ha encontrado el segundo extremo
              del gap
            d_ant = d;
            if(scan == forward){
```

```
            i_ant = x+s*i;
            j_ant = x+s*i+s*j;


          }else{
            j_ant = x+s*i;
            i_ant = x+s*i+s*j;
          }

          d_i = laser.ranges[i_ant];
          d_j = laser.ranges[j_ant];
          candau = true;
        }
      }
    }
  }else if((laser.ranges[x+s*i] < laser.range_max) && (laser.
     ranges[x+s*i+s*1] == laser.range_max)){ //Discontinuidad
     de tipo 2
    for(int j = 1; j <= angle; j++){
      if(laser.ranges[x+s*i+s*j] < laser.range_max){ //Se ha
         encontrado el segundo extremo del gap
        d_ant = sqrt((laser.ranges[x+s*i]*laser.ranges[x+s*i])
            + (laser.ranges[x+s*i+s*j]*laser.ranges[x+s*i+s*j
           ]) - (2*laser.ranges[x+s*i]*laser.ranges[x+s*i+s*j
           ]*cos(j*laser.angle_increment)));

        if(scan == forward){
          i_ant = x+s*i;
          j_ant = x+s*i+s*j;

        }else{
          j_ant = x+s*i;
          i_ant = x+s*i+s*j;
        }

        d_i = laser.ranges[i_ant];
        d_j = laser.ranges[j_ant];
        candau = true;

        break;

      }
    }
  }

  if(candau == true){ //Se guarda el gap
    k=k+1;

    if(scan == forward){
      f = 0;
      s = 1;
    }
    if(scan == backward){
```

```
        f = num_obstacles;
        s = -1;
      }

      step1.index_gaps_i[z] = f + s*(n_obstacles);
      step1.index_gaps_j[z] = f + s*(n_obstacles) + s;
      z++;
      step1.num_gap = k;
      step1.points_i[k-1] = i_ant;
      step1.points_j[k-1] = j_ant;
      step1.dist_gap[k-1] = d_ant;
      step1.dist_i[k-1] = d_i;
      step1.dist_j[k-1] = d_j;
    }

    if(laser.ranges[x + s*i] < laser.range_max){
      n_obstacles++;
    }

    d_ant=10000;
    candau= false;
  }

  num_obstacles = n_obstacles-1;

  return step1;
}


struct infoScanner scanner_step2 (struct infoScanner
    scanner_forward, struct infoScanner scanner_backward){

  int p_i[scanner_forward.num_gap + scanner_backward.num_gap];
  int p_j[scanner_forward.num_gap + scanner_backward.num_gap];
  float d[scanner_forward.num_gap + scanner_backward.num_gap];

  int p_i_gap[scanner_forward.num_gap + scanner_backward.num_gap
      ];
  int p_j_gap[scanner_forward.num_gap + scanner_backward.num_gap
      ];
  float d_gap[scanner_forward.num_gap + scanner_backward.num_gap
      ];
  float d_i_gap[scanner_forward.num_gap + scanner_backward.
      num_gap];
  float d_j_gap[scanner_forward.num_gap + scanner_backward.
      num_gap];


  int array_index_gaps_i[scanner_forward.num_gap +
      scanner_backward.num_gap];
  int array_index_gaps_j[scanner_forward.num_gap +
      scanner_backward.num_gap];
```

```
int p_i_equal[scanner_forward.num_gap + scanner_backward.
    num_gap];
int p_j_equal[scanner_forward.num_gap + scanner_backward.
    num_gap];



struct infoScanner step2;
step2.n_scans = scanner_forward.n_scans;

int n_2 = 0;
int n = 0; //Número de gaps > 2R
int n_gaps = 0;//Número total de gaps
bool key2 = false;
bool key1 = false;
int z = 0;
int k = 0;


for(int l = 0; l < scanner_forward.num_gap; l++){ //Guarda
    gaps de la búsqueda forward >2R

  if(scanner_forward.dist_gap[l] > 2*radio_PR2){
    p_i[n] = scanner_forward.points_i[l];
    p_j[n] = scanner_forward.points_j[l];
    d[n] = scanner_forward.dist_gap[l];

    array_index_gaps_i[n] = scanner_forward.index_gaps_i[l];

    array_index_gaps_j[n] = scanner_forward.index_gaps_j[l];

    n++;
  }
}

for(int l = 0; l < scanner_backward.num_gap; l++){ //Guarda
    gaps de la búsqueda backward >2R

  if(scanner_backward.dist_gap[l] > 2*radio_PR2){
    p_i[n] = scanner_backward.points_i[l];
    p_j[n] = scanner_backward.points_j[l];
    d[n] = scanner_backward.dist_gap[l];

    array_index_gaps_i[n] = scanner_backward.index_gaps_i[l];

    array_index_gaps_j[n] = scanner_backward.index_gaps_j[l];

    n++;
  }
}

for(int i = 0; i < n; i++){
```

```
  key2 = false;
  key1 = false;

  for(int j = 0; j < n; j++){

    if((i != j) && (p_i[i] >= p_i[j]) && (p_j[i] <= p_j[j])){

      key2 = true; //No se debe guardar debido a que esta
         dentro de otro gap

      if((p_i[i] == p_i[j]) && (p_j[i] == p_j[j])){

        for(int l = 0; l < n_2; l++){
          if((p_i[i] == p_i_equal[l]) && (p_j[i] == p_j_equal[
             l])){
            key2 = false;//Se debe guardar una vez como gap
               definitivo en p_gap debido a que es un gap
               encontrado por segunda vez, pero no se debe
               guardar dos veces
            key1 = true; //No lo volvemos a guardar en p_equal
          }
        }

        if(key1 == false){
          p_i_equal[n_2] = p_i[i];//Guardamos el primer gap
             encontrado en nuevo array p_equal para despues
             comparar
          p_j_equal[n_2] = p_j[i];
          n_2++;//Si existe un gap para comparar deberá hacer
             el for para comparar si de los próximos gaps hay
             alguno igual
        }
      }
    }
  }

  if(key2 == false){ //Guarda los gaps navegables

    step2.index_gaps_i[n_gaps] = array_index_gaps_i[i];
    step2.index_gaps_j[n_gaps] = array_index_gaps_j[i];

    p_i_gap[n_gaps] = p_i[i];
    p_j_gap[n_gaps] = p_j[i];
    d_gap[n_gaps] = d[i];
    d_i_gap[n_gaps] = laser.ranges[p_i_gap[n_gaps]];
    d_j_gap[n_gaps] = laser.ranges[p_j_gap[n_gaps]];

    n_gaps++;
  }

}
```

```
  for(int l = 0; l < n_gaps; l++){

    step2.points_i[l] = p_i_gap[l];
    step2.points_j[l] = p_j_gap[l];
    step2.dist_gap[l] = d_gap[l];
    step2.dist_i[l] = d_i_gap[l];
    step2.dist_j[l] = d_j_gap[l];
  }


  step2.num_gap = n_gaps;

  return step2;
}


void visualization_rviz (struct infoScanner scanner_gaps){

  float p_triangle_x[3*scanner_gaps.num_gap];
  float p_triangle_y[3*scanner_gaps.num_gap];
  float p_triangle_z[3*scanner_gaps.num_gap];
  int p_line_strip_x[num_points_line_strip];
  int p_line_strip_y[num_points_line_strip];

  float xp_gaps_i[scanner_gaps.num_gap];
  float yp_gaps_i[scanner_gaps.num_gap];
  float xp_gaps_j[scanner_gaps.num_gap];
  float yp_gaps_j[scanner_gaps.num_gap];
  float angle_gaps_i[scanner_gaps.num_gap];
  float angle_gaps_j[scanner_gaps.num_gap];
  float angle_line_strip[scanner_gaps.num_gap];

  if(2*scanner_gaps.num_gap - num_punts_ant < 0){ //Eliminar
     esferas que marcan los extremos de los gaps navegables

    marker_gaps.header.frame_id = "base_laser_link";
    marker_gaps.header.stamp = ros::Time();
    marker_gaps.ns = "id1";
    marker_gaps.action = visualization_msgs::Marker::DELETEALL;
    gaps_publisher.publish(marker_gaps);
  }

  marker_triangles.header.frame_id = "base_laser_link"; //
     Eliminar triángulos que marcan los gaps navegables
  marker_triangles.header.stamp = ros::Time();
  marker_triangles.ns = "id2";
  marker_triangles.action = visualization_msgs::Marker::
     DELETEALL;
  triangle_publisher.publish(marker_triangles);

  for(int j = 0; j < 2*scanner_gaps.num_gap; j++){  //Marker
     publica scan i y scan j de los gaps navegables
```

```
    marker_gaps.header.frame_id = "base_laser_link";
    marker_gaps.header.stamp = ros::Time();
    marker_gaps.ns = "id1";
    marker_gaps.id = j;
    marker_gaps.type = visualization_msgs::Marker::SPHERE;
    marker_gaps.action = visualization_msgs::Marker::ADD;

    if(j < scanner_gaps.num_gap){
      marker_gaps.pose.position.x = cloud_laser_scan.points[
          scanner_gaps.index_gaps_i[j]].x;
      marker_gaps.pose.position.y = cloud_laser_scan.points[
          scanner_gaps.index_gaps_i[j]].y;
      marker_gaps.pose.position.z = cloud_laser_scan.points[
          scanner_gaps.index_gaps_i[j]].z;

    }else{
      marker_gaps.pose.position.x = cloud_laser_scan.points[
          scanner_gaps.index_gaps_j[j-scanner_gaps.num_gap]].x;
      marker_gaps.pose.position.y = cloud_laser_scan.points[
          scanner_gaps.index_gaps_j[j-scanner_gaps.num_gap]].y;
      marker_gaps.pose.position.z = cloud_laser_scan.points[
          scanner_gaps.index_gaps_j[j-scanner_gaps.num_gap]].z;
    }

    marker_gaps.scale.x = 0.2; //dimensiones de la esfera
    marker_gaps.scale.y = 0.2;
    marker_gaps.scale.z = 0.2;
    marker_gaps.color.a = 1.0; //Para visualizar el marker
    marker_gaps.color.r = 0.0;
    marker_gaps.color.g = 1.0; //Color verde
    marker_gaps.color.b = 0.0;
    gaps_publisher.publish(marker_gaps);

  }

  for(int i = 0; i < scanner_gaps.num_gap; i++){ //Guardar en
      array 3 puntos del triángulo que marca espacio navegable

    for(int j = 0; j < 3; j++){

      if(scanner_gaps.index_gaps_i[i] - scanner_gaps.
          index_gaps_j[i] < 0){ //forward
        if(j == 0){ //scan i

          p_triangle_x[j + i*3] = cloud_laser_scan.points[
              scanner_gaps.index_gaps_i[i]].x;
          p_triangle_y[j + i*3] = cloud_laser_scan.points[
              scanner_gaps.index_gaps_i[i]].y;
          p_triangle_z[j + i*3] = cloud_laser_scan.points[
              scanner_gaps.index_gaps_i[i]].z;

        }else if(j == 1){ //scan j
```

```
        p_triangle_x[j + i*3] = cloud_laser_scan.points[
            scanner_gaps.index_gaps_j[i]].x;
        p_triangle_y[j + i*3] = cloud_laser_scan.points[
            scanner_gaps.index_gaps_j[i]].y;
        p_triangle_z[j + i*3] = cloud_laser_scan.points[
            scanner_gaps.index_gaps_j[i]].z;

      }else{ //centro del robot

        p_triangle_x[j + i*3] = 0.0 ;
        p_triangle_y[j + i*3] = 0.0 ;
        p_triangle_z[j + i*3] = 0.0 ;

      }

    }else{ //backward

      if(j == 0){ //scan i

        p_triangle_x[j + i*3] = cloud_laser_scan.points[
            scanner_gaps.index_gaps_j[i]].x;
        p_triangle_y[j + i*3] = cloud_laser_scan.points[
            scanner_gaps.index_gaps_j[i]].y;
        p_triangle_z[j + i*3] = cloud_laser_scan.points[
            scanner_gaps.index_gaps_j[i]].z;

      }else if(j == 1){ //scan j

        p_triangle_x[j + i*3] = cloud_laser_scan.points[
            scanner_gaps.index_gaps_i[i]].x;
        p_triangle_y[j + i*3] = cloud_laser_scan.points[
            scanner_gaps.index_gaps_i[i]].y;
        p_triangle_z[j + i*3] = cloud_laser_scan.points[
            scanner_gaps.index_gaps_i[i]].z;

      }else{ //centro del robot

        p_triangle_x[j + i*3] = 0.0 ;
        p_triangle_y[j + i*3] = 0.0 ;
        p_triangle_z[j + i*3] = 0.0 ;

      }
    }

  }
}

for(int j = 0; j < 3*scanner_gaps.num_gap; j++){  //
   Marker_triangles publica triángulos del espacio navegable

  marker_triangles.header.frame_id = "base_laser_link";
  marker_triangles.header.stamp = ros::Time();
  marker_triangles.ns = "id2";
```

```
  marker_triangles.id = j;
  marker_triangles.type = visualization_msgs::Marker::
     TRIANGLE_LIST;
  marker_triangles.action = visualization_msgs::Marker::ADD;

  geometry_msgs::Point point_triangle;

  point_triangle.x = p_triangle_x[j];
  point_triangle.y = p_triangle_y[j];
  point_triangle.z = p_triangle_z[j];

  marker_triangles.points.push_back(point_triangle);

  marker_triangles.scale.x = 1.0;
  marker_triangles.scale.y = 1.0;
  marker_triangles.scale.z = 1.0;
  marker_triangles.color.a = 1.0; //Para visualizar el marker
  marker_triangles.color.r = 0.0;
  marker_triangles.color.g = 0.0;
  marker_triangles.color.b = 1.0; //Color azul

}

triangle_publisher.publish(marker_triangles);


for(int j = 0; j < 3*scanner_gaps.num_gap; j++){
  marker_triangles.points.pop_back();
}


if(x_ant != car_pose.pose.pose.position.x or y_ant != car_pose
    .pose.pose.position.y){ //Publicación del camino recorrido
    por el robot

  global_path.header.frame_id = "odom";
  global_path.header.stamp = car_pose.header.stamp;
  global_path.header.seq = car_pose.header.seq;
  global_path.poses.push_back(pose_aux);


  cont_path = cont_path + 1;

  if (cont_path > 1000){
    global_path.poses.pop_back();
    cont_path = 0;
  }

  path_publisher.publish(global_path);
  x_ant = car_pose.pose.pose.position.x;
  y_ant = car_pose.pose.pose.position.y;
}
```

```
marker_line_strip.header.frame_id = "odom";
marker_line_strip.header.stamp = ros::Time();
marker_line_strip.ns = "id6";
marker_line_strip.action = visualization_msgs::Marker::
    DELETEALL;
line_strip_publisher.publish(marker_line_strip);

for(int j = 0; j < num_points_line_strip; j++){  //
    Marker_line_strip publica el camino hasta el punto objetivo
     generado por el algoritmo de planificación AD*

  marker_line_strip.header.frame_id = "odom";
  marker_line_strip.header.stamp = ros::Time();
  marker_line_strip.ns = "id6";
  marker_line_strip.id = j;
  marker_line_strip.type = visualization_msgs::Marker::
      LINE_STRIP;
  marker_line_strip.action = visualization_msgs::Marker::ADD;

  p_line_strip_x[j] = array_path_x[j];
  p_line_strip_y[j] = array_path_y[j];

  geometry_msgs::Point point_line_strip;

  point_line_strip.x = (float)((p_line_strip_x[j]*grid.info.
      resolution) + grid.info.origin.position.x);
  point_line_strip.y = (float)((p_line_strip_y[j]*grid.info.
      resolution) + grid.info.origin.position.y);
  point_line_strip.z = 0.5;

  marker_line_strip.points.push_back(point_line_strip);

  marker_line_strip.scale.x = 0.1;
  marker_line_strip.color.a = 1.0; //Para visualizar el marker
  marker_line_strip.color.r = 1.0; //Color rojo
  marker_line_strip.color.g = 0.0;
  marker_line_strip.color.b = 0.0;

}

line_strip_publisher.publish(marker_line_strip);

for(int j = 0; j < num_points_line_strip; j++){
  marker_line_strip.points.pop_back();
}

for(int j = 0; j < 1; j++){  //Marker publica el punto
    objetivo

  marker_goal.header.frame_id = "odom";
  marker_goal.header.stamp = ros::Time();
  marker_goal.ns = "id3";
  marker_goal.id = j;
```

```
    marker_goal.type = visualization_msgs::Marker::SPHERE;
    marker_goal.action = visualization_msgs::Marker::ADD;

    marker_goal.pose.position.x = x_goal;
    marker_goal.pose.position.y = y_goal;
    marker_goal.pose.position.z = 0.5;

    marker_goal.scale.x = 0.3; //Dimensiones de la esfera
    marker_goal.scale.y = 0.3;
    marker_goal.scale.z = 0.3;
    marker_goal.color.a = 1.0; //Para visualizar el marker
    marker_goal.color.r = 1.0;
    marker_goal.color.g = 0.0; //Color verde
    marker_goal.color.b = 0.0;
    goal_publisher.publish(marker_goal);

  }

  num_punts_ant = 2*scanner_gaps.num_gap;
  num_punts_ant2 = 3*scanner_gaps.num_gap;

}

double degrees2radians(double angle_in_degrees){
  return angle_in_degrees *PI /180.0;
}

double radians2degrees(double angle_in_radians){
  return angle_in_radians *180.0/PI;
}


void save_environment(){ //Crear fichero del entorno

  int array_map [grid.info.height][grid.info.width];
  int col = grid.info.width-1;
  int cost_function_unknown = 1;
  int m = 0;
  int j = 0;
  int c_x[grid.info.height*grid.info.width];
  int c_y[grid.info.height*grid.info.width];
  int cont_c = 0;
  int n_capas;
  ofstream myfile;

  myfile.open ("/home/usuario/Escritorio/environment.cfg");
  myfile << "discretization(cells): ";
  myfile << grid.info.width;
  myfile << " ";
  myfile << grid.info.height;
  myfile << "\n";
  myfile << "obsthresh: 100";
  myfile << "\n";
```

```
myfile << "start(cells): ";

myfile<< ((int)((-grid.info.origin.position.x + car_pose.pose.
    pose.position.x)/grid.info.resolution));
myfile << " ";
myfile<< ((int)((-grid.info.origin.position.y + car_pose.pose.
    pose.position.y)/grid.info.resolution));
myfile << "\n";
myfile << "end(cells): ";
myfile<< ((int)(((-grid.info.origin.position.x + x_goal)/grid.
    info.resolution)-1));
myfile << " ";
myfile<< ((int)(((-grid.info.origin.position.y + y_goal)/grid.
    info.resolution)-1));

myfile << "\n";
myfile << "environment: \n";


for(int i = 0; i < (grid.info.width*grid.info.height); i++){

  if((double)grid.data[i]== -1){ //unknown
    array_map[j][m] = cost_function_unknown;
  }else{
    array_map[j][m] = (double)grid.data[i];
  }

  if((double)grid.data[i] == 100){
    c_x[cont_c]=m;
    c_y[cont_c]=j;
    cont_c++;
  }

  m++;

  if(i == col){
    col = i + grid.info.width;
    m = 0;
    j++;
  }

}

//Agrandar obstáculos
n_capas = ceil(radio_PR2/(double)grid.info.resolution);

for(int i = 0; i < cont_c; i++){

  for(int t = 1; t <= n_capas; t++){

    array_map[c_y[i]-t][c_x[i]-t] = 100;
```

```
      array_map[c_y[i]-t][c_x[i]] = 100;

      array_map[c_y[i]-t][c_x[i]+t] = 100;

      array_map[c_y[i]][c_x[i]-t] = 100;

      array_map[c_y[i]][c_x[i]+t] = 100;

      array_map[c_y[i]+t][c_x[i]-t] = 100;

      array_map[c_y[i]+t][c_x[i]] = 100;

      array_map[c_y[i]+t][c_x[i]+t] = 100;

      for(int z = 1; z < t; z++){

        array_map[c_y[i]-t][c_x[i]-z] = 100;
        array_map[c_y[i]-t][c_x[i]+z] = 100;

        array_map[c_y[i]+t][c_x[i]-z] = 100;
        array_map[c_y[i]+t][c_x[i]+z] = 100;

        array_map[c_y[i]-z][c_x[i]-t] = 100;
        array_map[c_y[i]-z][c_x[i]+t] = 100;

        array_map[c_y[i]+z][c_x[i]-t] = 100;
        array_map[c_y[i]+z][c_x[i]+t] = 100;
      }
    }
  }

  for(int i = 0; i < grid.info.height; i++){
    for(int j = 0; j < grid.info.width; j++){

      myfile << array_map[i][j];
      myfile << " ";
    }

    myfile << "\n";

  }

  myfile.close();
}

void sbpl_path(){ //Llamar a la librería SBPL y leer el fichero
    sol.txt que contiene el recorrido generado por el algoritmo
    de planificaación AD*

  int bRet = 0;
  double allocated_time_secs = 100.0; // in seconds
  double initialEpsilon = 1.0;
  MDPConfig MDPCfg;
```

```
bool bsearchuntilfirstsolution = true;
bool bforwardsearch = 0; //forward=0, backward=1
const char* env = "/home/usuario/Escritorio/environment.cfg";


// Initialize Environment (should be called before
   initializing anything else)
MyEnvironment environment_nav2D;
if (!environment_nav2D.InitializeEnv(env)) {
  printf("ERROR: InitializeEnv failed\n");
  throw new SBPL_Exception();
}

// Initialize MDP Info
if (!environment_nav2D.InitializeMDPCfg(&MDPCfg)) {
  printf("ERROR: InitializeMDPCfg failed\n");
  throw new SBPL_Exception();
}

//Plan a path
vector<int> solution_stateIDs_V;

printf("Initializing ADPlanner...\n");
ADPlanner* planner = new ADPlanner(&environment_nav2D,
   bforwardsearch);

// set search mode
planner->set_search_mode(bsearchuntilfirstsolution);

if (planner->set_start(MDPCfg.startstateid) == 0) {
  printf("ERROR: failed to set start state\n");
  throw new SBPL_Exception();
}

if (planner->set_goal(MDPCfg.goalstateid) == 0) {
  printf("ERROR: failed to set goal state\n");
  throw new SBPL_Exception();
}

planner->set_initialsolution_eps(initialEpsilon);

printf("start planning...\n");
bRet = planner->replan(allocated_time_secs, &
   solution_stateIDs_V);
printf("done planning\n");
std::cout << "size of solution=" << solution_stateIDs_V.size()
    << std::endl;

environment_nav2D.PrintTimeStat(stdout);

const char* sol = "sol.txt";
FILE* fSol = fopen(sol, "w");
if (fSol == NULL) {
```

```
    printf("ERROR: could not open solution file\n");
    throw new SBPL_Exception();
  }
  for (unsigned int i = 0; i < solution_stateIDs_V.size(); i++)
      {
    environment_nav2D.PrintState(solution_stateIDs_V[i], fSol,
       true);
  }
  fclose(fSol);

  if (bRet) {
    //print the solution
    printf("Solution is found\n");
  }
  else {
    printf("Solution does not exist\n");
  }

  fflush(NULL);


  //Leer fichero sol.txt que contiene recorrido óptimo y
     guardarlo en dos arrays la posición x e y
  ifstream myReadFile;
  int x = 0;
  int l = 0;
  num_points_line_strip = 0;

  myReadFile.open("/home/usuario/catkin_ws/sol.txt");
  char output[100];
  if (myReadFile.is_open()) {
    while (!myReadFile.eof()) {
      if(x%2 == 0){
      myReadFile >> output;
      array_path_x[l] = atoi(output);
      x++;

    }else{
      myReadFile >> output;
      array_path_y[l] = atoi(output);
      x++;
      l++;
      num_points_line_strip++;
    }
    cout<<" ";
  }
}

myReadFile.close();
}

void rotateREL (double relative_angle, double linear_speed,
   double maxRotVel){
```

```
double angular_speed;
//set a 0 linear velocity in the y-z axis
vel_msg.linear.x =linear_speed;
vel_msg.linear.y =0;
vel_msg.linear.z =0;
//set a 0 angular velocity in the x-y-z axis
vel_msg.angular.x = 0;
vel_msg.angular.y = 0;
vel_msg.angular.z = 0;
ros::spinOnce(); //inicializar valores

//Cálculo de la w correspondiente al punto
if(relative_angle<=degrees2radians(90) || (relative_angle<=
    degrees2radians(270) && relative_angle>=degrees2radians(180))
    ){
  if(relative_angle<=degrees2radians(90)){
    angular_speed=maxRotVel-(relative_angle*maxRotVel/
        degrees2radians(90));
  }else{
    angular_speed=maxRotVel-((relative_angle-degrees2radians
        (180))*maxRotVel/degrees2radians(90));
  }
}
if(relative_angle>=degrees2radians(270) || (relative_angle<=
    degrees2radians(180) && relative_angle>=degrees2radians(90)))
    {
  if(relative_angle>=degrees2radians(270)){
    angular_speed=(relative_angle-degrees2radians(270))*
        maxRotVel/degrees2radians(90);
  }else{
    angular_speed=(relative_angle+degrees2radians(-270+180))*
        maxRotVel/degrees2radians(90);
  }
}
//asignar sentido de giro
if(relative_angle>=degrees2radians(90) && relative_angle<=
    degrees2radians(270))
vel_msg.angular.z =abs(angular_speed);
else
vel_msg.angular.z =-abs(angular_speed);

if(linear_speed==0){
  vel_msg.angular.z = 0;
}
//linear speed
if(relative_angle>degrees2radians(180)){
  vel_msg.linear.x = -linear_speed;
}
//publish
velocity_publisher.publish(vel_msg);

}
```

```
void rotateABS (double angular_speed){

//set a 0 linear velocity in the x-y-axis
vel_msg.linear.x =velocidades;
vel_msg.linear.y =0;
vel_msg.linear.z =0;
//set a 0 angular velocity in the x-y-axis
vel_msg.angular.x = 0;
vel_msg.angular.y = 0;
vel_msg.angular.z = 0; //por si acaso
ros::spinOnce(); //si no da 0 w

int clockwise=0;
double positive_w=car_pose.pose.pose.orientation.w;

if (car_pose.pose.pose.orientation.w<0)
positive_w=degrees2radians(360)+car_pose.pose.pose.orientation.w
    ;

double angulos360=angulos+2*PI;
double w360=positive_w+2*PI;

if (angulos>positive_w) //falta camino corto
if((w360-angulos)<(angulos-positive_w))
vel_msg.angular.z =-abs(angular_speed), clockwise=3;//de angulo
    positivo a negativo horario
else
vel_msg.angular.z =abs(angular_speed), clockwise=0;//antihorario
     de 0 2pi no de 2pi a 0
else if ((angulos360-positive_w)<(positive_w-angulos))
vel_msg.angular.z =abs(angular_speed), clockwise=1;//de negativo
     a positivo antihorario
else
vel_msg.angular.z =-abs(angular_speed), clockwise=2;//horario de
     2pi a 0

double end_angle = angulos-positive_w;//diference between goal
    and current angle
bool big2small_angle = 0;
bool firstTime=0;

if (angulos-positive_w>0)
big2small_angle=0;
else
big2small_angle=1;

ros::Rate loop_rate(80);

do{
  if(velocidades==0){
    vel_msg.angular.z =0,
    velocity_publisher.publish(vel_msg);
  }else{
```

```
    if (car_pose.pose.pose.orientation.w<0)
    positive_w=degrees2radians(360)+car_pose.pose.pose.
       orientation.w;
    else
    positive_w=car_pose.pose.pose.orientation.w;

    if (big2small_angle && clockwise==2) //horario
    end_angle = -(angulos-positive_w);
    else if (clockwise==1 && big2small_angle && angulos<
       positive_w && !firstTime) //current_angle>angulos
    end_angle = -(angulos-positive_w);
    else
    if(clockwise==3)
    if(positive_w<degrees2radians(180))
    end_angle = (angulos-positive_w);
    else
    end_angle = positive_w-angulos;
    else
    end_angle = angulos-positive_w,
    firstTime=1;

    ros::spinOnce();
    loop_rate.sleep();
    if(end_angle>0.17){
      velocity_publisher.publish(vel_msg);
    }
  }
}while(end_angle>0.17 && velocidades>0);
vel_msg.angular.z =0;
velocity_publisher.publish(vel_msg);

}
```

APÉNDICE B

# CÓDIGO *morse_pr2.py*

```
from morse.builder import *

# PR2 land robot
pr2 = BasePR2()

# An odometry sensor
odom = Odometry()
odom.add_interface('ros', topic="/odom")
pr2.append(odom)


# Keyboard control
keyboard = Keyboard()
pr2.append(keyboard)

# Motion controller
pr2.motion = MotionXYW()
pr2.motion.add_interface('ros', topic="/pr2/motion")
pr2.motion.properties(ControlType = 'Position')
pr2.append(pr2.motion)

# A laser sensor
laser = Hokuyo()
laser.properties(Visible_arc = True)
laser.properties(laser_range = 4)
laser.properties(resolution = 1.0)
laser.properties(scan_window = 270.0)
laser.frequency(10)
laser.add_interface('ros', topic='/base_scan')
laser.translate(0, 0, 0.5)
pr2.append(laser)
```

```
# Add default interface for our robot's components
pr2.add_default_interface('ros')

# Load the 3D environment
env = Environment('/home/usuario/Escritorio/Entornos/env9')
```

# BIBLIOGRAFÍA


[1] D. Arévalo Jiménez, "Desarrollo de una interfaz de teleoperación de robots móviles para dispositivos android," 2017. 1.3, 2.1, 2.2, 2.2, 4, 4.3, 6.1

[2] Wikipedia, "Sistema operativo robótico." [Online]. Available: https://es.wikipedia.org/wiki/Sistema_Operativo_Rob%C3%B3tico 1.3, 1.4, 1.4.1

[3] Openrobots, "What is morse?" [Online]. Available: https://www.openrobots.org/morse/doc/latest/what_is_morse.html 1.3, 1.4.1

[4] Wikipedia, "Android studio." [Online]. Available: https://es.wikipedia.org/wiki/Android_Studio 1.4, 1.4.2

[5] W. ROS, "Rviz." [Online]. Available: http://wiki.ros.org/rviz 1.4.1

[6] ——, "Gmapping." [Online]. Available: http://wiki.ros.org/gmapping 1.4.1

[7] ——, "Sbpl." [Online]. Available: http://wiki.ros.org/sbpl 1.4.1

[8] M. Likhachev, "Software sbpl." [Online]. Available: https://www.cs.cmu.edu/~maxim/software.html 2.4

[9] Wikipedia, "Algoritmo de búsqueda a*." [Online]. Available: https://es.wikipedia.org/wiki/Algoritmo_de_b%C3%BAsqueda_A* 2.4.2

[10] ——, "Algoritmo de búsqueda d*." [Online]. Available: https://es.wikipedia.org/wiki/D* 2.4.2

[11] W. ROS, "Display markers." [Online]. Available: http://wiki.ros.org/rviz/DisplayTypes/Marker#Message_Parameters 2.5

[12] ——, "Documentation visualization_msgs/marker message." [Online]. Available: http://docs.ros.org/melodic/api/visualization_msgs/html/msg/Marker.html 2.5

[13] ——, "Documentation sensor_msgs/laserscan message." [Online]. Available: http://docs.ros.org/melodic/api/sensor_msgs/html/msg/LaserScan.html 3.1

[14] B. M. Muhannad Mujahad, Dirk Fischer and H. Jaddu, "Closest gap based (cg) reactive obstacle avoidance navigation for highly cluttered environments," *The 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, October 18-22, 2010, Taipei, Taiwan. 3.3, 3.3.3

[15] W. ROS, "Documentation package laser_geometry." [Online]. Available: http://wiki.ros.org/laser_geometry 3.4.1

[16] ——, “Documentation geometry_msgs/posestamped.” [Online]. Available: http://docs.ros.org/lunar/api/geometry_msgs/html/msg/PoseStamped.html 3.4.2

[17] ——, “Ros tutorials.” [Online]. Available: http://wiki.ros.org/ROS/Tutorials 6.4

[18] ——, “Package tf.” [Online]. Available: http://wiki.ros.org/tf 6.4.1

Último acceso de todas las páginas web: 10-09-2018.