Universitat de les Illes Balears

escola politècnica superior

Trabajo Fin de Grado

GRADO DE INGENIERÍA ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA

Estudio e implementación del algoritmo de navegación reactiva *Closest Gap* sobre el entorno ROS con interfaz gráfica en Android

JUAN ISAAC CIFRE IZQUIERDO

Tutores

Javier Antich Tobaruela Alberto Ortiz Rodríguez

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

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

Índice de Tablas

ACRÓNIMOS

TFG Trabajo final de grado

CG Closest Gap

ROS Robot Operating System

MORSE Modular Open Robots Simulation Engine

MATLAB MATrix LABoratory

SND Smooth Nearness Diagram

ATRV All Terrain Robot Vehicle

RESUMEN

Poder navegar de forma autónoma de un punto a otro del espacio es una de las tareas más fundamentales que un robot móvil debe realizar. Esta funcionalidad tiene una importancia muy relevante en la actualidad, ya que de esta forma el robot puede llevar a cabo tareas de transporte, asistencia o incluso rescate sin la necesidad de un control humano.

Por ello, se propone el estudio e implementación de un algoritmo de navegación autónoma llamado *Closest Gap*, con el cual se pretende abordar situaciones en las que el robot se encuentre rodeado de muchos obstáculos. De esta forma, se quiere dar solución a los casos en los que una persona humana sería incapaz de controlar al robot debido a la complejidad que pueda existir. Por otra parte, también se propone la realización de una interfaz gráfica para dispositivos Android en la que en todo momento se podrá observar en que situación se encuentra el robot.

Parte inherente a la implementación del algoritmo, será la demostración de su correcto funcionamiento. Para ello, se realizarán una serie de simulaciones en las que se podrá observar cómo se comporta un robot móvil con el algoritmo ya incorporado.

CAPÍTULO 1

INTRODUCCIÓN

En este primer capítulo se aborda la motivación que llevó a cabo la realización del proyecto y los objetivos principales que se desean conseguir. Se explica de forma breve los principales algoritmos de navegación existentes y además, se deja constancia del punto de partida que forma la base del proyecto. Por otro lado, se nombran las herramientas utilizadas y finalmente, se expone la estructura del presente documento.

1.1 Motivación

El motivo principal por el cual se planteó la realización de este proyecto fue abordar uno de los problemas básicos de la robótica móvil, tal como es la navegación autónoma. Conseguir que un robot móvil complete un recorrido desde un punto A hasta un punto B sin colisionar con ningún obstáculo es una tarea primordial que desde hace tiempo se desea poder realizar de forma completamente autónoma.

En concreto, en este proyecto, estamos interesados en tratar entornos repletos de obstáculos (*cluttered environments*).

En la figura 1.1 se puede observar un ejemplo de entorno con una gran cantidad de obstáculos a evitar, en el cual además se requiere de una gran precisión a la hora de maniobrar debido al pequeño margen que existe entre los obstáculos presentes.

Debido a la existencia de entornos como el de la figura 1.1, se ha planteado el uso de un tipo de algoritmo que permita lidiar con este tipo de situaciones sin poner en riesgo la integridad del robot que se vaya a utilizar.

1.2 Objetivos

El objetivo principal de este proyecto es el de estudiar e implementar un algoritmo de navegación autónoma llamado *Closest Gap* [1]. Con la utilización de este algoritmo, se pretende que el robot navegue de forma autónoma, y a la vez segura, en entornos simulados parecidos a los descritos anteriormente.

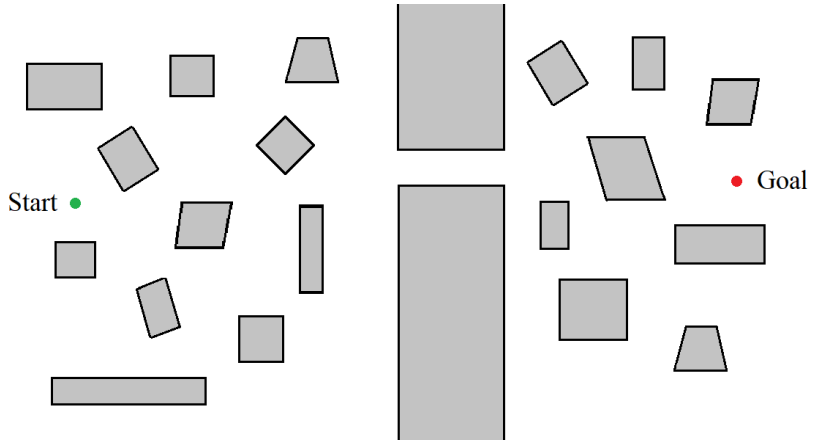


Figura 1.1: Ejemplo de entorno con gran densidad de obstáculos. El robot debe ir del punto *Start* al punto *Goal*.

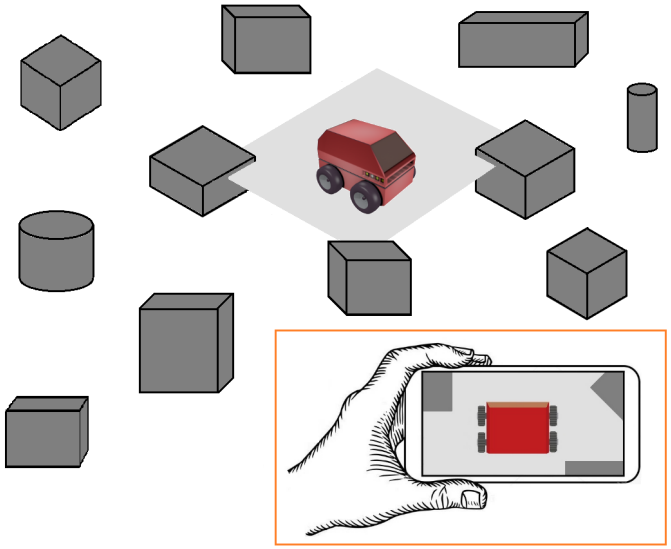


Figura 1.2: Ilustración de la aplicación Android a desarrollar para ver de forma local en qué situación se encuentra el robot.

Debido a que el robot tiene que moverse por espacios muy estrechos y con una alta densidad de obstáculos, se quiere implementar una interfaz gráfica en Android para poder visualizar de forma local y en todo momento los movimientos que realice el robot. Además, con esta interfaz, el usuario podrá apreciar en gran parte las decisiones que el robot va tomando. En la figura 1.2 , se puede ver una representación gráfica de la utilidad que se le quiere dar a la aplicación.

Finalmente, se quiere llevar a cabo unas exhaustivas pruebas en orden creciente de complejidad para así poder evaluar el algoritmo. Estas pruebas se realizarán en entornos diseñados específicamente para poder comprobar de forma clara las fortalezas y debilidades del algoritmo implementado.

1.3 Algoritmos de navegación

Para poder entender en qué posición se sitúa el algoritmo elegido, será necesario enunciar de forma breve las tres arquitecturas de navegación existentes [2]. Éstas se clasifican según el comportamiento del robot a la hora de generar acciones a partir de su percepción del entorno:

- Las arquitecturas deliberativas se basan en el paradigma *sense-plan-act (SPA)*. Se necesita obtener un modelo completo del entorno (mapa) para así poder construir un plan de acción anticipado al movimiento del robot. Una vez que se tenga el plan calculado, éste se podrá ejecutar.

- Las arquitecturas reactivas se deshacen de la necesidad de tener un modelo previo del entorno para así establecer una conexión directa entre la percepción del robot y el movimiento de éste. Desaparece el término *plan* para centrarse en los aspectos *sense-act*.

- Las arquitecturas híbridas combinan las características de las arquitecturas deliberativas y reactivas para poder aprovechar las fortalezas que ambas ofrecen. Se formará una estructura basada en capas para poder trabajar de forma paralela, de esta forma se puede planear con antelación los movimientos del robot y a la vez tener una buena capacidad de reacción frente a situaciones imprevistas.

Dentro de esta clasificación de arquitecturas de navegación, el algoritmo *Closest Gap* se encuentra en la lista de algoritmos reactivos. Es un algoritmo con un bajo requerimiento de cómputo y una respuesta rápida. Estas características son ideales para poder navegar en entornos con una gran cantidad de obstáculos, tal y como se ha propuesto anteriormente.

Por otra parte, hay que tener en cuenta que es complicado realizar tareas complejas con un algoritmo de este tipo y que no se garantiza la mejor solución a la hora de encontrar el camino hacia el punto objetivo.

1.4 Punto de partida

La base de este proyecto se establece sobre un trabajo final de grado el cual se encargaba de la teleoperación de robots móviles desde dispositivos Android [3].

Con una aplicación Android y un controlador en el robot se podía teleoperar un robot móvil gracias a la posición y orientación que ofrece un dispositivo de este tipo a través de los sensores inerciales (giróscopo, acelerómetros y compás) que integra. Mediante la fusión de los sensores mencionados, la aplicación resultante del anterior proyecto permitía controlar de forma precisa el robot.

A este trabajo ya realizado se le pretende añadir una nueva funcionalidad consistente en poder especificar el punto destino al cual tiene que navegar. Adicionalmente, se establecerán consignas de control desde la aplicación Android y el algoritmo de navegación *Closest Gap*, una vez implementado, se ejecutará sobre el robot.

1.5 Herramientas utilizadas

A continuación se revisan de forma breve las herramientas que se han utilizado durante la realización del proyecto.

1.5.1 Android Studio

Android Studio es el entorno de desarrollo integrado oficial (IDE) para la plataforma Android [4]. Gracias al servicio integral que ofrece (editor de código fuente, herramientas de construcción automáticas y un depurador) se le facilita mucho el desarrollo de software al programador.

Éste ha sido el entorno utilizado para el desarrollo de la funcionalidad extra para la aplicación Android ya existente, un modo a través del cual se puede especificar consignas de control al robot, además de permitir representar de forma local el entorno del mismo.

1.5.2 ROS

Robot Operating System (ROS) es un entorno de trabajo para el desarrollo de *software* para robots [5]. Alguno de los servicios que proporciona son la abstracción del *hardware*, una comunicación flexible y ágil entre procesos o el mantenimiento de paquetes de *software*, entre muchos otros. Gracias a ROS y a la gran cantidad de paquetes que ofrece la comunidad, se pueden desarrollar aplicaciones complejas con múltiples clases de robots, sin tener que preocuparse apenas de las comunicaciones robot-robot y robot-estación base.

Elementos de una aplicación ROS

Dentro de una aplicación creada en el entorno ROS, existen diversos elementos que son básicos a la hora de desarrollar aplicaciones. Estos elementos son los siguientes:

- Los paquetes son la unidad de organización de *software* del código que se programa en ROS. Cada paquete puede contener librerías, ejecutables, *scripts* y otros elementos. Los paquetes son útiles para poder encapsular todo lo referido a un proyecto en un mismo lugar.

- Los nodos son los procesos ejecutables en una aplicación ROS. Dentro de un mismo paquete pueden coexistir distintos nodos, los cuales a su vez se pueden comunicar entre ellos para poder llevar a cabo una tarea conjunta.

- El nodo máster es el nodo principal, el cual controla la transferencia de mensajes entre los demás nodos. Actúa como *broker* para así garantizar el correcto funcionamiento del sistema.

- Los mensajes son tipos de datos que utiliza ROS para poder intercambiar información entre nodos.

- Los nodos pueden suscribirse o publicar en *topics*, una de las formas a través de las que los nodos envían o reciben información.

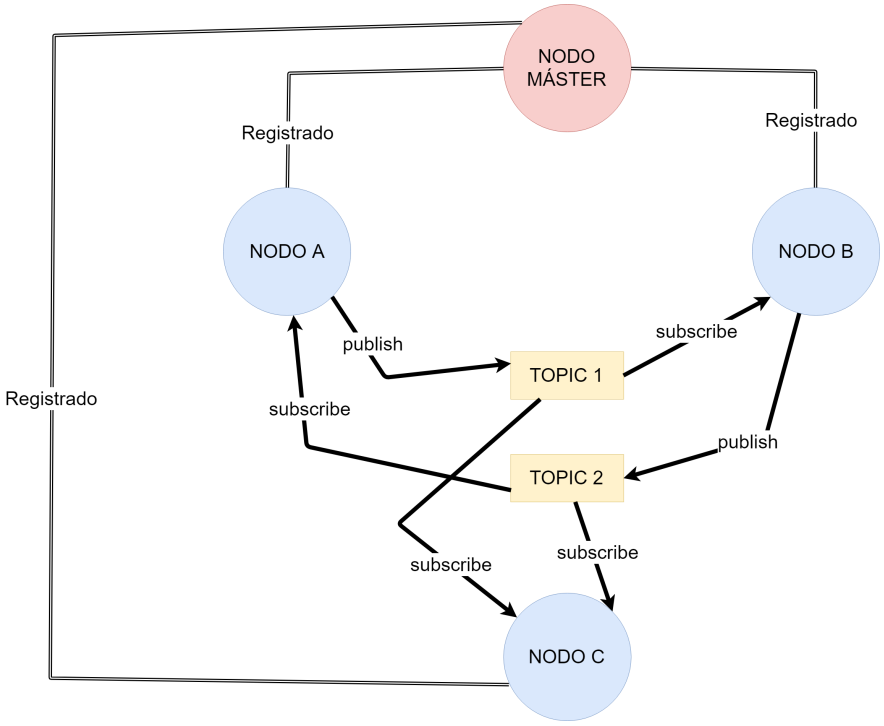


Figura 1.3: Ejemplo comunicación entre nodos en ROS.

- ROS emplea un mecanismo basado en publicadores y suscriptores (*publisher/-subscriber*) para definir los enlaces de comunicación entre nodos. Un nodo puede suscribirse a un *topic* en el cual otro nodo se encarga de publicar mensajes. Sólo cuando el nodo suscrito al *topic* correspondiente recibe nueva información, el nodo publicador genera un mensaje nuevo.

En la figura 1.3 se puede ver un pequeño ejemplo de comunicación entre nodos. Cada uno de los nodos está conectado al máster para que pueda gestionar el envío de mensajes. Se puede ver como gracias a la existencia de dos *topics* distintos, tres nodos se pueden comunicar entre ellos mediante el modelo *publisher/subscriber*.

MORSE

Modular Open Robots Simulation Engine (MORSE) es un simulador que se centra en simulaciones 3D realistas. Las simulaciones que se realizan en MORSE exigen de la elaboración de pequeños *scripts* escritos en *Python*, los cuales describen el entorno de simulación y los robots. Junto a la creación de robots también se puede hacer uso de una larga lista de sensores y actuadores para éstos.

MORSE se comporta como un nodo más de nuestro sistema. Se suscribe a los distintos *topics* relacionados con los actuadores del robot y se encarga de publicar en todo momento los valores de los sensores que se le añaden.

En la figura 1.4 se puede ver un ejemplo de entorno en MORSE. Se trata de un entorno sencillo donde se ha creado un robot con un sensor de barrido láser y tres obstáculos con forma de rectángulo.

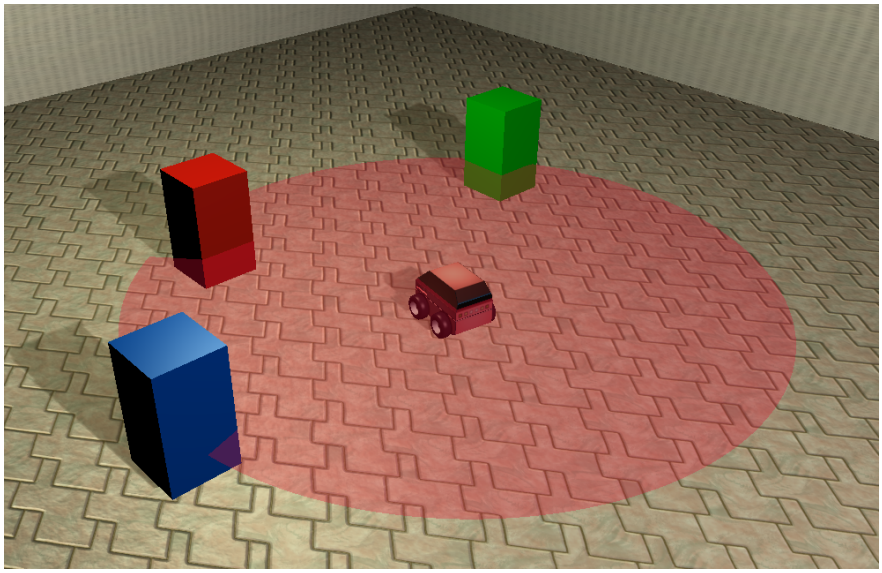


Figura 1.4: Ejemplo de entorno sencillo en MORSE.

1.5.3 MATLAB

MATrix LABoratory (MATLAB) es una herramienta de software matemático que ofrece un IDE con un lenguaje de programación propio (lenguaje M) [6]. MATLAB dispone de multitud de paquetes (*Toolboxes*) para poder expandir sus prestaciones. Las características que destacan a MATLAB del resto de plataformas son la posibilidad de trabajar de forma muy eficiente con matrices incluyendo su representación en memoria, y la facilidad para implementar algoritmos.

En este caso, se ha utilizado MATLAB para la fase final del proyecto, en particular, a la hora de documentar las pruebas que se describen en el capítulo 4.

1.6 Estructura del documento

Este documento se divide en cinco capítulos:

- Capítulo 1: Introducción, en el que se realiza una primera explicación de la temática general del trabajo.
- Capítulo 2: Descripción del algoritmo *Closest Gap*, en el cual se describe de forma detallada este algoritmo.
- Capítulo 3: Implementación y desarrollo de la interfaz Android, donde se explican los detalles del desarrollo de la aplicación Android.
- Capítulo 4: Informe y análisis de resultados, en el cual se reportan y discuten las pruebas realizadas.
- Capítulo 5: Conclusiones, donde se valora el trabajo realizado y los resultados obtenidos.

CAPÍTULO 2

DESCRIPCIÓN DEL ALGORITMO *Closest Gap* (CG)

El algoritmo *Closest Gap* se presentó en el artículo *Closest Gap Based (CG) Reactive Obstacle Avoidance Navigation for Highly Cluttered Environments* en el año 2010 y fue creado por Muhannad Mujahad, Dirk Fischer, Bärbel Mertsching y Hussein Jaddu [1].

En este capítulo, se realiza la explicación teórica de este algoritmo. En primer lugar, se exponen los antecedentes y los estudios previos realizados. Posteriormente, se presenta el objetivo principal del algoritmo, y para finalizar se realiza la explicación en detalle de las tres fases principales de que consta.

2.1 Antecedentes

Tal y como se explicó en el apartado 1, se pretende dar solución a los casos en los que un robot móvil se encuentre en entornos cambiantes y con una gran cantidad de obstáculos.

Muchos de los algoritmos reactivos presentan una serie de problemas a la hora de tratar con entornos de este tipo. Algunos de estos problemas son:

- *Mínimos locales*. En determinadas ocasiones un algoritmo reactivo puede quedar atrapado en el interior de obstáculos de grandes dimensiones debido a su falta de información global. En la figura 2.1 se puede observar este problema.
- *Oscilaciones del robot*. Debido a problemas con la captación de información a través del sensor que se utilice, el robot puede interpretar ciertas situaciones de forma errónea que le conducen a realizar movimientos bruscos que se observan en forma de oscilaciones.
- *Deadlock*. Existen situaciones en las que el robot se puede introducir en un lugar estrecho entre dos obstáculos, como por ejemplo un pasillo. Es este tipo de casos, puede ocurrir que el robot interprete mal la influencia que tienen las paredes sobre él y en consecuencia, que se quede atrapado en su interior.

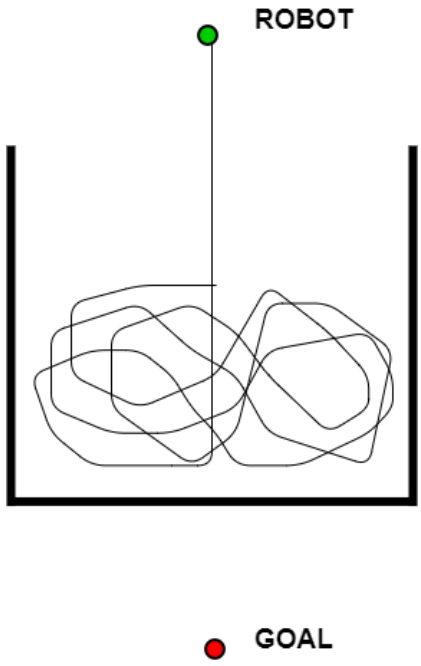


Figura 2.1: Ejemplo de un mínimo local. El robot es incapaz de poder llegar al punto *Goal* debido a la forma en U que tiene el obstáculo.

- *Complejidad computacional*. Al estar en un entorno repleto de obstáculos, el número de posibles decisiones existentes puede ser elevado. Por lo tanto, la complejidad a la hora de realizar operaciones puede ser elevada si el algoritmo que se utiliza no gestiona bien este tipo de entornos.

El algoritmo CG intenta dar solución a esta serie de problemas introduciendo un nuevo esquema local de navegación reactiva basándose en dos trabajos que se realizaron de forma previa [7, 8].

2.2 Introducción al algoritmo

El algoritmo *Closest Gap* (CG) se centra en la búsqueda de *gaps* alrededor del robot. Un *gap* es un espacio libre suficientemente ancho como para que el robot pase por él (figura 2.2).

El algoritmo se inicia determinando todos los *gaps* posibles, para posteriormente descartar los que no son necesarios y así contar únicamente con los potencialmente útiles. Al eliminar los *gaps* que no son necesarios, se consigue reducir la complejidad computacional, las oscilaciones del robot y una mayor suavidad en el camino del robot.

Una vez que se tiene la lista de *gaps*, se elige el *gap* navegable más cercano al objetivo de acuerdo con el campo de visión del robot. Posteriormente, se calcula la dirección que se debe tomar teniendo en cuenta las amenazas (*threats*) que existen alrededor del robot. Con el cálculo de los *threats* se consigue un comportamiento mucho más seguro a la hora de pasar por obstáculos estrechos, además de solucionar el problema de *deadlock* que presenta el algoritmo SND [8].

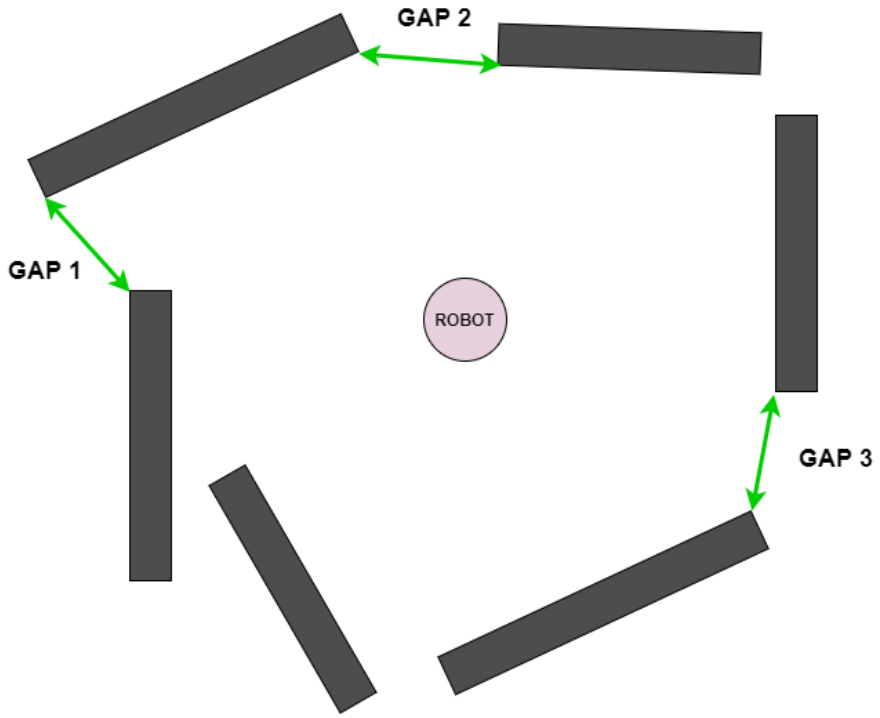


Figura 2.2: Ejemplo donde el robot se ve rodeado de seis obstáculos. Existen seis espacios posibles por dónde el robot puede pasar, pero únicamente los *gaps* señalados son válidos. El resto de espacios se descartan ya que el diámetro del robot es mayor que ellos.

El algoritmo CG es realmente útil a la hora de introducir el robot por *gaps* muy estrechos. Gracias al esquema de navegación reactiva incorporado, se puede realizar movimientos precisos, suaves y con una gran seguridad.

A continuación, se explica de forma detallada las fases principales del algoritmo: análisis de *gaps*, determinación de la dirección de movimiento en función del *gap* seleccionado y la modificación de ésta dirección juntamente con el cálculo de los comandos de velocidad.

2.3 Análisis de *gaps*

En esta primera fase del algoritmo, el objetivo principal es encontrar todos los *gaps* alrededor del robot gracias a la percepción del entorno que nos proporciona el sensor del que se dispone.

Antes de entrar en el detalle de cómo se encuentran los *gaps*, hay que tener en cuenta las siguientes condiciones:

- Se asume la disponibilidad de un vector de n elementos llamado *scan* que especifica la distancia al obstáculo más próximo al robot en una serie de direcciones alrededor del mismo. A través de este vector, es posible conocer el espacio navegable alrededor del robot. Los elementos i, j, k de este vector se referencian por *scan*[i], *scan*[j], *scan*[k], etc.

- El primer elemento de un *scan* es el que se corresponde con el índice 0 y el último es el que se corresponde con el índice (n-1).
- L es la lista de obstáculos detectados mediante el sensor.
- El rango máximo del sensor es d_{max}.
- d(A, B) es una función que devuelve la distancia entre los puntos A y B.
- Se tienen en consideración todos los puntos de escaneo, y no se divide el espacio en sectores.

Para poder realizar el análisis de los *gaps* se necesita:

- La localización del robot (x_{robot}) y el radio del robot (R).
- El rango máximo del sensor (d_{max}).
- Una lista L de obstáculos donde un obstáculo se denota (O_i^L), donde i es un punto determinado del *scan*, y L es la lista de obstáculos alrededor del robot.

Con estos *inputs* se obtiene una lista con los *gaps* detectados.

Para encontrar *gaps* se recorre el *scan* proporcionado por el sensor. A medida que se avanza sobre la lista, se va buscando discontinuidades entre *scans* adyacentes (*scan*[i] y *scan*[j]). Se contemplan dos tipos de discontinuidades:

- Discontinuidad de tipo 1: Ocurre cuando la distancia entre *scan*[i] y *scan*[j] es mayor que el diámetro del robot (véase la ecuación 2.1):

$$\left| d(x_{robot}, O_j^L) - d(x_{robot}, O_i^L) \right| > 2R \tag{2.1}$$

- Discontinuidad de tipo 2: Ocurre cuando uno de los dos *scans* adyacentes, *scan*[i] y *scan*[j], se corresponde con el rango máximo del sensor (véase la ecuación 2.2):

$$\left(d(x_{robot}, O_j^L) = d_{max} \right) \quad AND \quad \left(d(x_{robot}, O_i^L) < d_{max} \right) \tag{2.2}$$

Si j>i, se tiene una discontinuidad ascendente en el *scan*[i]. En cambio, si i>j, se tiene una discontinuidad descendente en el *scan*[j]. Las discontinuidades de tipo 1 tienen prioridad sobre las de tipo 2.

En la figura 2.3 se puede ver un esquema representando todos los conceptos explicados hasta este punto. Del concepto de discontinuidad se verán ejemplos más adelante.

La búsqueda de *gaps* se divide en dos partes: en la primera se encuentran todos los *gaps* alrededor del robot y en la segunda se eliminan aquellos *gaps* que son innecesarios. Para explicar los dos pasos se utilizará la misma figura que en el documento original [1], la cual se reproduce aquí como figura (2.4).

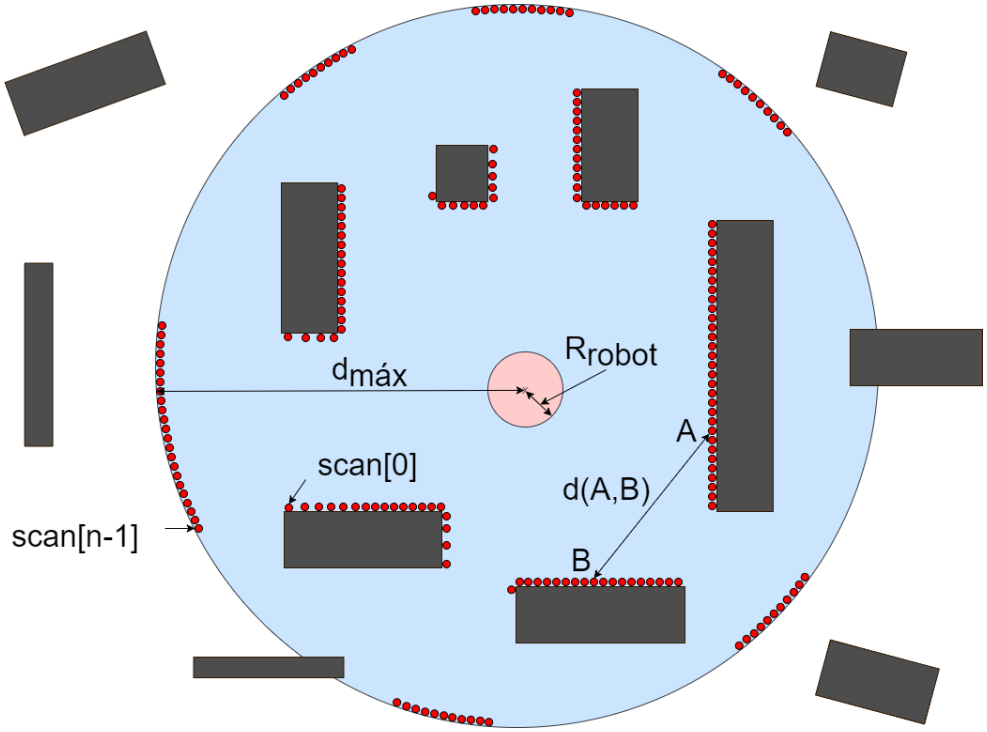


Figura 2.3: Conceptos básicos a la hora de realizar la búsqueda de *gaps*. Cada punto de color azul representa un obstáculo que ha detectado el sensor, y el sombreado de color azul simboliza el rango máximo de visión que tiene el robot con el sensor incorporado.

PASO 1: Primeramente, se buscan discontinuidades ascendentes desde el *scan*[0] hasta el *scan*[n-1] (búsqueda *forward*). Posteriormente, se buscan discontinuidades descendentes desde el *scan*[n-1] hasta el *scan*[0] (búsqueda *backward*).

- Búsqueda *forward*. Suponemos que la primera discontinuidad ascendente ocurre en el *scan*[i]. Este elemento del *scan* es el que determina el primer lado del *gap* (p.e. puntos D y H en la figura 2.4).

  Para encontrar el segundo lado del *gap*, hay que tener en cuenta de qué tipo de discontinuidad se trata:

  - Para una discontinuidad de tipo 1, se define $S^+ = \{i+1, ..., n-1\}$. El siguiente lado del *gap* se encontrará en $j \in S^+$. El punto j que se busca tiene que cumplir dos condiciones: ser el punto más cercano correspondiente a *scan*[i] y no haber tenido que recorrer más distancia angular que π para encontrarlo. Esta condición se expresa formalmente en la ecuación 2.3:

    $$\left(d(O_i^L, O_j^L) \leq d(O_i^L, O_k^L)\right) \quad AND \quad \left(dist_{angular}(\theta_i, \theta_j) \leq \pi\right) \tag{2.3}$$

    donde k es un elemento de S^+, θ_i es el ángulo correspondiente con el primer extremo del *gap* y θ_j es el ángulo correspondiente con el segundo extremo del *gap*.

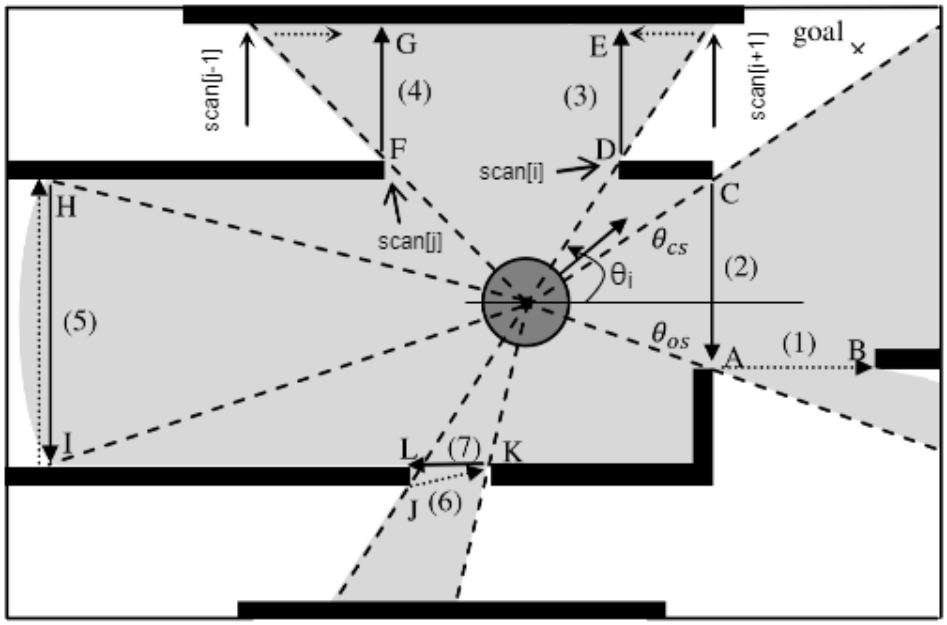


Figura 2.4: Análisis de *gaps* realizado por CG. En el paso 1 se detectan los *gaps* del 1 al 7. En el paso 2 se eliminan los *gaps* 1 y 6. Las flechas de puntos representan *gaps* que se han encontrado y finalmente no se han guardado. De la misma forma este tipo de flecha también representa la distancia angular que se ha recorrido hasta encontrar el punto más cercano al *scan*[i]. El otro tipo de flechas representan los *gaps* encontrados y el sentido de giro utilizado.

El punto E de la figura 2.4 se ha encontrado después de detectar una discontinuidad de este tipo.

– Para una discontinuidad de tipo 2, una vez encontrado *scan*[i] se continua escaneando hasta localizar un *scan*[j] cuyo valor es inferior al rango máximo del sensor. El punto I de la figura 2.4 se ha encontrado después de detectar una discontinuidad de este tipo.

Cada vez que se encuentre un *gap* se reanuda la búsqueda a partir de *scan*[j+1].

Como se puede observar en la figura 2.4, la búsqueda *forward* hace que se encuentren los *gaps* 1, 3, 5 y 6. Los *gaps* 3 y 6 se han encontrado gracias a la detección de discontinuidades de tipo 1. En cambio, los *gaps* 1 y 5 se han encontrado debido a discontinuidades de tipo 2.

- Búsqueda *backward*. En este caso se empieza desde *scan*[n-1] y se termina en *scan*[0]. Se supone que la primera discontinuidad ocurre en *scan*[j]. Se encuentran discontinuidades descendentes y se tratarán de la siguiente forma:

  – Para una discontinuidad de tipo 1, el espacio de búsqueda que se define es $S^- = \{j-1, ..., 0\}$. El primer lado del *gap* se encuentra en $i \in S^-$, el cual tiene que cumplir la ecuación (2.4):

$$\left(d(O_i^L, O_{j+1}^L) \leq d(O_k^L, O_{j+1}^L)\right) \quad AND \quad \left(dist_{angular}(\theta_{j+1}, \theta_i) \leq \pi\right) \quad (2.4)$$

donde k es un elemento de S^-.

Los *gaps* 7, 4 y 2 de la figura 2.4 se han encontrado debido a este tipo de discontinuidad.

– Para una discontinuidad de tipo 2, se escanean todos los puntos anteriores a *scan*[j] hasta encontrar uno cuya distancia sea inferior al rango máximo del sensor. El *gap* 5 se encuentra de nuevo ya que se trata de una discontinuidad de este tipo.

Cada vez que se encuentra un *gap* se reanuda la búsqueda a partir de *scan*[i-1].

En la figura 2.4 se puede ver un claro ejemplo de la prioridad que tienen las discontinuidades de tipo 1 sobre las de tipo 2. Al llegar al punto C mientras se realiza el barrido *backward*, se detectan las dos discontinuidades a la vez, ya que entre el punto C y el siguiente existe una distancia mayor a 2R (discontinuidad de tipo 1), y además el siguiente punto se encuentra en el rango máximo del sensor (discontinuidad de tipo 2). Al ser prioritaria la de tipo 1, se recorre una distancia angular de como máximo π hasta encontrar el punto más cercano al punto C; en este caso, ese punto es el A. Por lo tanto el *gap* 2 está formado por los puntos C y A y no por los puntos C y B.

PASO 2: una vez se han completado los dos barridos (*forward* y *backward*) se obtiene una lista de *gaps* provisional G. De esta lista G se tienen que eliminar:

- Los *gaps* que se encuentran dentro de otros *gaps* (por ejemplo los *gaps* 1 y 6 de la figura 2.4).
- Los *gaps* cuya anchura es menor al diámetro del robot (por ejemplo el *gap* 7 de la figura 2.4).
- Los *gaps* repetidos (por ejemplo el *gap* 5 de la figura 2.4 que se encuentra en el barrido *backward*).

Una vez eliminados los *gaps* descritos anteriormente, se obtiene la lista definitiva de *gaps* (L). Como se comentó en el punto 2.2 , el tener esta lista nos permite reducir el número de *gaps* total en entornos complejos. Esto hace que la probabilidad de tener oscilaciones y la complejidad computacional disminuya.

2.4 Determinar la dirección de movimiento del robot

En esta sección se explica cómo se calcula la dirección hacia la cual el robot se debe orientar en caso de que no haya camino directo libre hacia el objetivo. Para poder saber hacia dónde debe rotar, primero hay que seleccionar el *gap* adecuado. Una vez se tiene la lista final de *gaps* L, se debe elegir de forma angular el *gap* navegable más próximo al punto objetivo. Para elegir este *gap*, se debe hacer lo siguiente: seleccionar el *gap* que

forme un angulo menor con el punto objetivo y comprobar si es navegable o no. Si no es navegable, se elige otro *gap* de la misma forma y el proceso se repite hasta que se encuentra un *gap* navegable.

Cuando se ha encontrado el *gap* definitivo se define cada uno de sus extremos:

- El extremo del *gap* más cercano al punto objetivo se define como cs (*closest side*) y en ángulo se denota θ_{cs}.
- El extremo del *gap* más alejado del punto objetivo se define como os (*outer side*) y en ángulo se denota θ_{os}.

El *gap* elegido en la figura 2.4 es el 2. El punto C define el ángulo θ_{cs} y el punto A define el ángulo θ_{os}.

A continuación, se explica cómo saber si un *gap* es navegable o no (verificación de navegabilidad): si el ángulo del punto objetivo se encuentra dentro del *gap* previamente seleccionado, se comprueba si existe un camino posible hasta el punto objetivo haciendo uso del apéndice A donde se describe parte del algoritmo *Nearness Diagram*. En el caso de que el punto objetivo no se encuentre dentro de la zona angular del *gap*, se calcula de la misma forma la posible existencia de un camino viable hasta el punto medio del *gap*.

Una vez se ha elegido el *gap* al cual se tiene que dirigir el robot, se siguen una serie de pasos para calcular la dirección angular. El ángulo que se busca es θ_{md}(*angle motion direction*). Para empezar, se comprueba si existe un camino directo y navegable hasta el punto objetivo. Si este camino existe, se establece $\theta_{md} = \theta_{goal}$. En caso de que no exista este camino, se querrá navegar hasta el *gap* elegido anteriormente.

Para poder pasar de forma segura a través del *gap*, se necesitan calcular dos ángulos más:

$$\theta_{scs} = \begin{cases} \theta_{cs} - arcsin(\frac{R+D_s}{D_{cs}}), \text{ si } \theta_{cs} \text{ es el lado izquierdo del gap} \\ \\ \theta_{cs} + arcsin(\frac{R+D_s}{D_{cs}}), \text{ si } \theta_{cs} \text{ es el lado derecho del gap} \end{cases} \tag{2.5}$$

$$\theta_{mid} = \begin{cases} \theta_{cs} - dist_{angular}(\theta_{cs}, \theta_{os})/2, \text{ si } \theta_{cs} \text{ es el lado izquierdo del gap} \\ \\ \theta_{cs} + dist_{angular}(\theta_{cs}, \theta_{os})/2, \text{ si } \theta_{cs} \text{ es el lado derecho del gap} \end{cases} \tag{2.6}$$

donde R es el radio del robot, D_s la distancia de seguridad y D_{cs} la distancia que hay desde el centro del robot hasta el extremo del *gap* más cercano al punto objetivo. La interpretación de θ_{scs} y θ_{mid} se incluye a continuación:

- θ_{scs} (ecuación 2.5), surge del ajuste que se realiza al ángulo θ_{cs} para conseguir que el robot se dirija hacia el *gap* sin que la distancia de seguridad entre en contacto con el obstáculo que forma el *gap*. Este ángulo es muy útil a la hora de atravesar *gaps* anchos.
- θ_{mid} (ecuación 2.6), es de especial utilidad cuando se quieren cruzar *gaps* estrechos. Para ello, se ajusta el ángulo θ_{cs} para poder entrar de forma centrada en el *gap*.

Para poder elegir de forma definitiva la dirección de movimiento θ_{md}, se tienen en cuenta los dos siguientes aspectos:

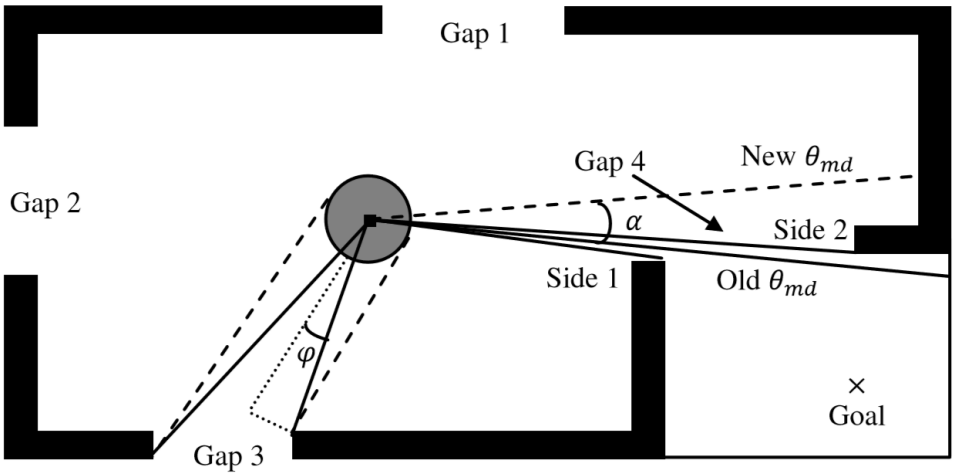


Figura 2.5: Situación en la que se observa la importancia de la modificación que se le realiza al ángulo θ_{md}. De esta forma, el robot se desvia lo suficiente para poder atravesar de forma más segura el *gap* número 4.

- La localización del punto objetivo, ya que si se detecta que θ_{goal} se encuentra entre θ_{cs} y θ_{os} se establecerá que $\theta_{md} = \theta_{goal}$.
- La anchura del *gap*, de forma que se elegirá θ_{scs} si el *gap* es amplio o se eligirá θ_{mid} si el *gap* es estrecho.

En la ecuación 2.7 se puede ver lo que se acaba de explicar:

$$\theta_{md} = \begin{cases} \theta_{goal}, \text{ si } \theta_{cs} \leq \theta_{goal} \leq \theta_{os} \\ \theta_{mid}, \text{ si } dist_{angular}(\theta_{cs}, \theta_{mid}) < dist_{angular}(\theta_{cs}, \theta_{scs}) \\ \theta_{scs}, \text{ en cualquier otro caso} \end{cases} \tag{2.7}$$

Una vez se tiene calculado el ángulo θ_{md}, éste se debe ajustar añadiendo otro denominado α. Al añadir este nuevo ángulo, se pretende conseguir un comportamiento más suave y seguro. En la figura 2.5, se puede ver la utilidad que tiene esta corrección. Entre los cuatro *gaps* que hay alrededor del robot, el elegido es el número 4 debido a que cumple todas las condiciones para ser un *gap* válido: suficientemente ancho para que el robot quepa, el más cercano angularmente al punto *Goal* y también navegable. Aunque el *gap* sea suficientemente ancho, debido a la perspectiva que el robot tiene de él, lo tratará como si fuera estrecho. Por lo tanto, en este caso, el ángulo θ_{md} será igual al ángulo θ_{mid}.

Para calcular el ángulo α se siguen los siguientes pasos:

- Se calcula el ángulo ϕ (véase figura 2.5), el cual es el mínimo para que abarque el radio del robot (véase ecuación 2.8):

$$\phi = arcsin\left(\frac{R}{D_{ns}}\right) \tag{2.8}$$

donde R es el radio del robot y D_{ns} es la distancia desde el centro del robot hasta el extremo del *gap* más cercano a él (punto *Side 1* en la figura 2.5).

- A continuación, se define otro ángulo $\beta = 2\phi$ para poder abarcar el diámetro del robot. La anchura real del *gap* (w) que el robot apreciará será la que aparece en la ecuación 2.9:

$$w = dist_{angular}(\theta_{cs}, \theta_{os}) \tag{2.9}$$

- Una vez calculados β y w, ya se puede proceder a calcular α (véase ecuación 2.10):

$$\alpha = sat_{[0,\beta]}(\beta - w) \tag{2.10}$$

donde la función $sat_{[a,b]}(x)$ se define en la ecuación 2.11:

$$sat_{[a,b]}(x) = \begin{cases} a, \text{ si } x \leq a, \\ x, \text{ si } a < x < b, \\ b, \text{ si } x \geq b. \end{cases} \tag{2.11}$$

La función $sat_{[0,\beta]}(\beta - w)$ retorna valor 0 cuando $w \geq \beta$ (por lo tanto no hace falta modificar θ_{md}), retorna β cuando $w = 0$ (el máximo) y retorna $(\beta - w)$ cuando $0 < w < \beta$.

- Finalmente, se ajusta el ángulo θ_{md} como se muestra en la ecuación 2.12:

$$\theta_{md} = \begin{cases} \theta_{md} - \alpha, \text{ si } D_{ls} < D_{rs}, \\ \theta_{md} + \alpha, \text{ en otra situación.} \end{cases} \tag{2.12}$$

dónde D_{ls} y D_{rs} son las distancias desde el centro del robot hasta los extremos izquierdo y derecho del *gap*, respectivamente.

El ángulo α se le resta o se le suma a θ_{md} para asegurar que el robot vaya hacia el *gap*.

2.5 Modificación de la dirección de movimiento y cálculo de los comandos de velocidad

Después de haber analizado la información sensorial (sección 2.3) y haber obtenido la dirección de movimiento del robot(sección 2.4), el robot debe verse influenciado por los obstáculos que le rodean para así poder evitarlos en pleno movimiento. El cálculo del ángulo θ_{md} no es suficiente, ya que si el robot se dirigiera hacia esta dirección sin tener en cuenta los obstáculos que puedan aparecer de forma repentina, podría colisionar con ellos. Por esta razón, deberá modificarse θ_{md}. El algoritmo CG [1] propone una mejora al método de navegación reactiva que aparece en el algoritmo SND [8]. Esta mejora ayuda

a conseguir trayectorias más seguras y previene la aparición de *deadlocks*. La solución propuesta se describe a continuación.

Cada obstáculo de un grupo de N obstáculos que se encuentre dentro de la distancia de seguridad D_s supondrá una amenaza (*threat*) t_i para el robot, dependiendo de la distancia a la cual se encuentre. El valor de cada *threat* se calcula según la ecuación 2.13:

$$t_i = sat_{[0,1]}\left(\frac{D_s - D_i}{D_s}\right) \tag{2.13}$$

donde D_i es la distancia hasta el obstáculo dentro del rango de visión del robot d_{max}. Por lo tanto, el valor de t_i será 0 cuando el obstáculo esté fuera de D_s y será 1 cuando el robot toque el obstáculo.

Dependiendo de cada *threat* calculado para cada obstáculo, puede aparecer una desviación (*deflection*) que modifica la dirección de movimiento del robot θ_{md} para poder evitar cada punto detectado como obstáculo. En la ecuación 2.14 se puede ver se cómo calcula la desviación δ_i.:

$$\delta_i = t_i \cdot proj(dist_{angular}(\theta_i + \pi, \theta_{md})) \in [-\pi, \pi] \tag{2.14}$$

$$proj(\theta) = ((\theta + \pi) \ mod \ 2\pi) - \pi \tag{2.15}$$

donde la función $proj(\theta)$ (ecuación 2.15) sirve para poder proyectar un ángulo cualquiera dentro del rango $[-\pi, \pi]$, en este caso servirá para calcular la nueva orientación de θ_{md}, en sentido opuesto al ángulo en que se encuentre el obstáculo. El ángulo θ_i se corresponde con la posición angular de cada obstáculo. El valor que se consigue de la función $proj(\theta)$ multiplica al valor de cada *threat* t_i calculado anteriormente para hacer que el valor de la desviación dependa de la distancia a la cual se encuentra cada obstáculo.

El problema principal que presenta el algoritmo SND [8] a la hora de calcular la desviación (*deflection*) total que sufre el robot es que tiene en cuenta todos los *threats* que se encuentran dentro de D_s. Al tenerlos en cuenta todos, si en un lado del robot hay muchos más obstáculos que en otro, al querer pasar por un *gap* muy estrecho, el robot tenderá a colisionar con el lado que hay menos obstáculos, ya que la desviación que sufre hacia ese lado es mucho mayor que la que sufre hacia el otro. Para poder solventar este problema se realiza una modificación de la función que calcula los pesos de cada *threat*. Antes de poder calcular la desviación que aporta cada obstáculo, hay que conocer el peso que tiene cada uno. Este peso se calcula según la ecuación 2.16:

$$w_i = \frac{1}{(1 - t_i)^k} \tag{2.16}$$

donde k define la fuerza que tiene cada peso. Incrementando el valor de k se aseguran conductas más seguras a la hora de pasar cerca de obstáculos.

A continuación, para poder calcular la desviación, se divide el espacio en dos regiones: una a la izquierda del robot y la otra a la derecha. El eje central será el ángulo hacia dónde el robot esté orientado. Para calcular la desviación total se tiene en cuenta cada región por separado.

Se supondrá que N_L y N_R son el número de obstáculos que se encuentran dentro de D_s a la izquierda y derecha del robot, respectivamente. Se pueden definir los pesos totales (W) para todos los obstáculos tal como se ve en las ecuaciones 2.17 y 2.18:

$$W_L = \sum_{i=1}^{N_L} w_i \tag{2.17}$$

$$W_R = \sum_{i=1}^{N_R} w_i \tag{2.18}$$

El valor total de la desviación (*deflection*) en cada lado del robot se define como la suma ponderada de todas las desviaciones del lado correspondiente. Se puede ver cómo se calcula en las ecuaciones 2.19 y 2.20:

$$D_L = \sum_{i=1}^{N_L} \frac{w_i}{w_L} \delta_i \in [-\pi, \pi] \tag{2.19}$$

$$D_R = \sum_{i=1}^{N_R} \frac{w_i}{w_R} \delta_i \in [-\pi, \pi] \tag{2.20}$$

El valor de D_L y D_R se tiene que modificar para ajustar la diferencia del número de obstáculos dentro de D_s entre los dos lados del robot. El ajuste que se realiza se puede ver en las ecuaciones 2.21 y 2.22:

$$D_L = \frac{D_L}{P_L} \tag{2.21}$$

$$D_R = \frac{D_R}{P_R} \tag{2.22}$$

donde $P_L = \frac{N_L}{N}$ y $P_R = \frac{N_R}{N}$. Hay que fijar P_L y P_R a 1 cuando N es 0 o N_L o N_R son 0, respectivamente. En la ecuación 2.23 se puede ver una forma genérica de representar esta condición:

$$P_x = 0 \; si \; (N = 0 \;\; OR \;\; N_x = 0) \tag{2.23}$$

donde $x \exists L, R$.

Una vez calculada la desviación de cada lado del robot, se puede calcular la desviación neta. Se calcula como indica la ecuación 2.24:

$$D_{net} = \frac{w_L \cdot D_L + w_R \cdot D_R}{w_L + w_R} \tag{2.24}$$

Para conseguir una navegación más segura, la dirección de movimiento del robot θ_{md} calculada anteriormente se modifica con la desviación neta que se acaba de calcular para así obtener la dirección angular definitiva a la que el robot se dirigirá(véase ecuación 2.25):

$$\theta_{traj} = \theta_{md} - D_{net} \tag{2.25}$$

Una vez calculada la dirección de movimiento final, se tendrá que calcular tanto la velocidad lineal como la velocidad angular del robot. El objetivo final del algoritmo es conseguir estos tres valores (dirección de movimiento, velocidad lineal, velocidad angular) para así poder tener un control total sobre el robot en todo momento.

A continuación se explica cómo se calcula cada una de las velocidades:

- Para la velocidad lineal, se tiene en cuenta la distancia entre el robot y el obstáculo que se encuentra más cerca. Se supone que d_{min} es la distancia entre el obstáculo más cercano y el radio del robot. La máxima velocidad lineal del robot debe estar limitada según la ecuación 2.26.

$$v_{limit} = \sqrt{1 - sat_{[0,1]}\left(\frac{D_{vs} - d_{min}}{D_{vs}}\right)} \cdot v_{max} \tag{2.26}$$

  donde v_{max} es la velocidad máxima que puede alcanzar el robot y D_{vs} es la distancia de seguridad.

  La velocidad lineal del robot se define finalmente a través de la ecuación 2.27:

$$v = sat_{[0,1]}\left(\frac{\frac{\pi}{4} - |\theta_{traj}|}{\frac{\pi}{4}}\right) \cdot v_{limit} \tag{2.27}$$

- Para la velocidad angular (véase ecuación 2.28), se tiene en cuenta la velocidad angular máxima que puede alcanzar el robot:

$$w = sat_{[-1,1]}\left(\frac{\theta_{traj}}{\frac{\pi}{2}}\right) \cdot w_{max} \tag{2.28}$$

Si en algún momento θ_{traj} apunta en sentido totalmente opuesto a la dirección de movimiento actual del robot, primero se gira $\frac{3\pi}{4} rad$. Posteriormente, cuando se encuentra dentro de la zona de los $\frac{\pi}{4} rad$ restantes, el robot empieza a avanzar con una velocidad proporcional a su alineación con θ_{traj}.

CAPÍTULO 3

IMPLEMENTACIÓN DEL ALGORITMO CG Y DESARROLLO DE SU INTERFAZ ANDROID

En este capítulo se aborda la implementación realizada del algoritmo CG. Se entrará en detalle en el proceso realizado para conseguir que funcione sobre un robot simulado, así como la visualización del correcto funcionamiento del algoritmo.

Durante el transcurso del capítulo se analizará la estructura general del sistema, la implementación que se ha realizado del algoritmo y se realizará una explicación de los métodos y funciones creados. Posteriormente, se explica como se ha realizado la interfaz gráfica en Android. Para finalizar, se comenta la existencia de diversos problemas surgidos durante la implementación.

3.1 Estructura general del sistema

Para comprender cómo funciona el sistema implementado, se requiere de una explicación de los componentes que forman el sistema general. El nodo ROS, MORSE y la aplicación Android son los tres componentes principales que dan vida al proyecto. Es necesario entrar en detalle en cada uno de ellos antes de explicar cómo se interconectan.

3.1.1 MORSE

Este es el simulador que se ha utilizado durante el transcurso del proyecto. En él se ha podido visualizar cómo se comporta un robot en distintos entornos. Sobre el robot se han simulado dos sensores: un sensor láser y un sensor de posición. También se ha simulado un actuador, en este caso es el que permite establecer los valores de velocidad lineal y angular.

Para crear una simulación, se necesita un fichero en código *Python*. En este fichero se detalla la información correspondiente al robot utilizado, los sensores incorporados y el

entorno en el que se situará al robot. Para poder crear los entornos 3D en los que navega el robot se ha utilizado la herramienta *Blender*, la cual permite al usuario crear entornos en tres dimensiones para posteriormente utilizarlos en un simulador como MORSE.

A continuación, se detalla información relevante sobre cada elemento que forma la simulación:

- El robot utilizado ha sido un robot terrestre llamado ATRV(All Terrain Robot Vehicle). En la figura 1.4 se puede ver una imagen de él. Se pueden modificar parámetros como su masa, la fricción que sufre con el suelo o la posición y orientación iniciales.

- El sensor láser es la pieza indispensable para poder implementar el algoritmo CG, es el que nos da información en todo momento sobre el entorno en el que se encuentra el robot. El sensor utilizado es el láser *Hokuyo*, configurable a través del fichero *Python* de tal forma que se puede modificar el arco que abarca (en radianes), el rango máximo (en metros), la resolución (en radianes) y la frecuencia de muestreo (en Hercios).

- El sensor de posición proporciona las coordenadas (x, y, z) en las que se encuentra el robot en todo momento. Este sensor funciona de forma global, es decir, indica en qué posición se encuentra el robot en función de las coordenadas de origen del entorno en el que se encuentra.

- *Blender* es necesario para poder crear los escenarios utilizados para validar la implementación del algoritmo. Es una herramienta de diseño 3D en la cual se pueden crear multitud de obstáculos. Al utilizar un entorno creado con *Blender*, éste nos proporciona las coordenadas iniciales (de origen) para que el robot pueda tener una referencia espacial en todo momento.

MORSE, al tratarse realmente de un nodo ROS, de forma automática se encarga respectivamente de las publicaciones y suscripciones de la información de los sensores y actuadores incorporados al robot. En el siguiente punto se explica en detalle cómo se implementan los tópicos en el nodo para que MORSE funcione correctamente.

3.1.2 Nodo ROS (*Robot_Controller.cpp*)

En el nodo implementado en ROS reside el algoritmo CG. Se ha utilizado el mismo nodo creado en el proyecto [3], reutilizando parte de la estructura que ya estaba implementada. Este punto se centra en las suscripciones y publicaciones que se han llevado a cabo en el nodo para hacer posible el intercambio de información entre el algoritmo y el robot. En secciones posteriores, se explicará cómo se implementa el algoritmo en el nodo, cómo se intercambia información con Android y todas las funciones y métodos que se han creado para el correcto funcionamiento del programa.

Debido a que solo se tienen tres vías de comunicación con el nodo MORSE (sensor de posición, sensor láser y actuador de velocidad), el sistema de comunicación que se establece es sencillo. En la figura 3.1 se puede ver cómo se interconectan los dos nodos gracias a los *topics*.

Para poder recibir la información de los sensores del robot, se utilizan las funciones *spinOnce()* y *Callback(type message & var)*. MORSE publica información sobre el láser y

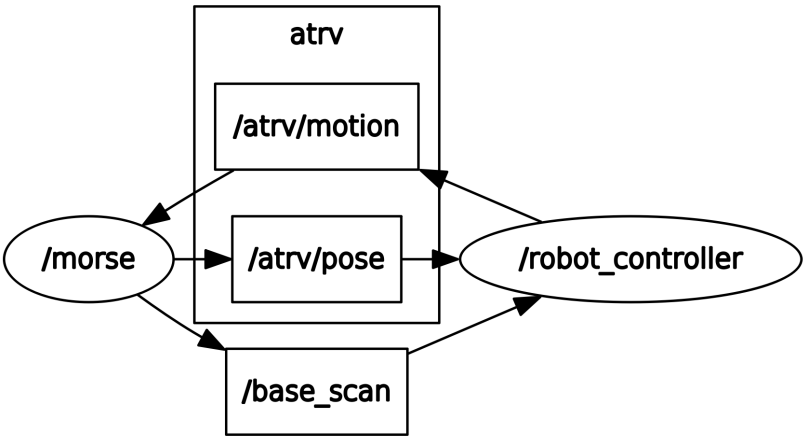


Figura 3.1: Conexión entre el nodo ROS y el nodo MORSE. Se puede observar cómo se interconectan a través de los *topics /atrv/motion*, */atrv/pose* (pertenecientes al robot ATRV) y */base_scan* (perteneciente al láser *Hokuyo*).

la posición del robot de forma constante. A nivel de código se utilizan estas dos funciones para poder recibir la información. Al utilizar la función *spinOnce()* se comprueba si se han publicado nuevos mensajes de cualquier suscripción existente. Si eso ocurre, se ejecutan las funciones *Callback* pertinentes para poder trabajar con la información que se ha recibido. En nuestro caso se utilizan dos *Callbacks*:

- *void* ***laserCallback****(const sensor_msgs::LaserScan::ConstPtr & laser_msg)*, para poder recibir la información que MORSE haya publicado del láser. Se recibe una mensaje de tipo *sensor_msgs::LaserScan* el cual contiene toda la información sobre el láser.
- *void* ***poseCallback****(const geometry_msgs::PoseStamped::ConstPtr & pose_message)*, que sirve para poder recibir información sobre la posición del robot, en este caso el mensaje es de tipo *geometry_msgs::PoseStamped*.

Para poder enviar a MORSE la información sobre la velocidad, se tiene que usar el publicador que se ha declarado. Primero se tiene que guardar la información sobre la velocidad lineal y angular en un mensaje de tipo *geometry_msgs::Twist*. Posteriormente, se utiliza la función *publish()* para publicar el mensaje.

De esta forma, mediante el método publicador/suscriptor, se puede crear un flujo de información entre el robot (nodo MORSE) y el algoritmo implementado (nodo *Robot_Controller.cpp*).

3.1.3 Android

Esta sección se centra en la forma como la aplicación Android se comunica con el nodo *Robot_Controller.cpp*. De la misma forma que se ha conservado la estructura original que tenía el nodo, también se ha conservado la forma a través de la que la aplicación y el nodo se comunican. En el proyecto [3] se explica cómo se implementa y utiliza una comunicación cliente-servidor.

Este tipo de comunicación se basa en la programación de *sockets*. Un *socket* designa un concepto abstracto por el cual dos programas pueden intercambiar cualquier flujo de datos [9]. Al utilizar un *socket* para intercambiar información también se necesita un cliente y un servidor.

En este caso, el cliente está programado en la aplicación Android y el servidor en el nodo ROS. Además, cabe destacar que se trata de una conexión *multithread* y que, por lo tanto, se podría conectar más de un cliente (varios dispositivos Android) de forma simultánea.

Para poder programar el cliente se utiliza la clase *Socket()*. Con el uso de esta clase se puede definir de forma sencilla el *socket* para poder establecer la conexión y así enviar o recibir información.

En el caso del servidor, se hace uso de un conjunto de librerías para poder crear el *socket*. En la sección 3.3 se entrará en detalle en los pasos a seguir para utilizar un *socket*.

3.1.4 Esquema de interconexión

Una vez explicado de forma general cómo funciona cada pilar del proyecto (nodo ROS, aplicación Android y nodo MORSE), se ha realizado un esquema sencillo para aclarar la conexión que existe entre cada elemento. En la figura 3.2 se puede ver tal esquema.

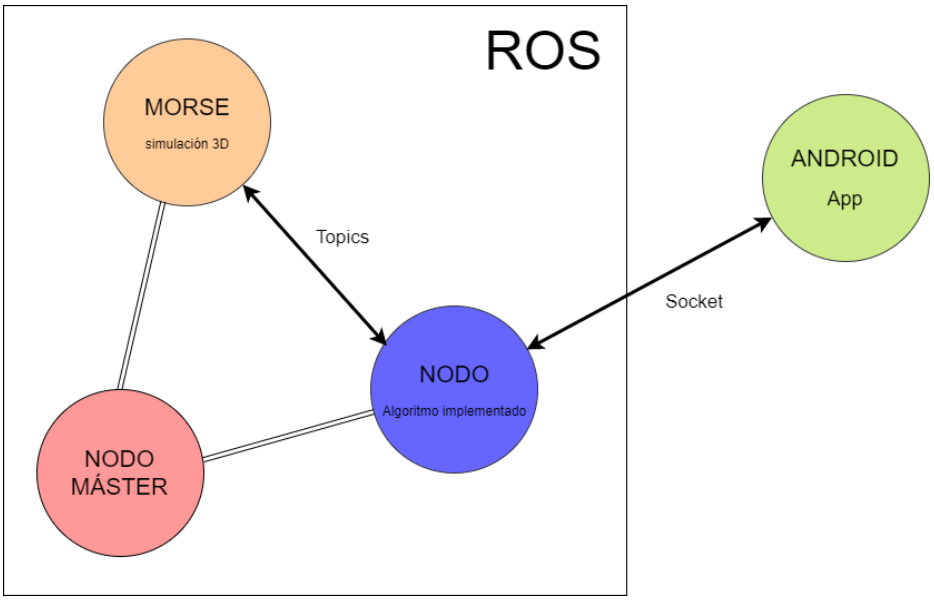


Figura 3.2: Conexión global entre el nodo ROS, MORSE y la aplicación Android.

3.2 Incorporación del algoritmo CG al nodo ROS

En el proyecto [3], el nodo ROS está compuesto por una estructura en la que de forma cíclica se ejecuta la función *int Server()*. Cada vez que se ejecuta, crea un nuevo *socket*; si se crea de forma correcta se ejecuta otra función llamada *void* ***connection_handler**(void *socket_desc)*. Esta función es la que se encarga de recibir información de la aplicación Android. En esta misma función es dónde se ha decidido incorporar el algoritmo CG.

Se ha decidido implementar el algoritmo de forma "ajena" al nodo ROS, es decir, que no se encuentre implementado en él de forma directa. Se ha realizado una clase llamada *algoritmo_cg*, compuesta por una cabecera (*algoritmo_cg.h*) y el desarrollo de la clase (*algoritmo_cg.cpp*). De esta forma, el algoritmo puede ser usado de forma genérica como una librería por cualquier usuario que la desee utilizar.

Durante el transcurso de esta sección se utilizan funciones para explicar cómo se implementa el algoritmo en el nodo *Robot_Controller.cpp*. En la sección 3.3 se comentará cómo funcionan los métodos y funciones que residen en esta clase.

Para poder utilizar el algoritmo, primero se debe tener constancia de las variables que se necesitan para su correcto funcionamiento. Se ha decidido crear una estructura (*struct*) pública en *algoritmo_cg.h* llamada *config* para poder almacenar todas las variables necesarias para utilizar el algoritmo. En la tabla 3.1 se puede ver la definición de esta estructura. Una vez se tengan todas las variables definidas y guardadas en la estructura, se puede hacer uso de la función pública *textvoid init(struct config init_config)*. Ésta se encargará de inicializar las variables internas de la clase para la utilización de todas las funciones que contiene.

Cabe destacar que dentro de la cabecera de la clase existen dos estructuras más. Se han creado con el objetivo de poder compactar información de una forma sencilla. Las estructuras son las siguientes:

- *struct info_scan*: se ha creado con el objetivo de poder guardar toda la información referente a la búsqueda de *gaps*. Se guarda la siguiente información: primer y segundo extremo de cada gap, la distancia a cada extremo, la anchura del gap y la cantidad de gaps que se han encontrado.

- *struct point*: se ha creado con el simple objetivo de definir un punto de forma más sencilla. Contiene las coordenadas x e y del punto que se desee almacenar.

De la misma forma que se necesitan variables de entrada para que se pueda utilizar el algoritmo, también se necesitan variables de salida para poder guardar los resultados que nos proporcione. Estas variables serán la velocidad lineal y angular del robot. Una vez se conozca su valor, se realizarán las publicaciones pertinentes y así el nodo MORSE podrá simular el movimiento del robot. En la tabla 3.1 también se puede ver la definición de estas variables.

Una vez declaradas todas las variables comentadas, se puede hacer uso del algoritmo CG. Los pasos que se han seguido se detallan a continuación:

- Declaración de variables de entrada y salida.

- Uso de la función *ros::spinOnce()* para poder consultar los *Callbacks* y así actualizar los valores del láser y de la posición del robot.

- Asignación de valor a todas las variables e inicialización de la clase utilizando la función *void **init**(struct config init_config)*.

- Análisis de *gaps*, utilizando el método *void **step1**(struct info_scan *_scan, bool type_scan)* en modo *forward* y *backward*. Posteriormente, se utiliza el método *void **step2**(struct info_scan *list_of_gaps, struct info_scan *scan_forward, struct info_scan *scan_backward)* para eliminar los *gaps* innecesarios.

Variables de entrada (*struct* config)	
Variable	**Descripción**
float R	Radio del robot
float Ds	Distancia de seguridad
float k	Constante que define los pesos en la navegación reactiva
sensor_msgs::LaserScan laser	Mensaje que recoge toda la información del láser
struct point X_goal	Punto objetivo
struct point X_robot	Posición del robot
float orientation_robot	Orientación del robot
float v_max	Velocidad lineal máxima
float w_max	Velocidad angular máxima
float Dvs	Distancia segura de velocidad
Variables de salida	
Variable	**Descripción**
float vels[2]	Velocidad lineal y angular, respectivamente

Tabla 3.1: Variables de entrada y salida que se utilizan en el nodo ROS para poder utilizar la clase creada. Las variables de entrada están dentro de una *struct* para poder facilitar su uso.

- Cálculo del el ángulo θ_{md} con el método *void* ***calculate_md****(struct info_scan *list_of_gaps)*.
- Cálculo de la velocidad lineal y la velocidad angular con el método de navegación reactiva *void* ***navigation_method****(float *velocidades)*.
- Una vez se conoce el valor de las velocidades que ha calculado el algoritmo, se puede realizar la publicación de estos valores con la función *publish()*.

3.3 Métodos del algoritmo

Como se ha comentado en el apartado 3.2, existe una clase dónde se ha implementado el algoritmo CG. Esta clase está formada por elementos públicos y privados. Cuando una clase se utiliza en un programa determinado (como en el caso de nuestro nodo ROS) sólo se debe poder tener acceso a los elementos imprescindibles. Estos se definen como públicos para que se puedan utilizar de forma externa. Los elementos que se utilizan para realizar cálculos internos en la clase se definen como privados; así, el usuario no debe preocuparse de ellos a la hora de utilizar la clase. A continuación, en el fragmento 3.3 se pueden observar los métodos y funciones que posee la clase:

```
private:
```

```
double d2r(double angle_in_degrees);
double r2d(double angle_in_radians);

float distBetween2Points(float x1, float y1, float x2, float y2)
   ;
float distBetween2Points(struct point P1, struct point P2);

float minDistBetween2Angles(float angle1, float angle2);
float minDistBetween2AnglesWithSign(float angle1, float angle2);

bool angle_inside_2angles(float angleX, float angleY, float
   angleZ);

void points_xy_gaps(float *xp_gaps_i, float *yp_gaps_i, float *
   xp_gaps_j
, float *yp_gaps_j, struct info_scan *list_of_gaps);

bool appendixA_ND(struct point _X_robot, struct point _X_goal,
 struct point *_list_obstacles, int n_obstacles);
bool appendixA_ND_D(struct point _X_robot, struct point _X_goal,
 struct point *_list_obstacles, int n_obstacles);

float f_sat(float a, float b, float x);

float convert02PI (float angle);

float f_proj(float angle);

std::string ConvertFtS (float number);
std::string ConvertItS (int number);

std::string calculate_xy(struct info_scan *info_scan);

public:

void init(struct config init_config);
void initFile();

void reserv_memory(struct info_scan *scan_forward, struct
   info_scan
 *scan_backward, struct info_scan *list_of_gaps);
void liber_memory(struct info_scan *scan_forward, struct
   info_scan
 *scan_backward, struct info_scan *list_of_gaps);

void step1(struct info_scan *_scan, bool type_scan);
void step2(struct info_scan *list_of_gaps, struct info_scan *
   scan_forward,
 struct info_scan *scan_backward);

void calculate_md(struct info_scan *list_of_gaps);
void navigation_method(float *velocidades);
```

```
geometry_msgs::Twist msg_MORSE(float *velocidades);
int msg_app(char *buffer, struct info_scan *list_of_gaps);
```

Se puede ver cómo los métodos públicos son aquellos que componen cada fase del algoritmo CG. En cambio, los privados son los que se utilizan de forma auxiliar para que cada fase funcione de forma correcta.

A continuación, se detallará la operativa de cada función para conocer cómo se ha implementado el algoritmo. Se ha decidido separar en dos partes distintas los métodos que componen el algoritmo, por un lado aquellos que se han utilizado de forma auxiliar para realizar funciones específicas dentro del algoritmo y, posteriormente, aquellos que forman parte de cada fase del algoritmo y que se han explicado en el capítulo 2. También se ha referenciado cada método con su correspondiente fragmento de código. Primeramente se explican los métodos auxiliares debido a que los métodos que componen las fases del algoritmo los necesitan para poder funcionar correctamente.

3.3.1 Métodos auxiliares del algoritmo CG

Los métodos auxiliares son fundamentales para el correcto funcionamiento del algoritmo. Se utilizan de forma interna en los distintos métodos que forman las fases del algoritmo CG. En la definición de la clase constan como funciones privadas; es decir, son propias del algoritmo y no se puede acceder directamente desde el nodo ROS.

A continuación se detallan los métodos auxiliares utilizados:

- *double* ***d2r****(double angle_in_degrees) [véase fragmento B.3] y double* ***r2d****(double angle_in_radians)*[véase fragmento B.3] se encargan de realizar la conversión de grados a radianes o de radianes a grados, respectivamente.
- *float* ***distBetween2Points****(float x1, float y1, float x2, float y2) [véase fragmento B.3] y float* ***distBetween2Points****(struct point P1, struct point P2)*[véase fragmento B.3] sirven para conocer la distancia que existe entre dos puntos cualesquiera del espacio. Se puede elegir la forma en la que se pasan los puntos: en coordenadas individuales o con la estructura específica para crear puntos.
- *float* ***minDistBetween2Angles****(float angle1, float angle2)* [véase fragmento B.3] se utiliza para poder calcular la distancia angular mínima que existe entre dos ángulos cualesquiera. La distancia angular que devuelve esta función es en valor absoluto.
- *float* ***minDistBetween2AnglesWithSign****(float angle1, float angle2)* [véase fragmento B.3] aparte de calcular la distancia angular mínima entre dos ángulos, tiene en cuenta el sentido del recorrido para que sea el ángulo mínimo. El sentido horario se designa como ángulo negativo y el sentido antihorario como ángulo positivo. Hay que tener en cuenta que *angle1* se considera como el ángulo de partida. Por ejemplo, si los dos ángulos de entrada son respectivamente 233° y 190°, la función debe devolver -43° ya que para cumplir la mínima distancia angular el recorrido se hace de forma horaria.
- *bool* ***angle_inside_2angles****(float angleX, float angleY, float angleZ)* [véase fragmento B.3] sirve para saber si *angleX* se encuentra angularmente entre *angleY* y *angleZ*.

- *void* ***points_xy_gaps****(float *xp_gaps_i, float *yp_gaps_i, float *xp_gaps_j, float *yp_gaps_j, struct info_scan *list_of_gaps)* [véase fragmento B.3] se utiliza para calcular las coordenadas de los extremos de cada *gap* encontrado.

- *bool* ***appendixA_ND****(struct point _X_robot, struct point _X_goal, struct point *_list_obstacles, int n_obstacles)* [véase fragmento B.3] sirve para saber si se puede navegar de forma segura hasta un punto determinado del espacio. Los detalles de funcionamiento se encuentran en el apéndice A. Cabe destacar la utilización del tutorial [10] para saber si una recta y una circunferencia intersectan o no.

- *bool* ***appendixA_ND_D****(struct point _X_robot, struct point _X_goal, struct point *_list_obstacles, int n_obstacles)* [véase fragmento B.3] se trata de una modificación del método *bool appendixA_ND(struct point _X_robot, struct point _X_goal, struct point *_list_obstacles, int n_obstacles)* para poder saber si un punto cualquiera del espacio es directamente navegable. La realización de este método se debe a que en la tercera fase del algoritmo CG, se requiere saber si el punto objetivo es directamente navegable; es decir, si se puede viajar desde la posición en la que se encuentra el robot hasta el punto objetivo en línea recta sin colisionar con ningún obstáculo.

- *float* ***f_sat****(float a, float b, float x)* [véase fragmento B.3] implementa la función $sat_{[a,b]}(x)$ explicada en el capítulo 2. Se describe en la ecuación 2.11.

- *float* ***convert02PI****(float angle)* [véase fragmento B.3] se encarga de convertir un ángulo negativo que se encuentra en el rango [0,-2π] a su correspondiente ángulo en positivo en el rango [0, 2π]. Por ejemplo, si el ángulo de entrada es -173°, la función lo convertirá al ángulo 187°.

- *float* ***f_proj****(float angle)* [véase fragmento B.3] implementa la función $proj(\theta)$ que se explicó en el capítulo 2. Se describe en la ecuación 2.15.

- *std::string* ***ConvertFtS****(float number) [véase fragmento B.3] y std::string* ***ConvertItS****(int number)*[véase fragmento B.3] sirven para convertir una variable de tipo *Float* a una de tipo *String* y viceversa.

- *std::string* ***calculate_xy****(struct info_scan *info_scan)* [véase fragmento B.3] sirve para poder calcular las coordenadas de todos los elementos que se visualizarán en la interfaz gráfica de Android.

3.3.2 Métodos del algoritmo CG

Una vez mencionadas las funciones auxiliares que se utilizan durante las fases del algoritmo CG, en este punto se explica la implementación realizada de cada fase que compone el algoritmo además de la inicialización y la creación de los mensajes que se deben enviar. Cabe destacar que no se entrará en excesivo detalle respecto a como se ha implementado el algoritmo a nivel de código. Nuevamente se ha decidido referenciar cada método para poder visualizar su contenido en caso de que sea necesario. Estos métodos forman la parte pública de la clase; es decir, se puede acceder a ellas desde el nodo ROS para así poder implementar el algoritmo CG.

Inicialización del algoritmo.

Para inicializar el algoritmo, se utiliza la estructura que se ve en la tabla 3.1. La inicialización de las variables se realiza en el nodo ROS y posteriormente se utiliza el método *void **init**(struct config init_config)* B.3 para asignar un valor a las variables de la clase. De esta forma, todos los métodos y funciones podrán utilizar los elementos de la estructura.

Análisis de *gaps*: Fase 1.

En la función *void **step1**(struct info_scan *_scan, bool type_scan)* B.3 se implementa el primer paso (sección 2.3) del algoritmo CG. Los parámetros de la función son:

- Un puntero a una estructura de tipo *info_scan* en la que se almacenará la información que se obtenga al aplicar esta fase del algoritmo.
- Un *booleano* para determinar si se quiere aplicar la búsqueda en modo *forward* o en modo *backward*.

En el fragmento 3.3.2 se puede encontrar una descripción en pseudocódigo de este método. Se compone principalmente de un bucle que recorre el *scan* proveniente del sensor láser. Dentro del bucle se buscan las discontinuidades. Es importante notar que las discontinuidades de tipo 1 son prioritarias a las de tipo 2; por lo tanto se separan los dos casos con una sentencia *else if*. Al encontrar una discontinuidad se procede a seguir recorriendo *scans* hasta encontrar el segundo extremo del *gap*.

```
STEP 1 CLOSEST GAP

- Declaración de variables e inicialización

- Comprobar si es tipo backward para girar todos los puntos del
   láser

for (laser_scans) {

  Calcular d, el resto hasta llegar al total de scans.

  if (discontinuidad de tipo 1) {

    for (hasta d) {

      Buscar la distancia mínima para encontrar el segundo lado
         del gap

      Guardar posible gap

    }

  } else if (discontinuidad de tipo 2) {

    for (hasta d) {
```

```
      Buscar primer punto que no tenga valor de rango máximo del
          láser

      Guardar posible gap

    }

  }

}

- Guardar información en la estructura dependiendo de si es
    forward o backward

- Si es backward volver a dejar en el orden original los puntos
    del láser
```

Una vez se ha acabado de recorrer el *scan*, se procede a guardar la información sobre los *gaps* encontrados dentro de la estructura de tipo *info_scan*.

Cabe notar que cuando se realiza el *scan backward* se debe invertir el *scan*; de esta forma se puede utilizar el mismo bucle que en el *scan forward*. Al acabar el escaneo se deben dejar los *scans* en su orden original para el uso que le puedan dar futuros métodos.

A continuación se mencionará la funcionalidad de algunas variables importantes que aparecen en el fragmento de pseudocódigo 3.3.2:

- *laser_scans* es la cantidad de puntos que nos ha devuelto el láser. Por ejemplo, si se utiliza un láser con una ventana de 270° y con una resolución de 1°, *laser_scans* valdrá 270 ya que es la cantidad de puntos que el láser devuelve. Otro ejemplo sería la utilización de un láser con una ventana de 360° y una resolución de 0.5°, en este caso *laser_scans* valdría 720 ya que el láser devolvería el doble de puntos.

- d es el número de posiciones del *scan* que se deben recorrer después de encontrar una discontinuidad de tipo 1. En el capítulo 2, se explicó que al encontrar una discontinuidad de este tipo se debe recorrer una distancia angular de π radianes y buscar el punto más cercano al primer extremo del *gap*. Por lo tanto, d servirá para que, cuando al llegar a una posición cuya distancia con *laser_scans* es menor que π, se reajuste la distancia a recorrer entre posiciones para así no sobrepasar el valor de *laser_scans*.

Análisis de *gaps*: Fase 2.

En el método *void **step2**(struct info_scan *list_of_gaps, struct info_scan *scan_forward, struct info_scan *scan_backward)* B.3 se implementa la segunda parte de la búsqueda de *gaps*. Se eliminan los *gaps* innecesarios que se han encontrado en la primera parte. Se puede ver cómo los parámetros del método son: la estructura tipo *info_scan* llamada *list_of_gaps* para poder guardar los *gaps* finales y las dos estructuras que se han calculado en el método *step1*, correspondientes a las búsquedas *forward* y *backward*, respectivamente.

En el fragmento 3.3.2 se puede encontrar el pseudocódigo del método. Para poder realizar cada paso de eliminación de *gaps* se utiliza un bucle para recorrer los *gaps* que van sobreviviendo a cada eliminación.

```
STEP 2 CLOSEST GAP

- Declaración de variables e inicializarlas

if (no existen gaps) {

  Registrar en list_of_gaps que no hay gaps

} else {

  Girar índices de los gaps (punto i y j) para que coincidan
     backward y forward

  Fusionar gaps que se han encontrado en forward y backward en
     un mismo array

  Eliminar gaps que se encuentren dentro de otros gaps y gaps
     repetidos

  Guardar gaps resultantes en list_of_gaps

}
```

Determinar dirección de movimiento del robot: *Calculate_md*.

En el método *void* ***calculate_md****(struct info_scan *list_of_gaps)* B.3 se implementa el apartado 2.4 del algoritmo CG. El único parámetro que se necesita es la estructura que contiene toda la información sobre la lista de *gaps* final.

En el fragmento 3.3.2 se puede encontrar el pseudocódigo del método. El elemento principal de esta función es el bucle que va eliminando de la lista de *gaps* aquellos que han sido elegidos previamente pero no han cumplido las condiciones de navegabilidad. Una vez que se elige de forma correcta el *gap* al que se tiene que dirigir el robot, se realizan todos los cálculos mencionados en la sección 2.4 para poder determinar θ_{md}.

```
CALCULATE MD

- Crear e inicializar variables necesarias

- Crear lista de obstáculos

- Comprobar si existe un camino directamente navegable hasta el
   punto objetivo

if (is_direct_navigable == FALSE) {
```

```
Crear coordenadas x e y de los extremos de cada gap

Calcular ángulos de cada extremo de cada gap

Realizar copia de coordenadas y ángulos a variables auxiliares

while (remaining_gaps > 0) {

  Calcular gap más cercano a goal angularmente

  Establecer Ocs y Oos del gap elegido

  Comprobar si θ_goal se encuentra entre θ_cs y θ_os con la función
     angle_inside_2angles

  if (is_inside) {

    Mirar si navegable hasta objetivo con apéndice A_ND

  } else {

    Mirar navegabilidad hasta mid gap

  }

  if (is_navigable) {

    break;

  }else{

    Eliminar gap que se acaba de comprobar y reajustar lista
       de gaps

  }

}

Verificar si Ocs está a la izquierda o derecha del gap y
   calcular Oscs y Omid en
consecuencia

Calcular θ_md según si el gap es ancho o estrecho eligiendo θ_scs
    o θ_mid, respectivamente

Calcular D_ns, ϕ, β y ω

Calcular α en función de β y ω

Reajustar ángulo θ_md con el ángulo α
```

Modificación de la dirección de movimiento y cálculo de los comandos de velocidad: *Navigation_method*.

En el método *float* ***navigation_method****(float *velocidades)* B.3, se implementa el apartado 2.5 del algoritmo CG. El único parámetro que contiene es un vector llamado velocidades, el cual está formado por la velocidad lineal y angular que posteriormente servirá para poner al robot en movimiento.

En el fragmento 3.3.2 se puede encontrar el pseudocódigo de este método. Cabe destacar la división del espacio realizada para poder calcular de forma correcta la desviación que sufrirá el robot. Esta división se ha hecho de la misma forma que en el apéndice A.

```
NAVIGATION METHOD

- Declaración de variables e inicializarlas

- Calcular threats

- Calcular pesos y desviaciones para cada lado del robot

- Calcular desviación neta y desviaciones anteriores

- Modificar θ_md con desviación neta y obtener dirección final de
    movimiento θ_traj (θ_trayectoria)
```

Creación de los mensajes.

Se han implementado los métodos geometry_msgs::Twist **msg_MORSE**(float *velocidades)[véase fragmento B.3] y int **msg_app**(char *buffer, struct info_scan *list_of_gaps)[véase fragmento B.3] para poder crear los mensajes que se envían a través del *topic* y a través del *socket*, respectivamente.

En el caso del mensaje que se debe enviar al nodo MORSE, se utilizará el vector velocidades para asignar el valor al mensaje de tipo *geometry_msgs::Twist* y así posteriormente poder publicarlo.

Para poder crear el mensaje con destino a la aplicación Android, se necesitará la información correspondiente a los *gaps* encontrados para así utilizar la función void **calculate_md**(struct info_scan *list_of_gaps) y calcular todas las coordenadas de los elementos que se desean representar en la interfaz gráfica.

3.4 Interfaz gráfica en Android

Uno de los objetivos que se comentan en el apartado 1.2 es conseguir visualizar de forma local la posición en la que se encuentra el robot. Poder observar el recorrido que realiza el robot nos permite:

- Conocer los obstáculos que el sensor láser detecta.
- Observar el punto objetivo y, en consecuencia, el *gap* por el cual se decide navegar en todo momento.

- Visualizar de forma precisa a qué distancia se encuentra el robot de los obstáculos a la hora de navegar por un *gap* estrecho.

A grandes rasgos, se puede decir que la interfaz gráfica es de gran utilidad a la hora de comprobar el correcto funcionamiento del algoritmo implementado. A continuación, se realizará la explicación en detalle de cómo se ha llevado a cabo la interfaz. Se empezará por la explicación del uso que se le da a la clase *AsyncTask*, el envío de mensajes entre el nodo ROS y la aplicación, y, finalmente, la visualización en la pantalla del dispositivo Android.

3.4.1 Creación del *socket* y clase *AsyncTask*

Tal y como se ha explicado en la sección 3.1.3, la comunicación entre el nodo ROS y Android se realiza a través de *sockets*. A continuación, se explica cómo se crea el *socket* para poder recibir y enviar información.

En el proyecto [3] se implementó la clase *Client*. Esta es una extensión de la clase *AsyncTask*, la cual permite tratar en un segundo plano la recepción y envío de información. Originalmente esta clase solo tenía en consideración el envío de información sobre la fusión de sensores que se realizaba. Añadido a esta funcionalidad, se han implementado los siguientes contenidos:

- Un constructor que permita inicializar de forma sencilla un cliente (un nuevo *socket*).
- Tener en cuenta la recepción de información por parte del nodo, para así poder visualizarla en la pantalla del dispositivo.

Cuando se crea un cliente nuevo, se debe indicar a través de los parámetros del constructor de la clase si se desea enviar o recibir información. El constructor se define como *public* ***Client****(String addr, int port, int _mode)*. Los parámetros son la dirección IP, el puerto y el modo de operación. Se debe utilizar la función *execute()* para que se ejecuten instrucciones en segundo plano. Dependiendo del modo que se haya elegido al crear el cliente, al utilizar la función *execute()*, se enviará o se recibirá información de la siguiente forma:

- En el caso de enviar información, al utilizar el *execute()* dentro de la función *doInBackground()* del cliente se utiliza la función *print()* para poder transmitir el mensaje a través del *socket*.
- Cuando se trata de una recepción, después de utilizar la función *execute()*, se debe usar la función *get()* para guardar el mensaje que se ha tratado dentro de la función *doInBackground()* del cliente.

En general, la clase *AsyncTask* resulta muy útil a la hora de programar aplicaciones multitarea. Por ello, se ha decidido seguir utilizándola ya que con ella se puede trabajar de una forma muy sencilla, evitando así la programación manual de hilos (*threads*) para conseguir una ejecución en segundo plano.

3.4.2 Envío de mensajes

Este apartado se centra en los mensajes que se envían a través del *socket*, para que exista una comunicación entre el nodo ROS y la aplicación Android.

Mensaje de la aplicación hacia el nodo

El envío de información en el sentido aplicación-nodo se realiza para enviar comandos que configuran parámetros del algoritmo. En concreto, se desea enviar la siguiente información:

- Coordenadas del punto objetivo, en formato x y, para definir desde el dispositivo el punto objetivo que el robot debe alcanzar.
- La distancia de seguridad D_s, para definir la mínima distancia a los obstáculos a evitar de forma reactiva.
- El parámetro k del algoritmo, para poder conseguir comportamientos más seguros al modificar la influencia de los obstáculos sobre el robot.
- La distancia segura de velocidad, parámetro necesario para modificar la limitación de velocidad que se le introduce al robot de acuerdo con la ecuación 2.26.
- Velocidad lineal y angular máximas para el robot.

Mensaje del nodo hacia la aplicación

El mensaje que se envía del nodo hacía la aplicación se utiliza para representar de forma gráfica la posición local en la que se encuentra el robot. La información que contiene el mensaje es la siguiente:

- Posición del punto objetivo desde el punto de vista del robot, la cual se necesita conocer para dar la impresión de que el punto objetivo se aproxima, aleja o rota en función del movimiento del robot.
- Orientación del robot, para conocer la dirección de movimiento desde la pantalla del dispositivo.
- La posición de todos los puntos detectados por el láser, para, de esta forma, crear una nube de puntos en la pantalla simulando los obstáculos que el robot percibe en todo momento.
- La posición de los puntos que forman los extremos de cada *gap*, para así dibujar de forma independiente los *gaps*.
- El *gap* elegido para navegar, para destacarlo sobre el resto y poder observar de forma clara hacia dónde se dirigirá el robot en caso de que tenga más de un *gap* a elegir.
- El radio del robot, para visualizar una circunferencia que lo represente.
- La dirección de movimiento final, para visualizar de forma clara hacia dónde se dirige en robot en todo momento (ángulo θ_{traj}).

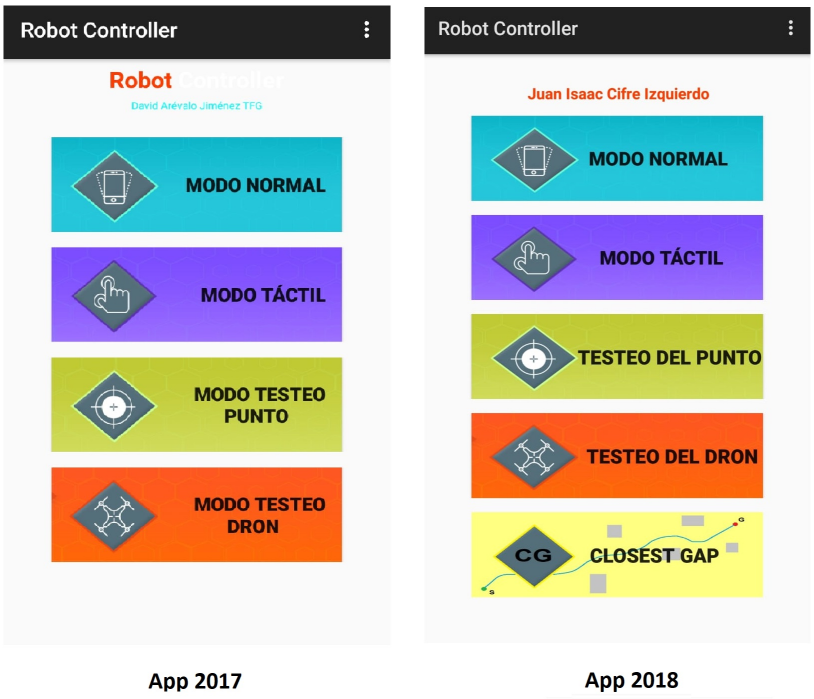


Figura 3.3: Representación de la aplicación una vez se le ha añadido el modo correspondiente interfaz gráfica del algoritmo CG.

Toda esta información se transmite en forma de coordenadas, las cuales se crean en el método *calculate_xy* dentro de *msg_app*. El *string* que devuelve se guarda en un vector y posteriormente se envía gracias a la función *int* ***send****(int fd,const void *msg,int len,int flags)*. Ésta es la función predeterminada de envía de mensajes de la clase *socket*.

3.4.3 Visualización de la interfaz

La interfaz se ha implementado como un modo de operación más a la aplicación ya existente. En la figura 3.3 se puede ver el modo añadido. Cabe decir que este modo sólo se pone en funcionamiento cuando se establece una conexión correcta entre el nodo ROS y la aplicación. Al establecer conexión, el flujo de información se inicia. Se envían de forma constante comandos de la aplicación al nodo para ejecutar el algoritmo y, posteriormente, se envían los resultados del nodo a la aplicación para visualizarlos. Este apartado se centra en cómo se visualiza la información que se recibe en la aplicación.

Como se ha comentado en la sección 3.4.1, para poder recibir la información en la aplicación Android se debe crear un cliente en modo de recepción de información. Una vez se haya creado, se utilizan las funciones *execute()* y *get()* para recibir la información.

Estas dos funciones se han implementado en la clase *CubeRenderer*. Esta clase se ha reutilizado del proyecto [3]. Gracias a ella se puede visualizar figuras en la pantalla del dispositivo gracias al motor *OpenGL*. Una vez recibido el mensaje, se trata cada uno de sus elementos para guardar la información en variables para su posterior visualización. Para dibujar cada elemento de forma correcta sobre la pantalla del dispositivo se ha utilizado las funciones *glTranslatef()* y *draw()*. Gracias a la primera se puede elegir en qué coordenada

de la pantalla dibujar, mientras que con la segunda se puede dibujar el elemento que se desee.

En el caso de nuestro modo de visualización, se ha optado por seguir un estilo minimalista y sencillo. Únicamente se han utilizado puntos y líneas para representar toda la información. De este modo, se consigue visualizar lo que el robot sensoriza de una forma rápida y sencilla.

En la figura 3.4 se puede observar un entorno sencillo creado con *Blender* (figura 3.4a) y el aspecto final de la interfaz gráfica para este caso (figura 3.4b). Mediante este nuevo modo de operación podemos controlar de forma local en qué posición del espacio se sitúa el robot. En la interfaz gráfica disponemos de los siguientes elementos:

- El conjunto de puntos rojos, que representan los obstáculos detectados por el sensor láser.
- Circunferencia interior, que representa la dimensión del robot ATRV.
- Circunferencia intermedia, que representa la distancia de seguridad.
- Circunferencia exterior, que representa el rango máximo detectado por el sensor láser.
- El punto amarillo, que representa el punto objetivo. Si se encuentra fuera del alcance del láser del robot se visualiza fuera de la circunferencia exterior.
- La flecha amarilla, que representa la orientación del robot.
- Los triángulos, que representan los *gaps* navegables, de color verde el escogido para navegar y el resto de color lila.
- La flecha lila, que representa la dirección de movimiento según el algoritmo CG.

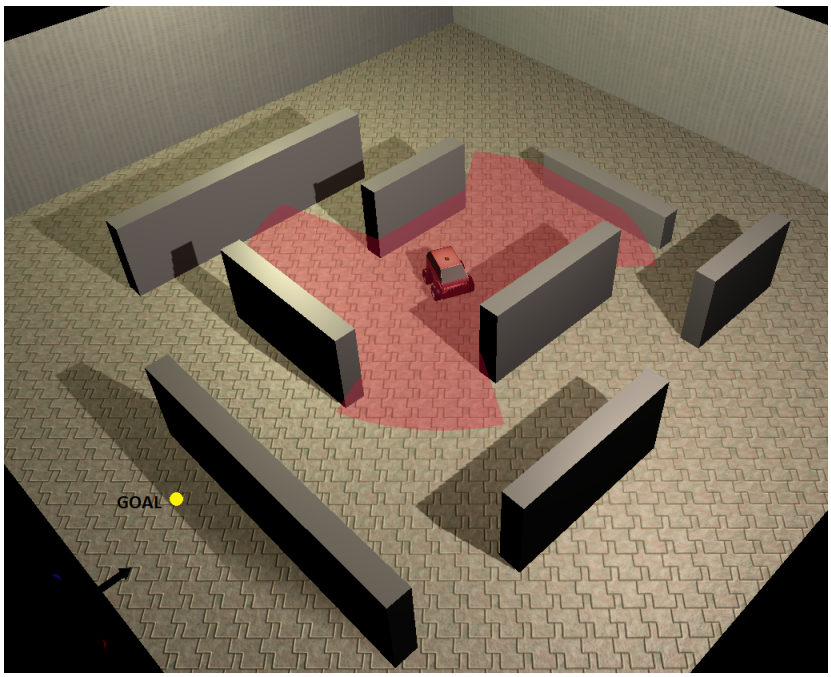
Por otro lado, en la figura 3.5 se puede observar la pantalla de configuración que aparece al presionar los tres puntos situados arriba a la derecha de la interfaz. Se puede elegir cambiar cada parámetro mencionado en la sección 3.4.2. Cada vez que se introduce un nuevo valor, éste cambia de forma automática y se envía a través del *socket* para utilizarlo en el algoritmo.

3.5 Dificultades y problemas encontrados

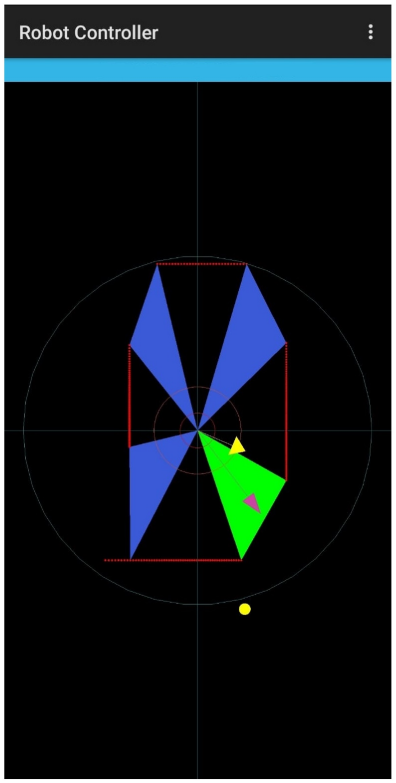
Durante el desarrollo del sistema han surgido diversos problemas, algunos de un mayor peso que otros. Este apartado se centra en la descripción de algunos de los más importantes.

3.5.1 Memoria ocupada por las estructuras

Como se ha visto en la sección 3.2, durante la ejecución del algoritmo CG se utilizan unas estructuras específicas para guardar la información sobre la búsqueda de *gaps*. Dentro de la estructura existen vectores que ocupan una cantidad de espacio determinada. Esta cantidad no es fija ya que depende de la longitud del *scan* que devuelve el sensor láser. Durante las primera pruebas de implementación siempre se utilizaba la misma ventana de



(a) Entorno en MORSE.



(b) Interfaz gráfica.

Figura 3.4: Aspecto final de la interfaz gráfica creada en Android de un entorno visualizado en MORSE.

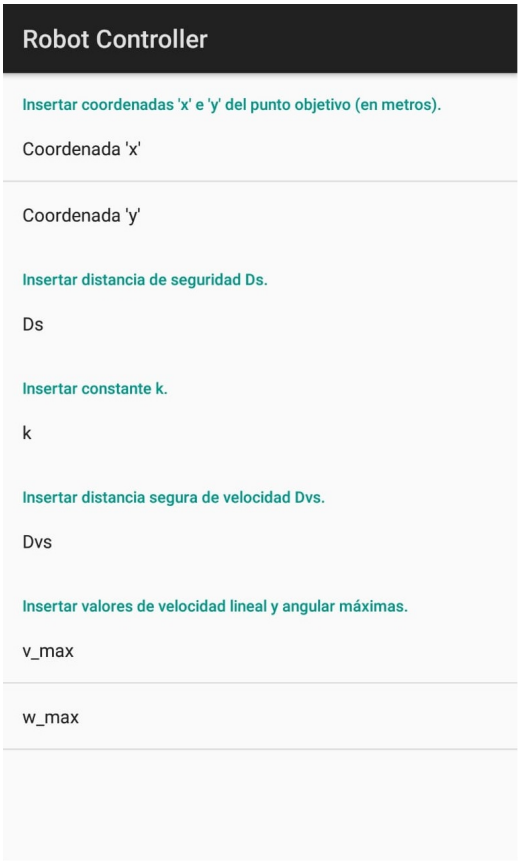


Figura 3.5: Pantalla de opciones dónde se pueden configurar diversos parámetros para poder utilizar el algoritmo CG.

escaneo y la misma resolución para el láser. Por lo tanto, la longitud siempre era fija. Al mejorar el algoritmo y hacer pruebas con distintos parámetros del láser, surgió el problema de que en cada ocasión se tenía que cambiar de forma manual la cantidad de espacio que ocupaban los vectores.

Ya que es obvio que los parámetros del láser se deben cambiar para probar la funcionalidad del algoritmo, se decidió implementar de forma correcta la cantidad de memoria que ocupan estas variables. Se utilizan dos funciones diseñadas específicamente para tratar con este tipo de situaciones:

- La función *void* ***malloc****(size_t size)* se utiliza para reservar una cantidad de memoria determinada. La cantidad que se desee reservar se debe introducir en bytes en el parámetro *size*. Por ejemplo, si el láser tiene una ventana de escaneo de 270° y una resolución de 1°, se deben reservar 270 bytes de memoria por cada variable que la vaya a necesitar.

- La función *void* ***free****(void* ptr)* se utiliza para liberar toda la memoria que se ha reservado con la función anterior. En el parámetro de entrada se debe introducir el puntero correspondiente al bloque de memoria.

Gracias a estas dos funciones se puede cambiar los parámetros del láser, ya que simplemente se ocupará la memoria que se necesite en cada ocasión.

Cabe destacar la creación de los métodos void **reserv_memory**(struct info_scan *scan_forward, struct info_scan *scan_backward, struct info_scan *list_of_gaps) B.3 y void **liber_memory**(struct info_scan *scan_forward, struct info_scan *scan_backward, struct info_scan *list_of_gaps) B.3 para poder solventar el problemas que se acaba de exponer. Utilizando como parámetros las estructuras utilizadas durante el algoritmo, estas dos funciones se encargarán de reservar y liberar la memoria para hacer un uso correcto del espacio de almacenamiento.

3.5.2 Longitud del mensaje enviado del nodo a la aplicación

Como se ha explicado la sección 3.4.2, se envía multitud de información del nodo ROS hacia la aplicación para alimentar la interfaz gráfica. Desde un primer momento y para no complicar el código, se decidió enviar toda la información en un solo mensaje a través del *socket*. El principal problema al tener un solo mensaje es la longitud de éste; es decir, dependiendo del tamaño de las variables enviadas, la longitud del mensaje aumenta o disminuye.

En primera instancia, se decidió fijar la longitud del mensaje. De esta forma, independientemente del tamaño de las variables, en su interior siempre se utilizaría un mensaje suficientemente grande para enviar toda la información. El problema es que, al necesitar un número tan grande de *bytes* para poder satisfacer el máximo tamaño, el envío cíclico de este mensaje provocaba que la recepción fallara constantemente.

Para solventar el problema de forma definitiva, se ha decidido implementar la posibilidad de que el mensaje sea de longitud variable. Cada vez que se envía el mensaje, únicamente se utilizan los *bytes* necesarios. Se utiliza una cabecera localizada de forma fija al principio del mensaje, la cual especifica el tamaño del mensaje. De esta forma, al recibir el mensaje en la aplicación (gracias a la función *read()* y a la cabecera del propio mensaje) se verifica si han llegado todos los *bytes* de forma correcta.

CAPÍTULO 4

INFORME Y ANÁLISIS DE RESULTADOS

En este capítulo se exponen los resultados que se han obtenido al implementar el algoritmo CG. Se empezará con una explicación sencilla de cómo se crean los entornos de simulación, seguido de las simulaciones realizadas en diversos escenarios. Se ha decidido distribuir los escenarios según el comportamiento esperado del algoritmo. De esta manera se puede observar de forma clara las fortalezas y debilidades que presenta.

4.1 MATLAB, MORSE y Blender

MATLAB, MORSE y Blender son las herramientas que se han utilizado para poder llevar a cabo las pruebas realizadas. Para poder completar una simulación se llevan a cabo los siguientes pasos:

- **Creación del entorno con *Blender***. Cada entorno está representado por un plano y un punto de luz. Posteriormente se le pueden añadir elementos con diferentes formas geométricas. En todas las simulaciones se ha cambiado el tamaño, posición y orientación de diversos rectángulos para así simular obstáculos. Una vez finalizado cada entorno, éste se añade en el fichero *Python* que recoge la simulación en MORSE. En la figura 4.1 se puede observar el entorno correspondiente a la figura 2.4, el cual se ha creado a modo de ejemplo. Cabe mencionar que todas las longitudes que se utilizan están declaradas en metros, por lo tanto a la hora de representar los resultados está será la norma que se siga.

- **Ejecución algoritmo sobre el nodo ROS y análisis de cómo se comporta el robot en la simulación que realiza MORSE**. Durante la ejecución del algoritmo se guardan en dos ficheros distintos la posición del robot en cada instante y cada uno de los puntos detectados por el sensor láser para posteriormente poder representarlos.

- **Visualización del resultado en MATLAB**. Se ha utilizado un pequeño programa ya existente al cual se le han añadido diversas modificaciones para adaptarlo a la

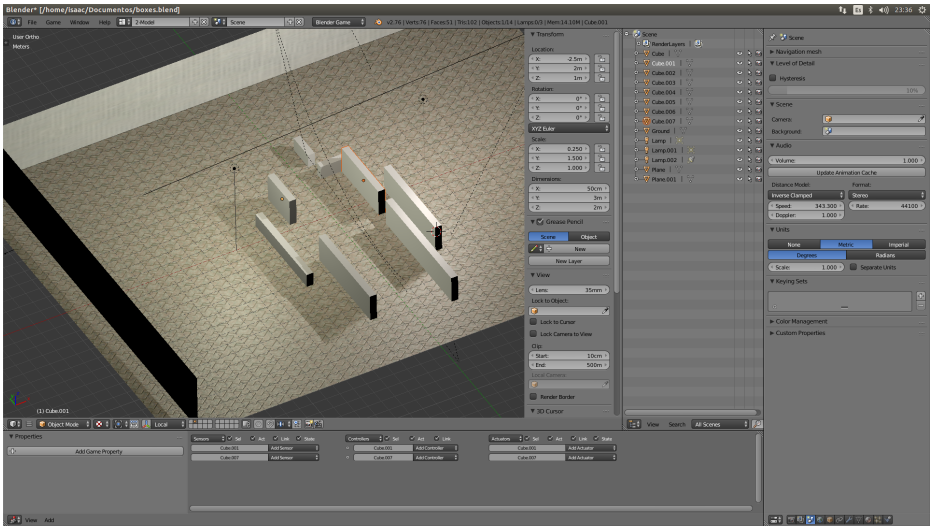


Figura 4.1: Ilustración del uso de Blender como entorno de diseño de escenarios de prueba.

situación requerida. El programa se encarga de representar en un gráfico el recorrido que ha realizado el robot, junto con la detección de obstáculos proporcionada por el sensor láser. De esta forma, se puede comprobar de forma precisa el recorrido que el robot ha seguido y, en consecuencia, verificar que el algoritmo implementado funciona correctamente.

4.2 Escenarios favorables

Como se ha comentado en el capítulo 2, existen diversos motivos por los cuales el algoritmo CG es tan útil. Entre ellos destacan su una buena reactividad, acompañada de movimientos suaves, su gran capacidad para moverse entre multitud de obstáculos y, sobretodo, la facilidad que presenta el robot para navegar entre espacios estrechos. En este apartado se ha querido enfatizar este tipo de situaciones para poder demostrar la validez del algoritmo CG. Se presentarán un total de siete escenarios en los cuales se podrá observar el comportamiento que ha tenido el robot.

Cada escenario se presentará con su entorno en MORSE, seguido de la representación que se ha realizado en MATLAB. En algunos escenarios se presentará la visualización que se observa en los ejemplos del documento [1] para contrastar los resultados.

4.2.1 Escenario 1

Este primer entorno se encuentra representado en la figura 4.2. En este caso, el sensor láser tiene un rango de alcance de diez metros.

El objetivo de esta simulación es que el robot detecte que se trata de un *gap* muy estrecho debido a la visión que tiene de él. Para ello, se necesita que el *gap* sea detectado desde el inicio de la simulación. En la figura 4.3 se puede ver que, en el momento inicial de la simulación, el *gap* detectado, que se resalta en verde, es extremadamente estrecho. Por lo tanto, el robot deberá orientarse de forma adecuada respecto del *gap* para posteriormente

atravesarlo de forma centrada. Cabe destacar la importancia del ángulo α de la ecuación 2.10 para poder modificar la dirección de movimiento del robot.

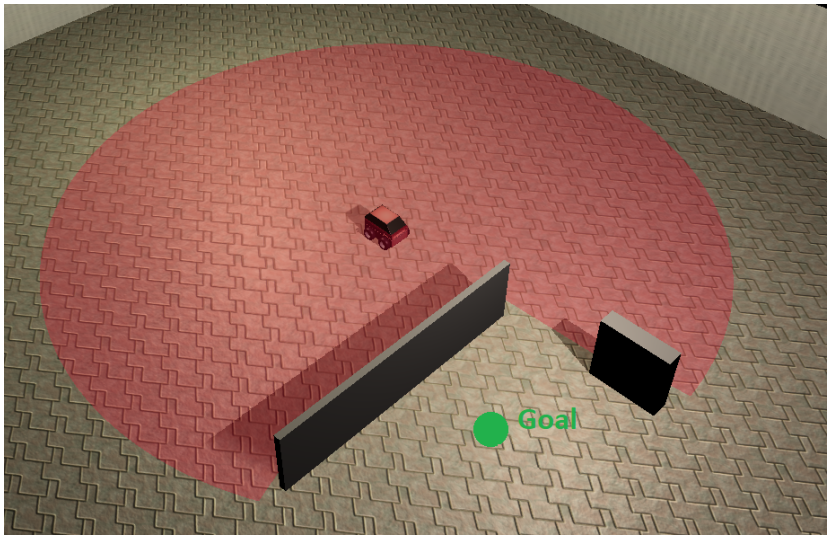


Figura 4.2: Entorno correspondiente al primer escenario favorable. El máximo alcance del láser es de 10 metros y el campo de visión es de 360°.

En la figura 4.4 se puede observar el resultado que ofrece el documento [1] respecto a esta simulación, contrastado con el obtenido por nuestra implementación. Se puede observar como ambas simulaciones son muy parecidas, así como que desde el principio del recorrido el robot empieza a orientarse de forma adecuada para posteriormente atravesar el *gap* sin dificultades.

4.2.2 Escenario 2

Este entorno se puede observar en la figura 4.5. Se puede ver como se necesita de una buena maniobrabilidad para poder pasar por el túnel y llegar al punto objetivo. Al tratarse de un túnel, gracias al *gap* que se encontrará en todo momento y a la influencia que tienen las paredes sobre el robot, se podrá pasar por el túnel de forma segura hasta llegar al punto objetivo.

En la figura 4.6 se compara el resultado obtenido y la ejecución que presenta el documento original. Se puede decir que el recorrido es bastante bueno, transmitiendo una gran sensación de seguridad.

4.2.3 Escenario 3

Este entorno se puede observar en la figura 4.7. Es una modificación que se ha realizado sobre el escenario de la figura 4.5, añadiendo complejidad al trazado. En particular, se puede observar que el espacio de maniobra entre cada *zigzag* dentro del túnel es estrecho, hecho que ayuda a comprobar cómo se comporta el algoritmo en estas situaciones.

En la figura 4.8 se puede observar el resultado obtenido. Se trata de un escenario en el que, en gran parte del recorrido, únicamente existe un *gap*. Por lo tanto, el aspecto que cobra más importancia del comportamiento del robot es la influencia que tienen los

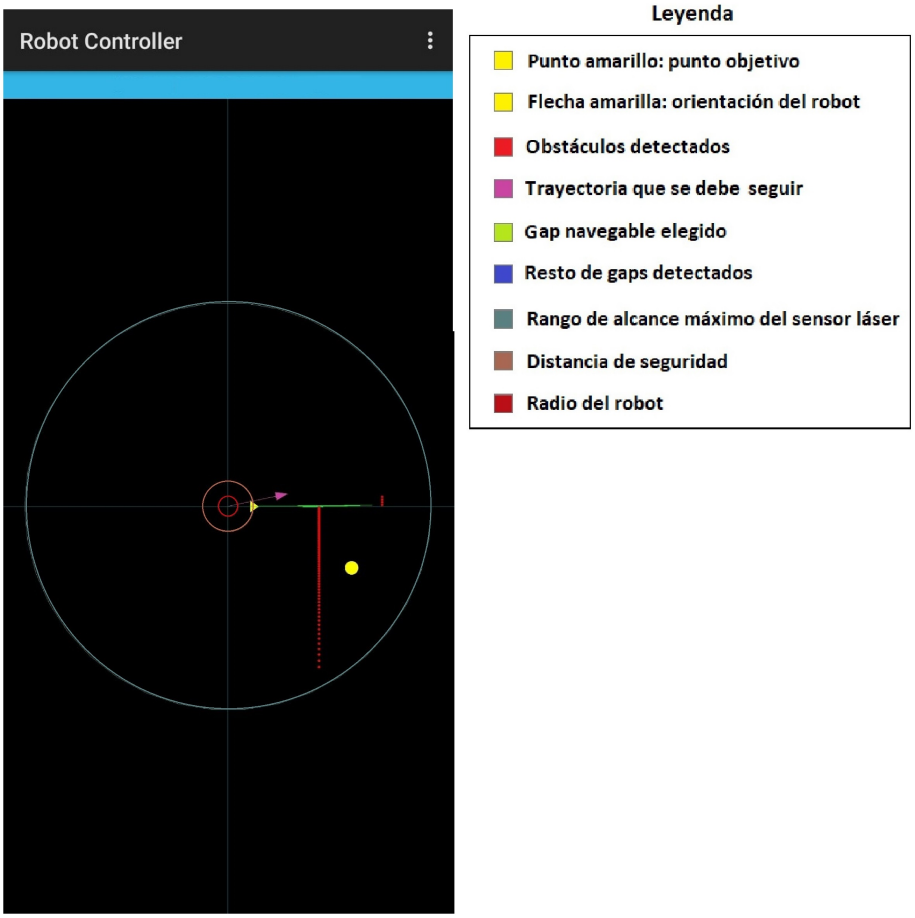


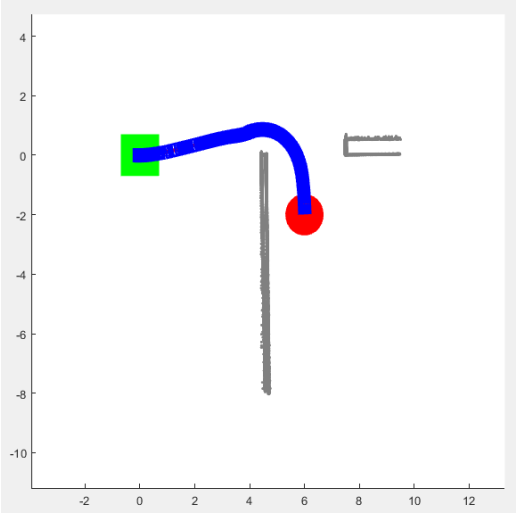
Figura 4.3: Aspecto de la interfaz al inicio de la simulación realizada en el primer escenario. Se puede observar como el *gap* que detecta el robot es muy estrecho desde su ángulo de visión.

obstáculos sobre el mismo para que éste pueda pasar bien centrado entre ellos. Como se puede observar, se cumplen las expectativas, el algoritmo funciona bien para espacios estrechos donde se deben realizar maniobras delicadas.

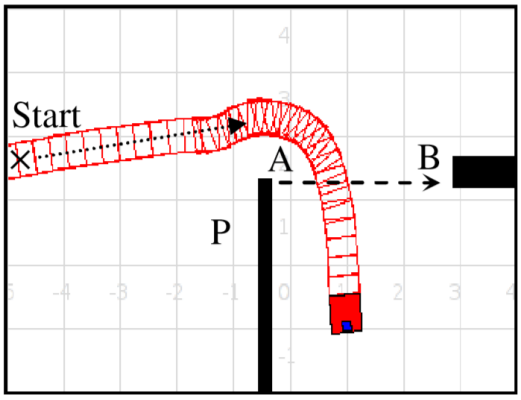
4.2.4 Escenario 4

Podemos ver la representación del entorno en la figura 4.9. Este entorno esta diseñado para que el robot navegue a través del pasillo de extremo a extremo. La complejidad se encuentra en que, al no tratarse de un pasillo lineal, el robot se verá muy influenciado por la posición en la que se encuentra cada obstáculo. Por lo tanto, la dirección de movimiento final irá cambiando en cada momento, haciendo que el robot vaya oscilando de una forma suave para así poder evitar chocar con ningún obstáculo.

En la figura 4.10 se puede ver el recorrido que se ha realizado. Además, en la figura 4.11 se puede encontrar una ampliación de la zona relevante de la figura 4.10. Se puede observar perfectamente como el robot no colisiona en ningún momento con los obstáculos, demostrando así que se consiguen comportamientos suaves y seguros gracias al método



(a)



(b)

Figura 4.4: (a) Comparación entre la simulación realizada y (b) la trayectoria que presenta el documento original del algoritmo CG para el primer escenario.

reactivo implementado. De forma adicional, en la figura 4.12 se puede ver un instante de la ejecución del experimento en la interfaz gráfica, donde se indican los *gaps* que detecta el robot y la dirección de movimiento que calcula.

4.2.5 Escenario 5

La visualización de la escena se encuentra en la figura 4.13. Este entorno se inspira en otro experimento que aparece en el documento [1]. En la figura 4.14, se puede ver el recorrido que el robot ha llevado a cabo juntamente con la trayectoria publicada en el documento original. Se puede ver como los trazados son muy similares.

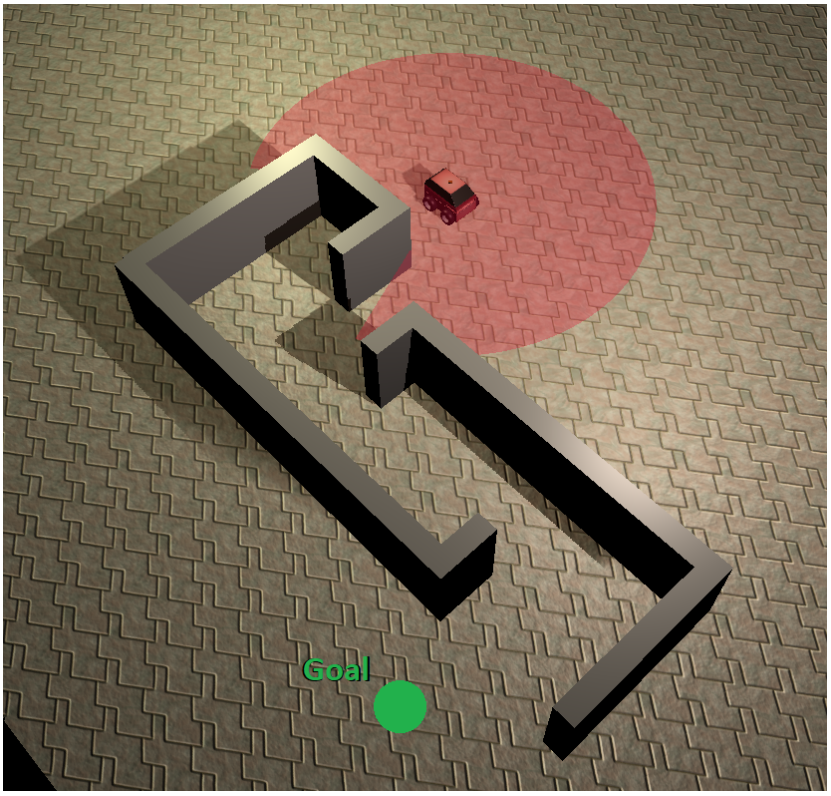


Figura 4.5: . Entorno correspondiente al segundo escenario favorable. El máximo alcance del láser es de 5 metros y el campo de visión es de 360°.

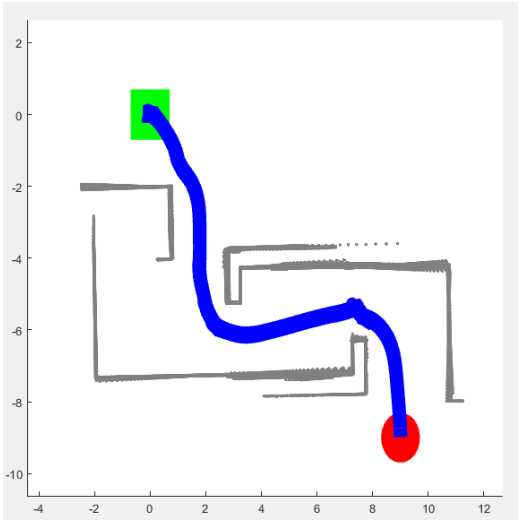
4.2.6 Escenario 6

Se puede observar el entorno en la figura 4.15. Se trata de un escenario repleto de obstáculos. Como se puede apreciar, el objetivo se encuentra en la esquina contraria al punto de partida. Por lo tanto, el robot deberá evitar multitud de obstáculos para llegar a su destino. Este entorno es muy favorable para el algoritmo, ya que, al existir tantos obstáculos, encontrará muchos *gaps* y, por lo tanto, tendrá más posibilidades de elección antes de navegar entre ellos.

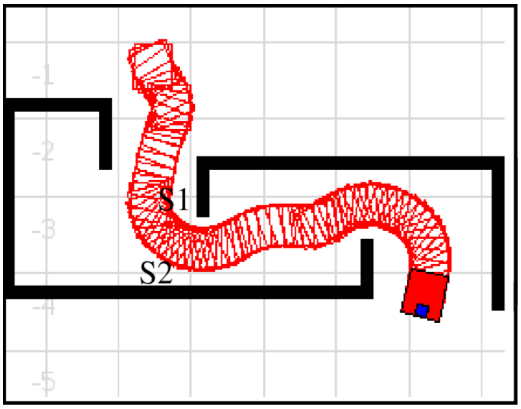
En la figura 4.16, se puede ver el trazado que el robot ha realizado. Por lo general, en este tipo de escenarios, se tenderá a seguir un camino sin demasiadas desviaciones al existir tantos *gaps*. Se ha querido resaltar la zona A en la figura 4.17, ya que es un paso realmente estrecho donde se demuestra la capacidad de poder navegar entre *gaps* pequeños. Se puede ver como el robot no impacta con los laterales y pasa completamente centrado. Adicionalmente, también se puede observar en la figura 4.18 el paso por el *gap* estrecho a través de la interfaz Android.

4.2.7 Escenario 7

Este entorno se muestra en la figura 4.19. El objetivo de este escenario es el de crear una combinación de dificultades para así realizar una simulación más completa. Se pueden observar cuatro fases diferenciadas. La primera se corresponde con un entorno repleto



(a)



(b)

Figura 4.6: (a) Comparación entre la simulación realizada y (b) la trayectoria que presenta el documento original del algoritmo CG para el segundo escenario.

de obstáculos entre los cuales no existe mucho espacio para navegar. Posteriormente se debe pasar por un pasillo formado por repleto de obstáculos; de esta forma el robot deberá corregir de forma continua su dirección de movimiento para no colisionar. La tercera etapa vuelve a ser un espacio lleno de obstáculos. Finalmente, se deberá pasar a través de un pasillo formado por paredes lisas, el cual no es completamente recto para así dificultar la llegada hasta el punto objetivo.

En la figura 4.20 se puede ver la simulación completa obtenida. Se ha decidido ampliar diferentes zonas para poder observar mejor cómo el robot pasa a través de distintos puntos. En la figura 4.21 se puede observar cada zona ampliada. En la zona A se puede ver como el robot evita correctamente la primera tanda de obstáculos y además, pasa por un *gap* estrecho antes de entrar dentro del túnel. La zona B se centra en el túnel y su salida, se puede observar como el recorrido se adapta perfectamente a las imperfecciones del túnel, creando un camino seguro a través de él. La zona C es delicada, ya que, al existir tan

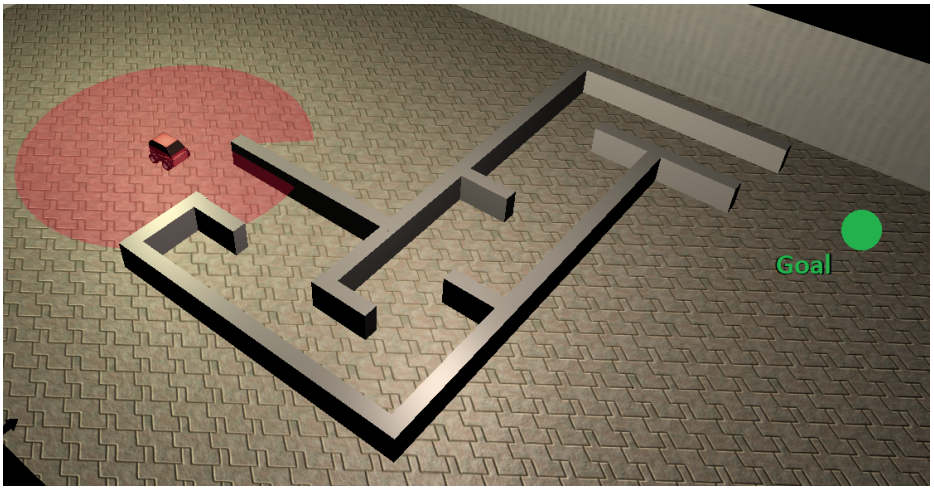


Figura 4.7: Entorno correspondiente al tercer escenario. El máximo alcance del láser es de 5 metros y el campo de visión es de 360°.

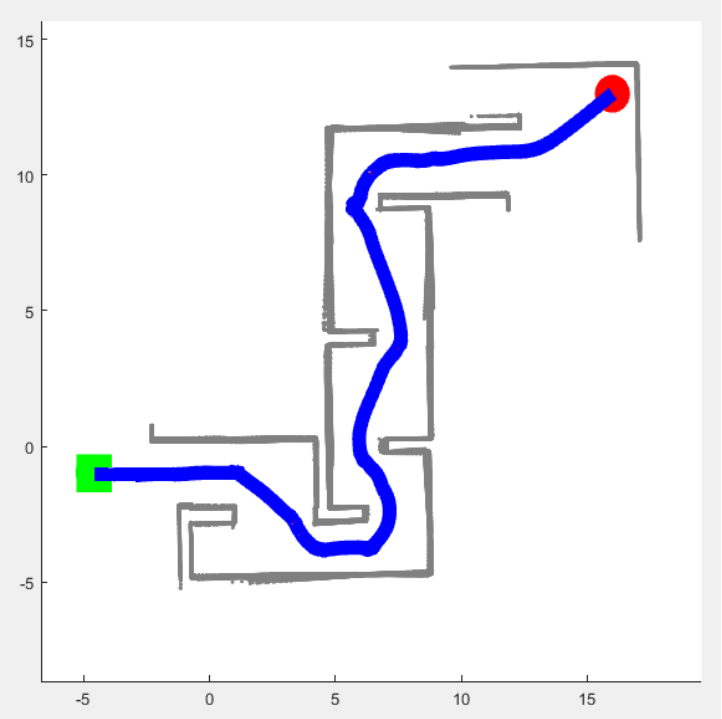


Figura 4.8: Resultado de la simulación realizada en el tercer escenario.

poco espacio entre los obstáculos, el robot se encuentra obligado a pasar por *gaps* muy estrechos en todo momento. Finalmente, en la zona D, se puede ver como el robot pasa a través del túnel con paredes lisas. Este trazado se realiza sin demasiada complicación ya que las dos paredes ejercen la misma influencia sobre el robot; por lo tanto, la trayectoria resultante era de esperar que fuera muy rectilínea.

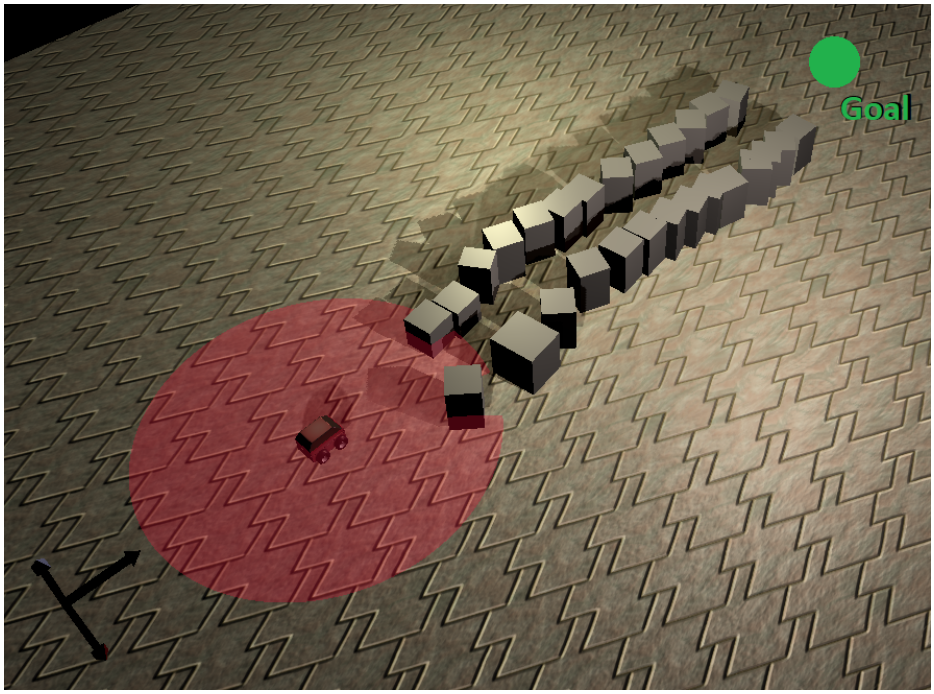


Figura 4.9: Entorno correspondiente al cuarto escenario. El máximo alcance del láser es de 5 metros y el campo de visión es de 360°.

4.3 Escenarios desfavorables

Esta sección se centra en la consideración de entornos que consigan provocar errores o situaciones comprometidas al algoritmo implementado. Al ser un algoritmo reactivo, hay que tener en cuenta que existen situaciones en las que puede suceder que el robot no se comporte de forma que se alcance el objetivo. En concreto, una vez se ha estudiado el algoritmo de forma exhaustiva, se puede notar que las siguientes situaciones pondrían en un serio compromiso al robot:

- Debido a que la mayor parte de decisiones a la hora de calcular el ángulo de movimiento son en base al ángulo que forman los *gaps* detectados con el punto objetivo, el cambio de posición de éste, en función del entorno en el que se encuentre el robot, puede provocar oscilaciones de forma continua en torno a diversos *gaps*. Esto se debe a que la posición del punto objetivo no favorece a la elección de uno en concreto.

- El rango máximo de alcance del sensor láser juega un papel fundamental, ya que dependiendo de su valor, es posible que se detecten demasiados *gaps* a la vez. Este hecho puede dar lugar a que el robot no escoja el más adecuado por su falta de visión global.

- En general los espacios abiertos y los obstáculos de grandes dimensiones no son adecuados a la hora de utilizar este algoritmo. El robot necesita estar rodeado de obstáculos para así encontrar *gaps* y navegar entre ellos. En espacios abiertos con

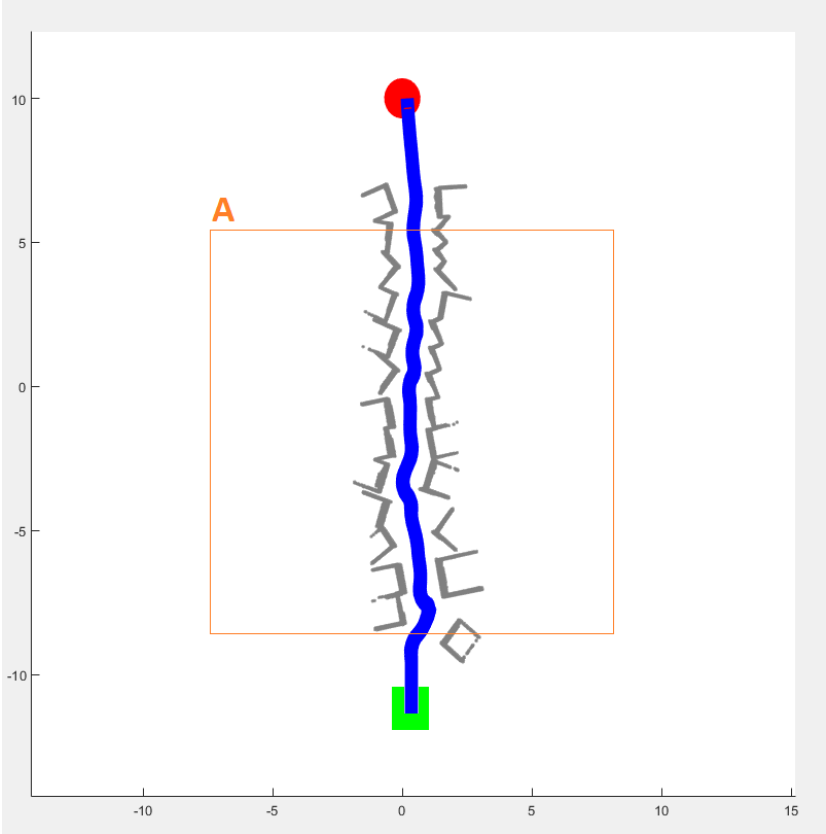


Figura 4.10: Resultado de la simulación realizada en el cuarto escenario. El máximo alcance del láser es de 5 metros y el campo de visión es de 360°.

obstáculos grandes, el robot no encontraría una cantidad suficiente de *gaps* como para poder moverse de forma adecuada.

- Hay que tener en cuenta que como algoritmo reactivo, no llegará a saber si el objetivo es alcanzable o no. Es decir, puede ser que se quede atrapado y oscilando de forma continua hasta que el usuario decida acabar con la simulación.

Una vez explicados los principales problemas que a priori se pueden observar, se presentarán una serie de escenarios para poder ejemplificarlos.

4.3.1 Escenario 1

Este primer escenario se puede observar en la figura 4.22. Se puede ver como se trata de un túnel largo por el cual existe salida por cada uno de los lados. Sería posible llegar al punto objetivo, pero con un algoritmo reactivo como CG no es posible. Primeramente, el robot navegaría hacía uno de los dos lados dependiendo de qué *gap* se encuentre más próximo al objetivo; recorrida una cierta distancia, el *gap* que se encontrara detrás se convertiría en el más próximo angularmente al objetivo. Por lo tanto, daría media vuelta y así se repetiría el proceso sin que el robot pudiera salir del túnel. En la figura 4.24 se puede observar el momento inicial de la simulación, así como que se detectan dos *gaps* a lo largo del túnel.

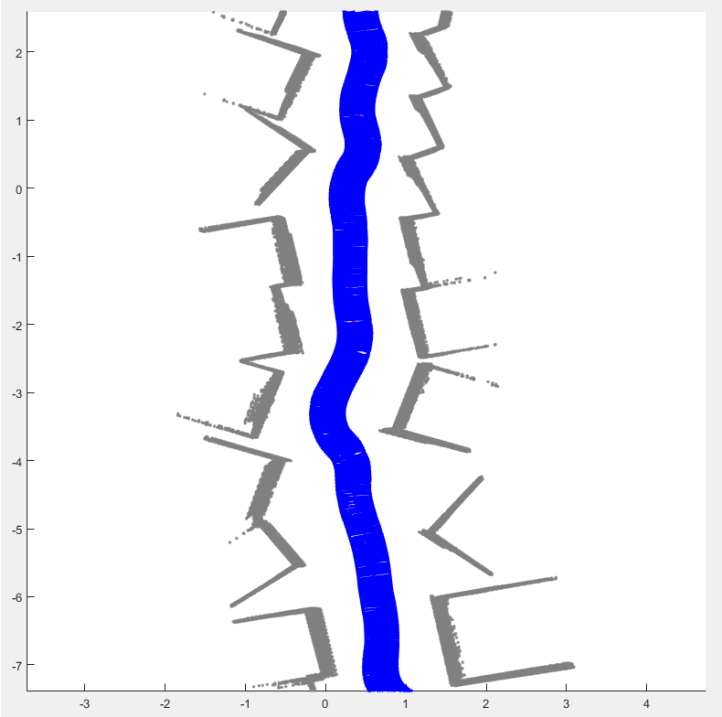


Figura 4.11: Ampliación de la zona A de la figura 4.10.

Hay que notar que una vez el robot se ha orientado hacia el *gap* de la izquierda, él continuará el trazado hasta el final del túnel. Esto es así por que no detecta el *gap* que se encuentra en su retaguardia. Este *gap* no se considera debido al ángulo en el que empiezan los *scans forward* y *backward*. Cuando llega al final del túnel se da cuenta de que vuelve a haber un *gap* que le dirige hacia el interior del túnel y por lo tanto navegará por él. En la figura 4.23, se puede ver el trazado realizado de forma cíclica a través del túnel. Además, en la figura 4.24 se puede ver la detección de un *gap* en la parte derecha del túnel, el cual redirige nuevamente al robot hacia su interior.

4.3.2 Escenario 2

Este segundo escenario se puede observar en la figura 4.25. Se trata de un caso en el que hay un obstáculo de grandes dimensiones y además espacios muy abiertos. El problema principal de querer llegar hasta el punto objetivo es que no se detectan *gaps*. Debido a que se ha considerado que un *gap* es un espacio entre dos obstáculos, no se considera un *gap* el espacio existente alrededor de los obstáculos. El robot se aproximará al obstáculo y posteriormente detectará *gaps* debido a la fugacidad de las tomas del *scan*, y detectará que entre dos puntos existe una distancia mayor a 2R, debido a la posición angular respecto a esos dos puntos. En la figura 4.26, se puede ver los *gaps* que se detectan. Posteriormente, el robot se decantaría por uno de esos dos *gaps*; en nuestro caso ha sido el de su izquierda. Finalmente, el robot se quedará oscilando alrededor de la pared ya que no encontrará más *gaps* y no podrá avanzar (figura 4.27). Se puede ver el recorrido que el robot realiza en la figura 4.28.

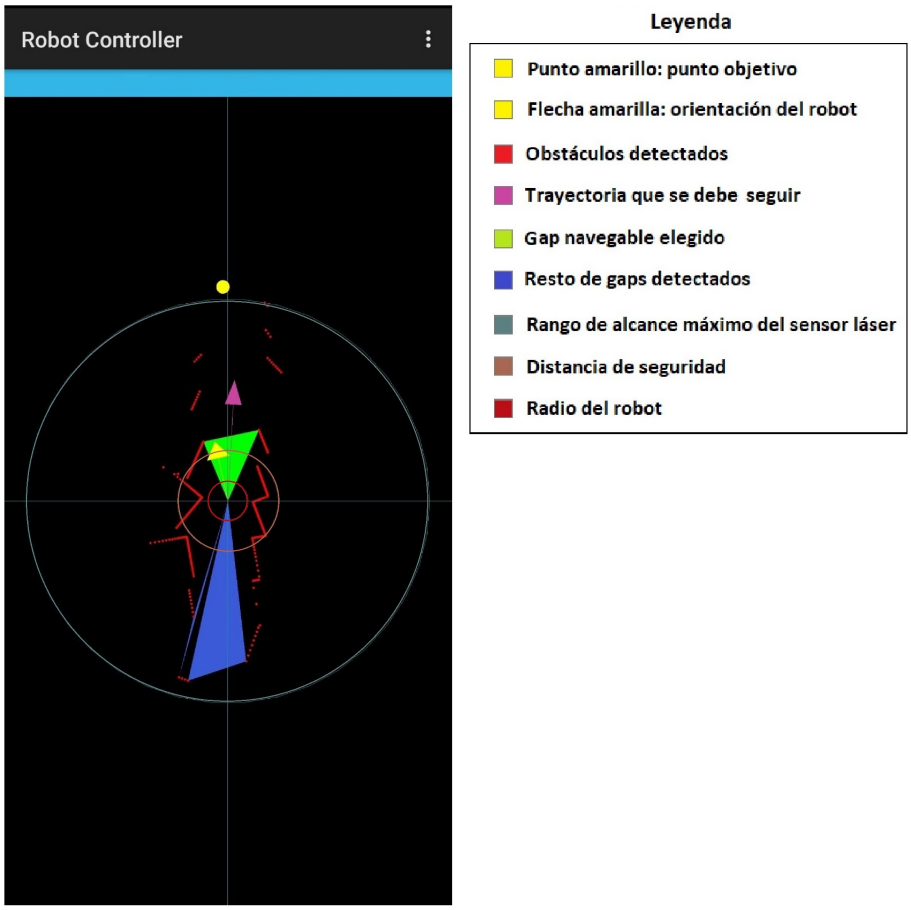


Figura 4.12: Visualización a través de la interfaz gráfica de la percepción del entorno del robot en un momento determinado de la cuarta simulación.

4.3.3 Escenario 3

Este tercer escenario es el mismo que se utilizó en la figura 4.7. En este caso, se ha decidido cambiar la posición del punto objetivo y además se ha eliminado una pared que se encontraba al principio del recorrido. El escenario modificado se puede observar en la figura 4.29. Lo que ocurrirá en este caso es que al llegar al primer giro dentro del túnel, debido a la posición en la que se encuentra el punto objetivo, el robot detectará *gaps* que provocarán pequeñas oscilaciones antes de entrar al túnel (figura 4.30). Una vez que se entre en el túnel, llegará a un punto en el que oscilará de forma continua debido a la detección que realiza del *gap* más próximo al objetivo. En la figura 4.31, se puede ver la simulación realizada. Al entrar al túnel no se aprecia de forma muy notable la oscilación que realiza, ya que enseguida puede decidir correctamente el *gap* por el que navegar. Al llegar al punto que forma una perpendicular con el punto objetivo se quedará oscilando (figura 4.32).

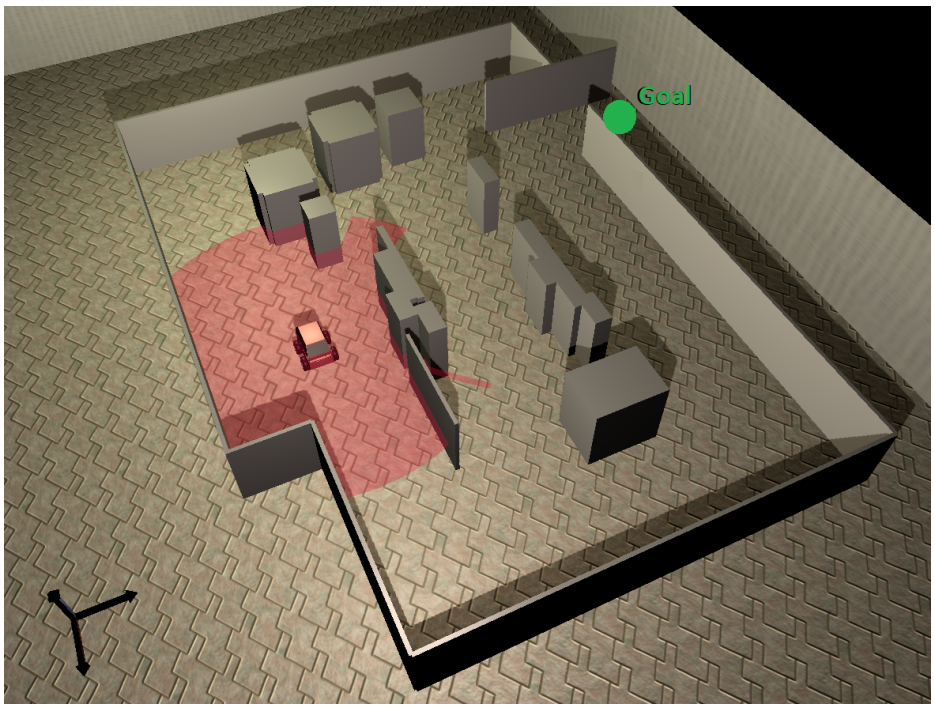
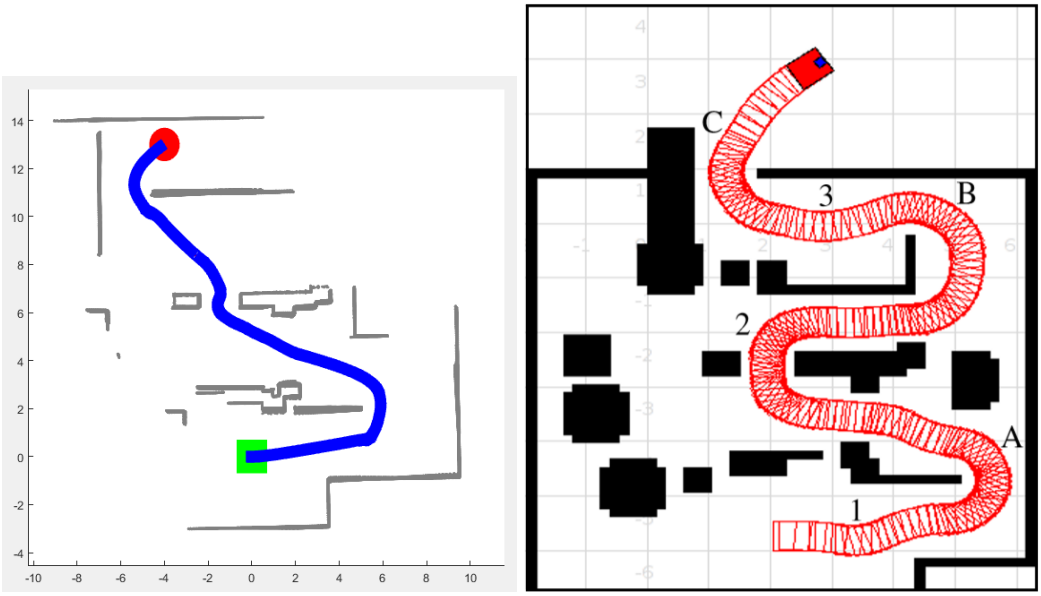


Figura 4.13: Entorno correspondiente al quinto escenario. El máximo alcance del láser es de 5 metros y el campo de visión es de 360°.



(a) (b)

Figura 4.14: (a) Comparación entre la simulación realizada y (b) la trayectoria que presenta el documento original del algoritmo CG en el quinto escenario.

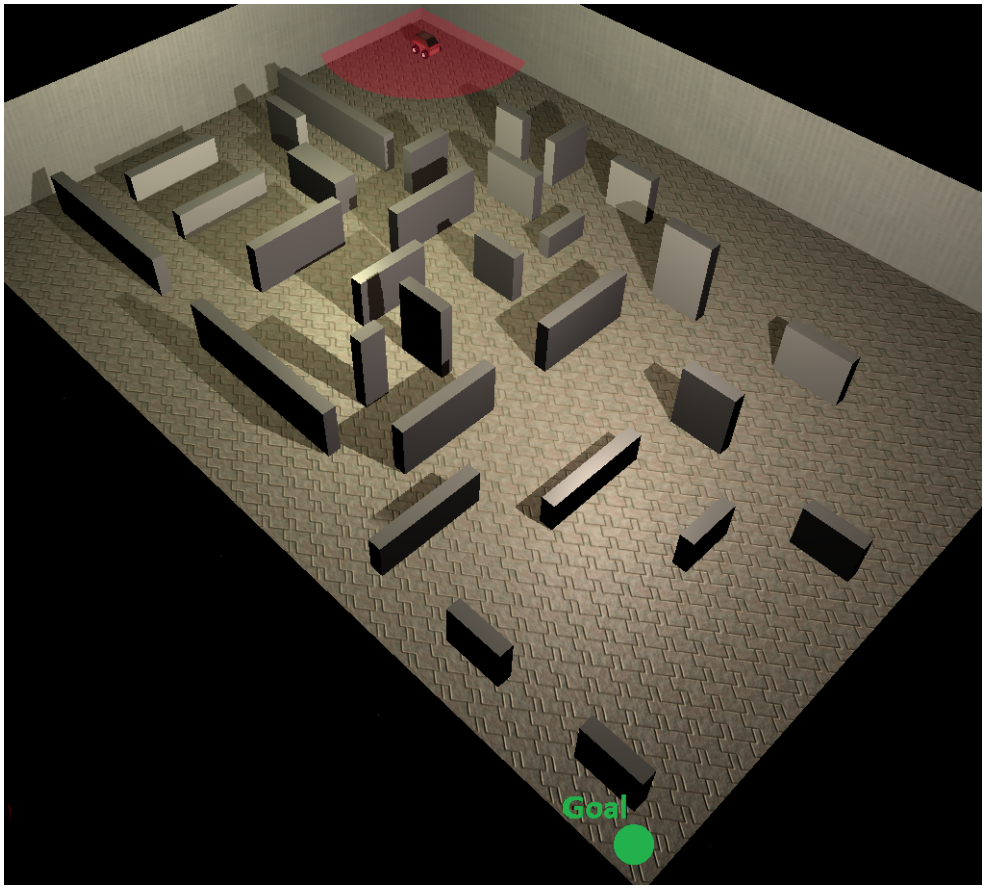


Figura 4.15: Entorno correspondiente al sexto escenario. El máximo alcance del láser es de 5 metros y el campo de visión es de 360°.

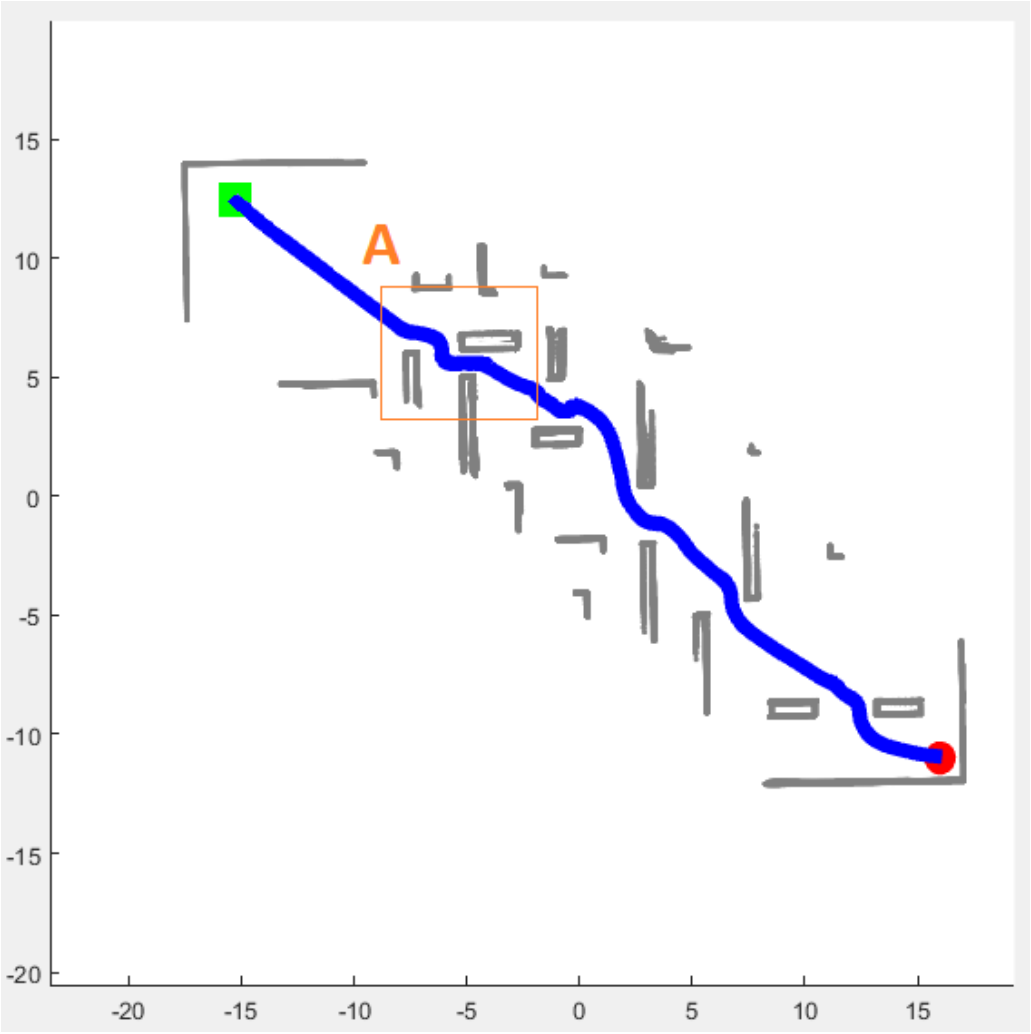


Figura 4.16: Resultado de la simulación realizada en el sexto escenario.

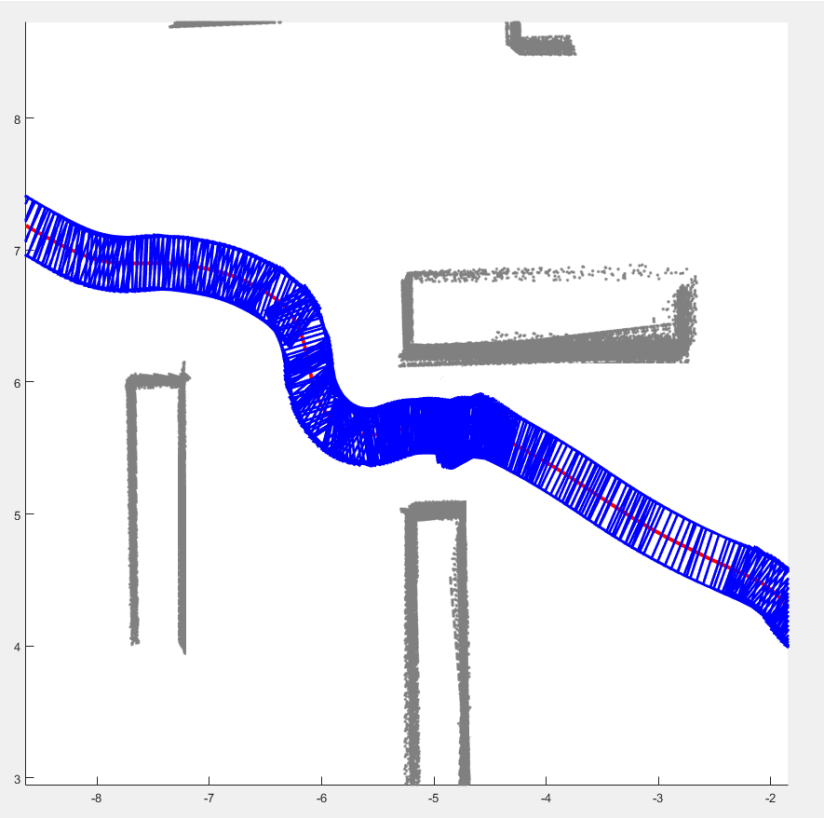


Figura 4.17: Ampliación de la zona A que se destaca en la figura 4.16.

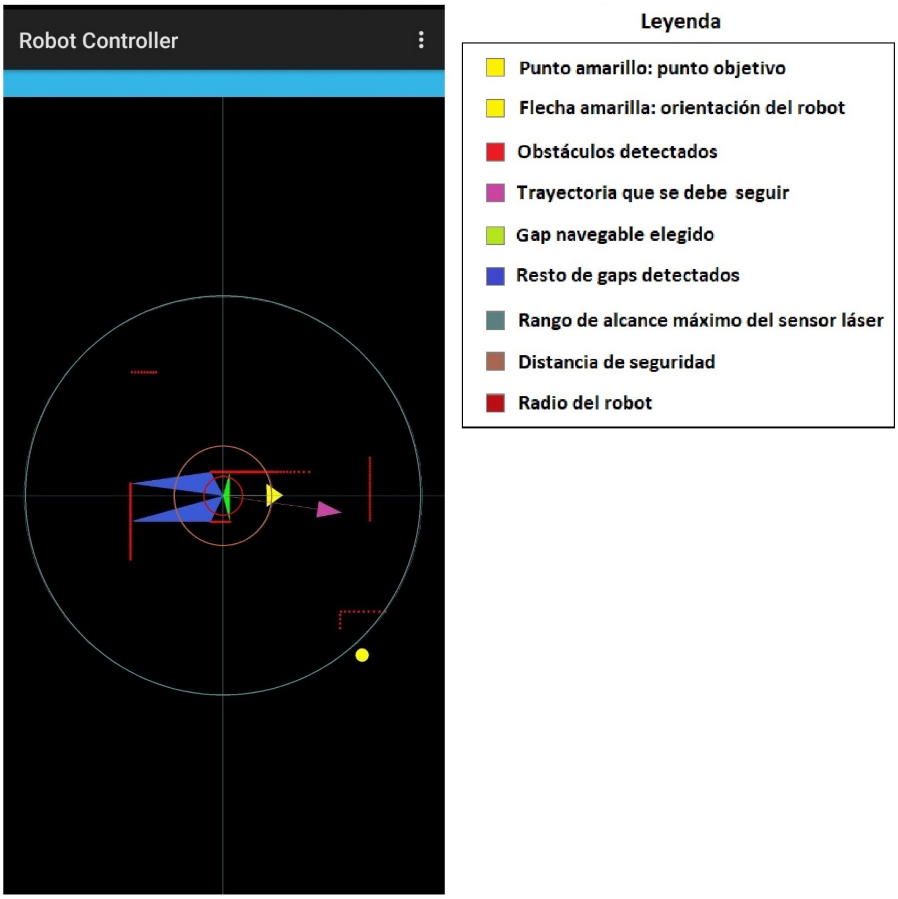


Figura 4.18: Visualización a través de la interfaz gráfica del paso por un *gap* estrecho como parte de la sexta simulación.

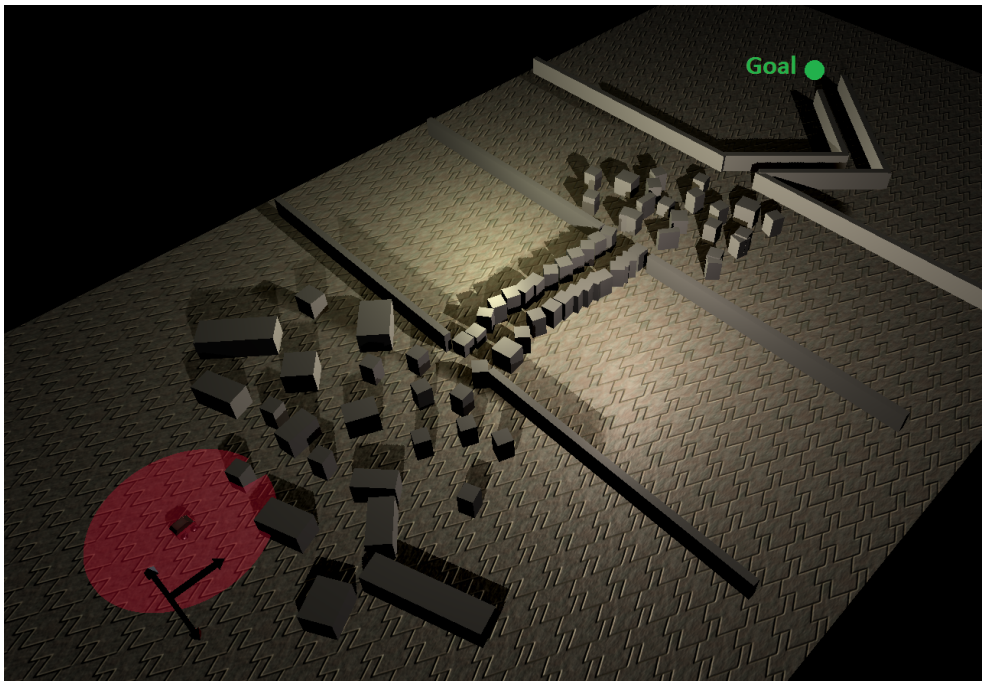


Figura 4.19: Entorno correspondiente al séptimo escenario. El máximo alcance del láser es de 5 metros y el campo de visión es de 360°.

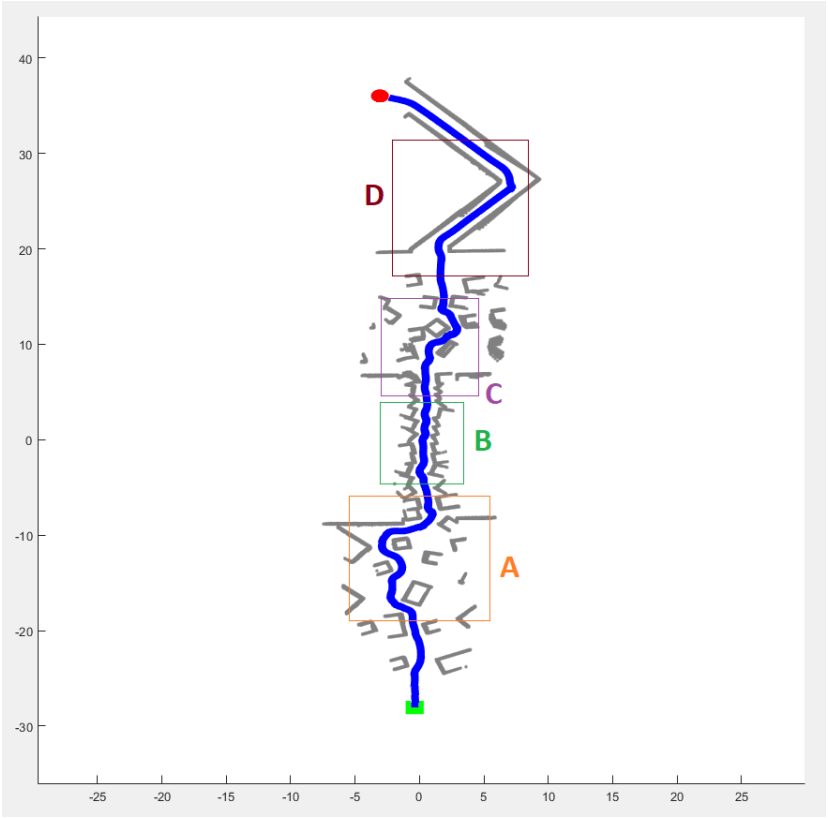
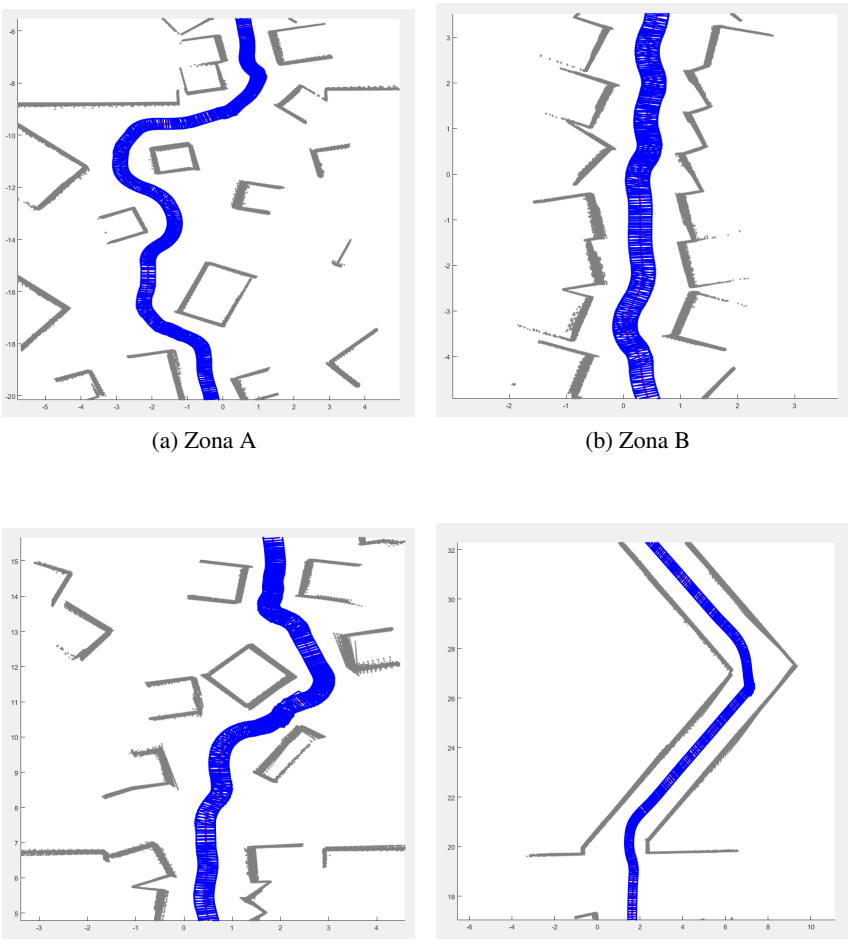


Figura 4.20: Resultado de la simulación realizada en el séptimo escenario.



(a) Zona A

(b) Zona B

(c) Zona C

(d) Zona D

Figura 4.21: Ampliación de las zonas señaladas en la figura 4.20

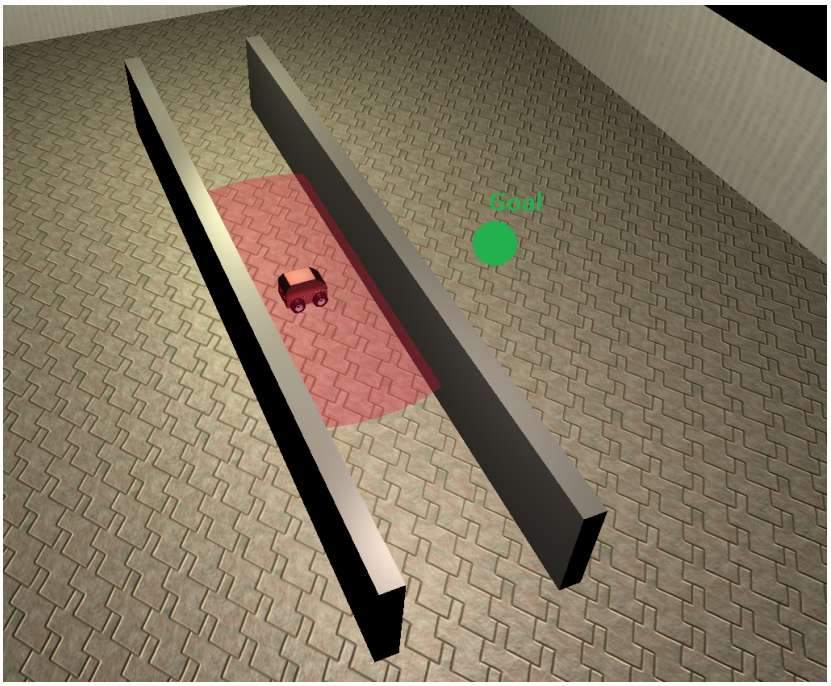


Figura 4.22: Entorno correspondiente al primer escenario desfavorable. El máximo alcance del láser es de 5 metros y el campo de visión es de 360°.

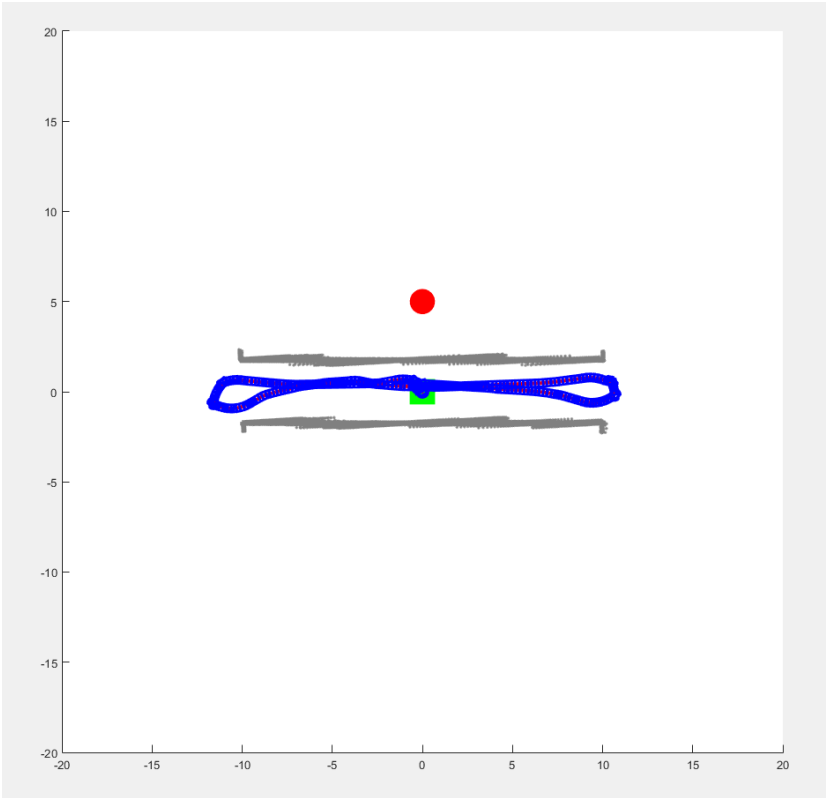


Figura 4.23: Resultado de la simulación realizada en el primer escenario desfavorable.

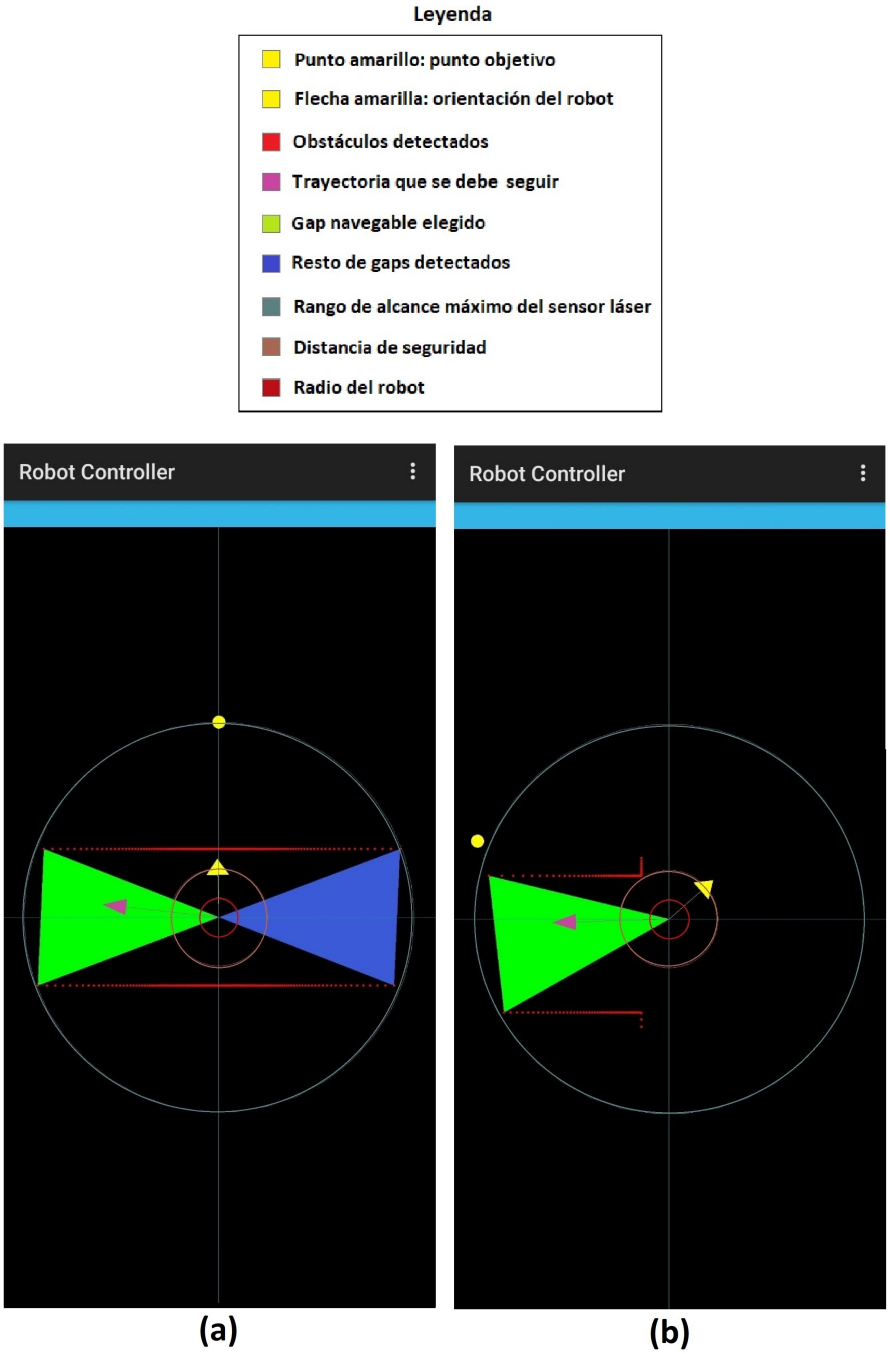


Figura 4.24: Visualización de dos momentos determinados de la simulación en el escenario desfavorable 1. En la figura (a) se puede observar el inicio de la simulación y en la (b) el problema de detección de *gaps* en el extremo del túnel.

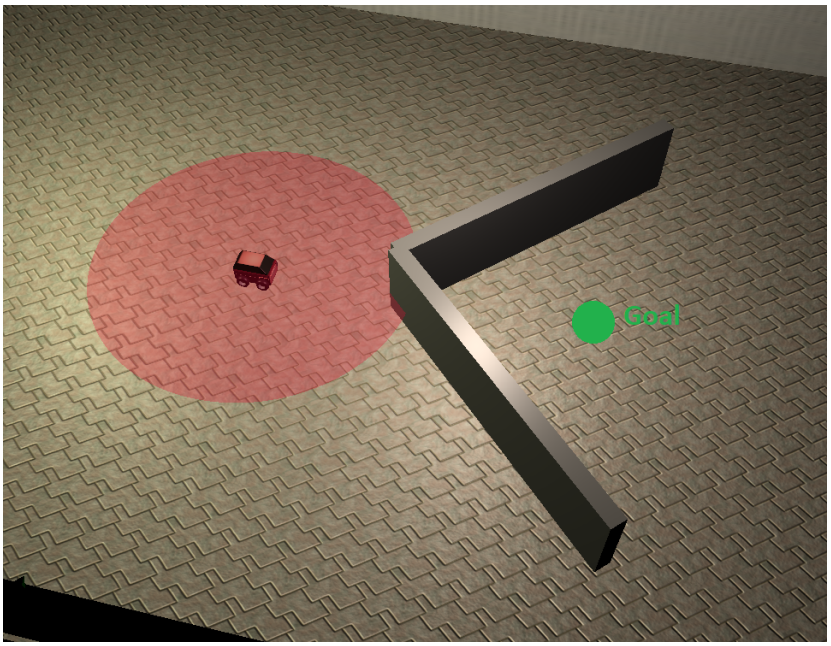


Figura 4.25: Entorno correspondiente al segundo escenario desfavorable. El máximo alcance del láser es de 5 metros y el campo de visión es de 360°.

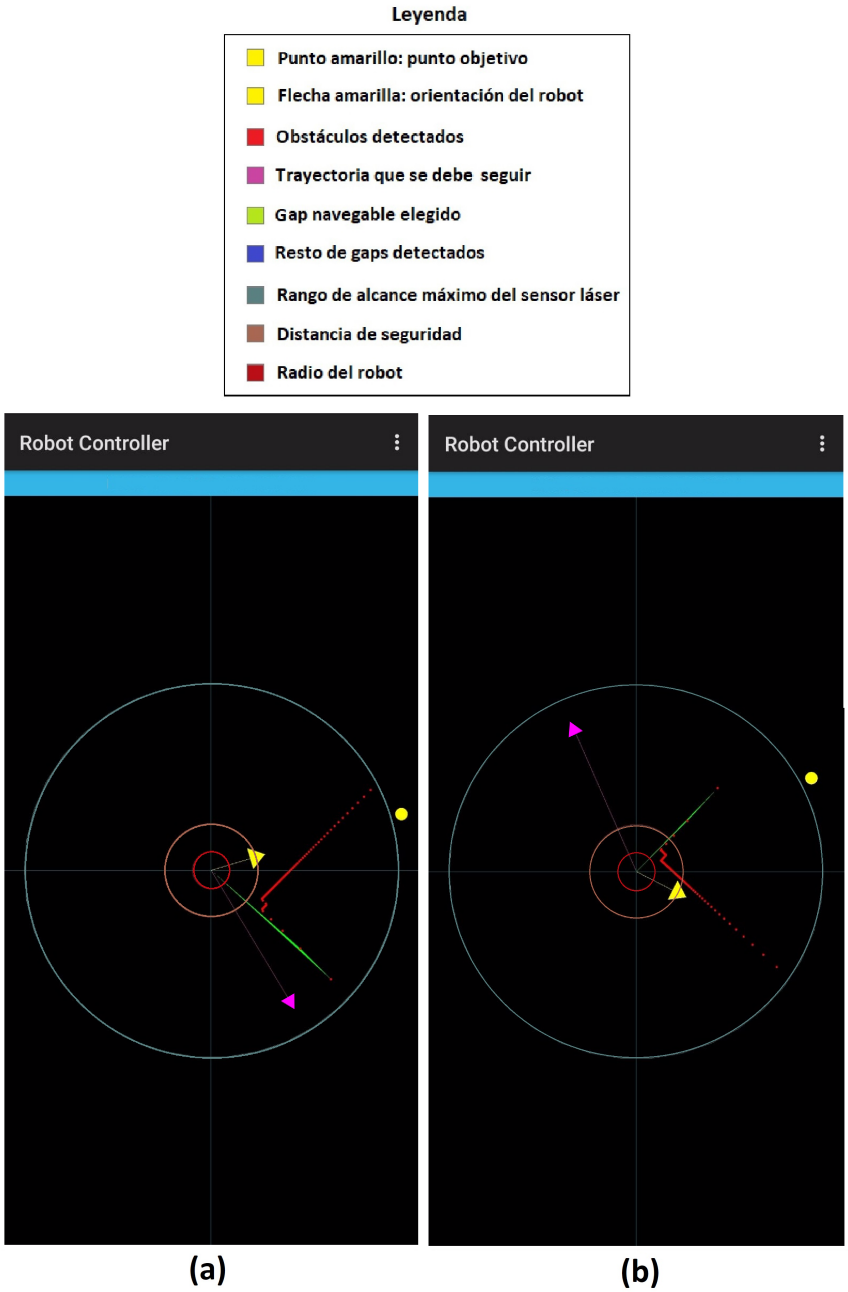


Figura 4.26: Visualización de la detección de dos *gaps* debido a la visión que tiene el robot de los puntos del *scan*. En la figura (a) se observa la detección de un *gap* fugaz en la parte derecha del obstáculo, mientras que en la (b) se observa un *gap* fugaz encontrado en la parte izquierda del obstáculo.

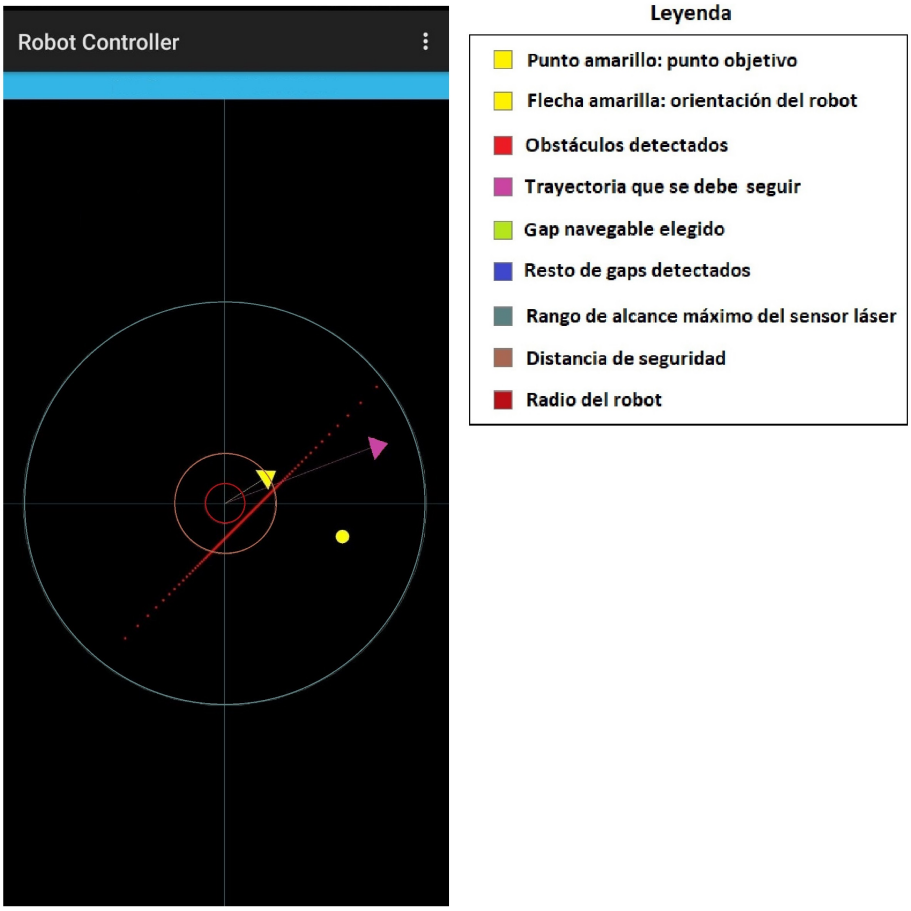


Figura 4.27: Visualización correspondiente al experimento desfavorable 2, de como el robot se queda atrapado al lado de la pared debido a que es incapaz de detectar ningún *gap*.

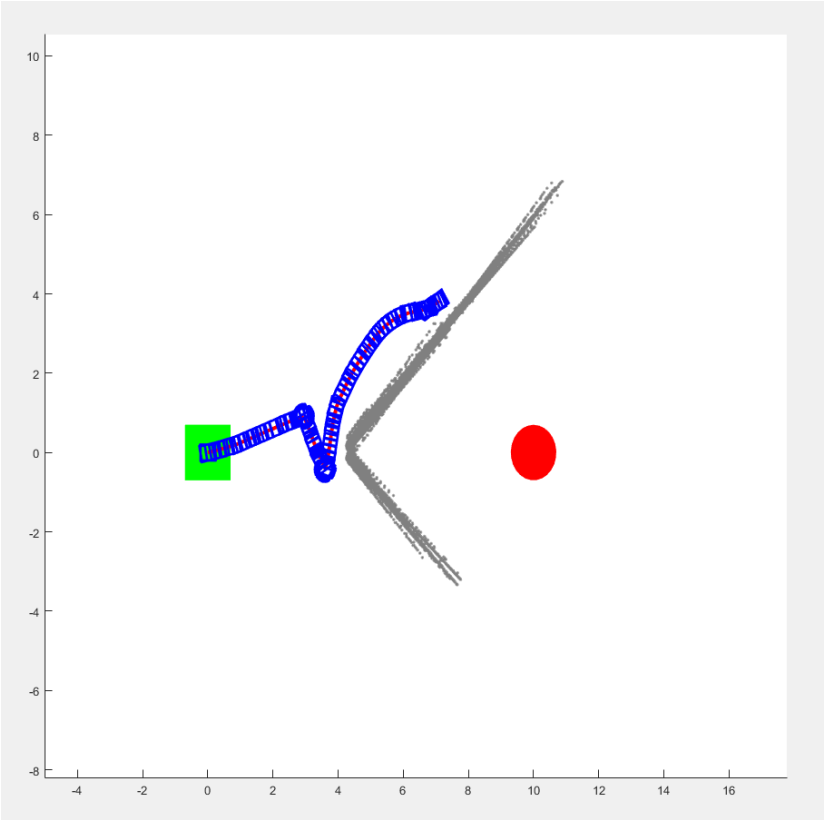


Figura 4.28: Simulación realizada en el segundo escenario desfavorable.

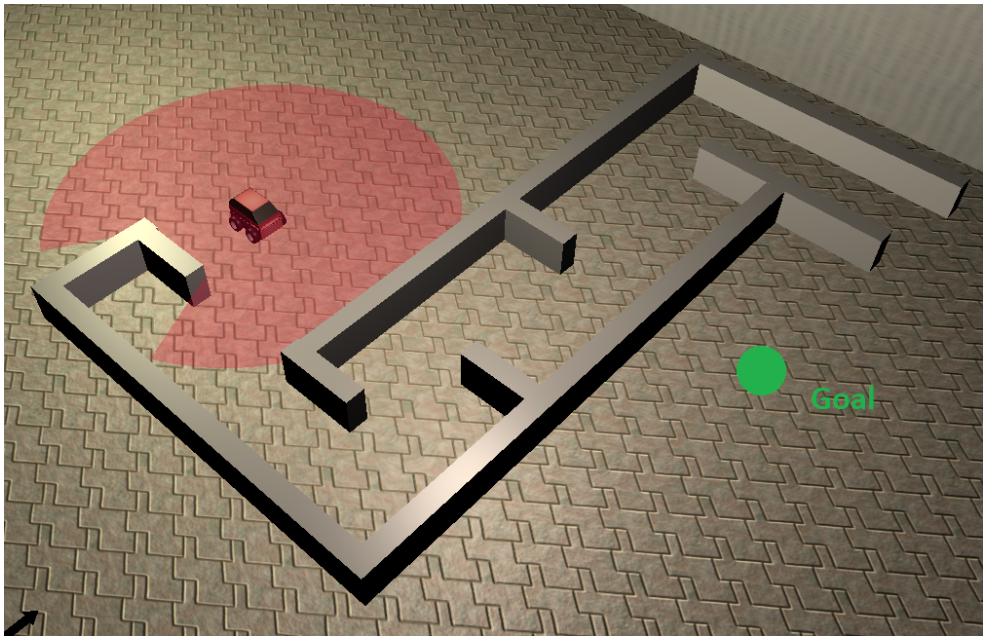


Figura 4.29: Entorno correspondiente al tercer escenario desfavorable. El máximo alcance del láser es de 5 metros y el campo de visión es de 360°.

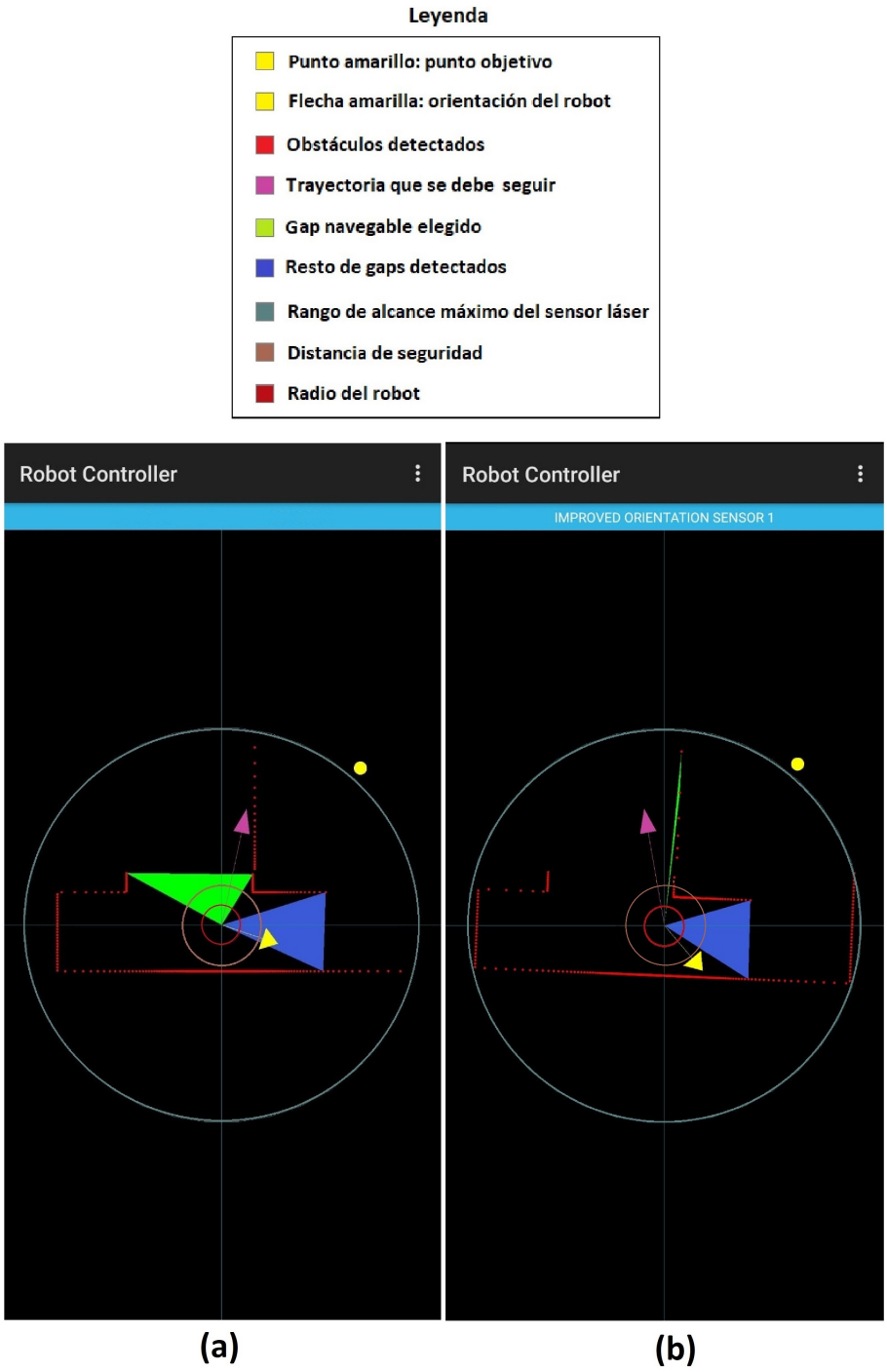


Figura 4.30: Visualización de la detección de dos *gaps* debido a la posición en la que se encuentra el punto objetivo. En la figura (a) se puede ver cómo recién entrado al túnel el robot detecta un *gap* a su izquierda detectando que es el más próximo angularmente al punto objetivo. En la figura (b) se vuelve a detectar un *gap* a la izquierda del robot, en este caso se trata de uno fugaz debido a que los puntos que lo forman son consecutivos.

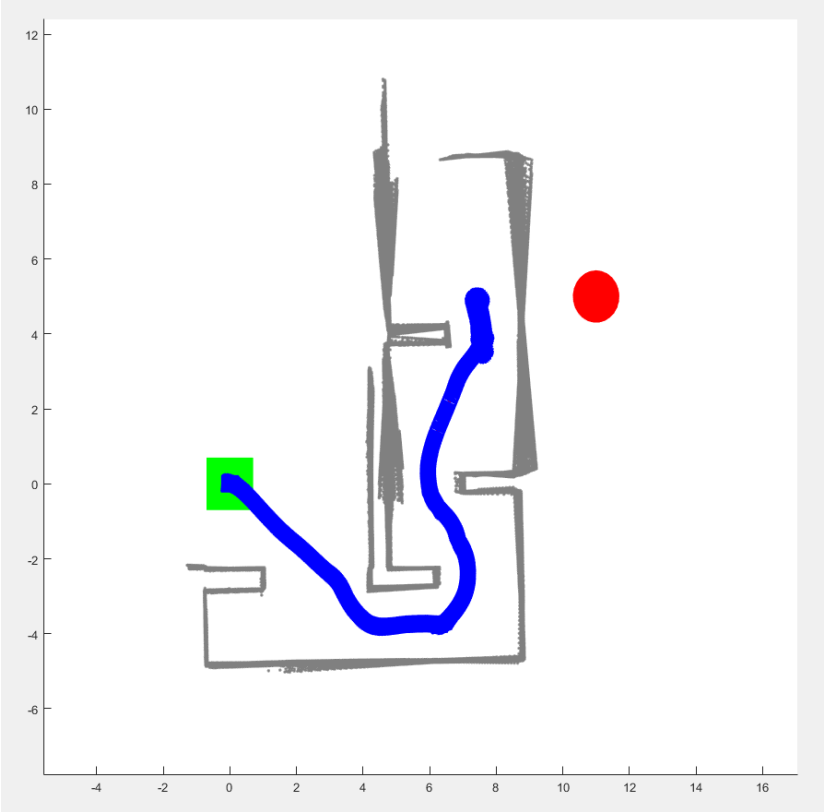


Figura 4.31: Resultado de la simulación realizada en el tercer escenario desfavorable.

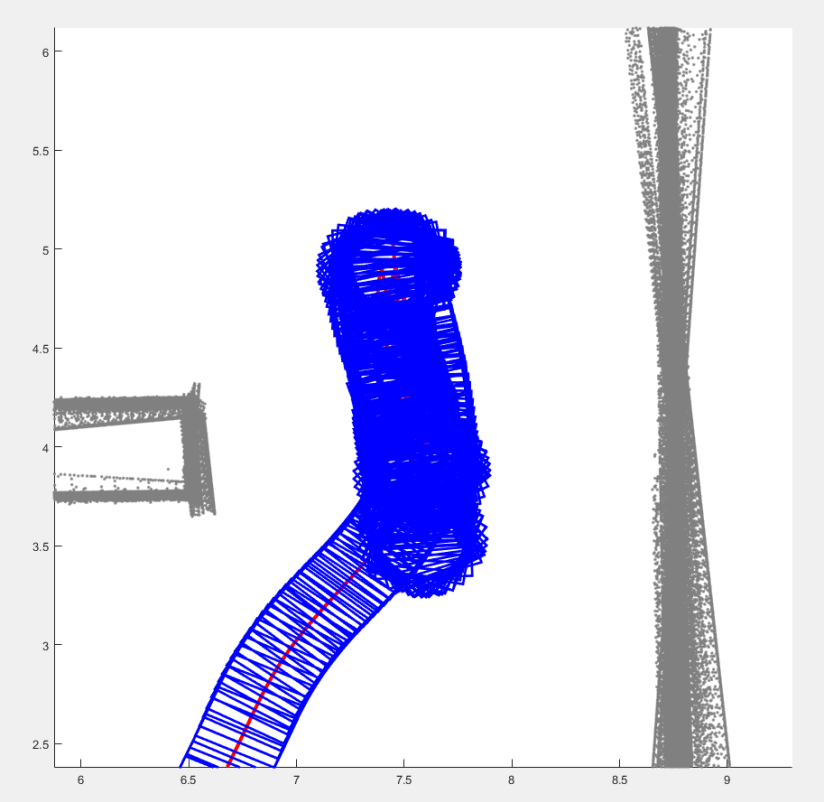


Figura 4.32: Ampliación de la oscilación que sufre el robot en el tercer experimento desfavorable. Se lleva a cabo al llegar al punto que forma la perpendicular con el punto objetivo.

CAPÍTULO 5

CONCLUSIONES

En este capítulo, se analiza el desarrollo del proyecto revisando las fases que lo componen y, finalmente, se exponen futuras ampliaciones que se podrían llevar a cabo.

5.1 Desarrollo del proyecto y conclusiones

La implementación de un algoritmo de evitación de obstáculos llamado *Closest Gap* sobre un robot terrestre ha necesitado las siguientes fases de implementación:

- En una fase previa, se ha acometido el estudio por separado de cada una de las herramientas utilizadas. Familiarizarse con un sistema operativo como Linux Ubuntu y con un entorno de desarrollo como es ROS, ha resultado ser una tarea costosa. La realización de diversos tutoriales y la búsqueda de documentación en la red, ha sido crucial para desarrollar el proyecto. Asimismo, ha sido necesario el estudio del TFG realizado previamente [3] para poder entender su estructura y así utilizarlo como punto de partida.
- La fase de desarrollo constituye la pieza principal del proyecto, donde se encuentra la mayor cantidad de tiempo invertida. El estudio e implementación del algoritmo se encuentran dentro de esta fase. Se necesitó tiempo para comprender la funcionalidad a implementar, así como la forma de estructurarla. La implementación se ha realizado a través de pequeños pasos, comprobando en todo momento que lo que se iba construyendo funcionaba correctamente. Finalmente, una vez se tenía cada parte del algoritmo implementada, se unieron las piezas para así probarlo de forma conjunta.
- En la fase de pruebas, se verificó el correcto funcionamiento del sistema. Se plantearon diversas posibilidades para poder representar de forma gráfica el comportamiento del algoritmo. Al final, la opción de usar MATLAB fue la escogida. Posteriormente, se fueron creando escenarios y se ejecutaron simulaciones para poder comprobar que el algoritmo estaba correctamente implementado.

El desarrollo de un proyecto de este tipo ha necesitado una fase de preparación previa, una fase de desarrollo bien estructurada y las correspondientes pruebas para verificar el trabajo realizado. Cada una de estas fases y en particular la fase de pruebas con las correspondientes simulaciones realizadas, se han llevado a cabo de forma satisfactoria consiguiendo así los resultados esperados.

5.2 Futuras ampliaciones

Durante el transcurso del proyecto se han tenido diversas ideas sobre qué ampliaciones se podrían incorporar para incrementar su funcionalidad. Algunas de las posibles mejoras son las siguientes:

- *Fusionar el algoritmo implementado con otro de tipo deliberativo*. De esta forma, se podría abordar escenarios más complejos, aprovechando la reactividad y comportamiento de CG.
- *Mejora de la aplicación para conseguir una visualización global de lo que el robot ha sensorizado*. En definitiva, se trataría de incorporar a la aplicación Android una función de construcción incremental de un mapa del entorno. Para ello se tendría que tener en cuenta las limitaciones computacionales de los dispositivos móviles.

APÉNDICE A

APÉNDICE A DOCUMENTO *Nearness Diagram* (ND)

El apéndice A del documento [7] se centra en la explicación del algoritmo encargado de verificar la navegabilidad desde un punto A hasta un punto B. Cuando se utilice el algoritmo se podrá saber si es posible alcanzar un punto cualquiera del espacio.

Los *inputs* del algoritmo son los siguientes:

- La localización del robot (X_{robot}) y el radio (R).
- La localización del punto objetivo (X_{goal}).
- Una lista (L) de puntos pertenecientes a obstáculos, dónde un punto considerado como un obstáculos es X_j^L.

El *output* del algoritmo es la verificación de si el punto objetivo puede ser alcanzado por el robot.

El primer paso consiste en dividir el plano en cuatro semiplanos (FL, FR, BL, BR) (*forward left*, *forward right*, *backward left*, *backward right*) por la línea (llamada P) que contiene X_{robot} y X_{goal} y por la línea perpendicular a la previa sobre X_{robot}. En la figura A.1 se pueden ver representados los conceptos definidos hasta el momento.

A continuación, se procederá a ejecutar cada uno de los pasos que componen el algoritmo:

- 1) Si $\exists i \mid d(X_{goal}, X_i^L) < R$ entonces X_{goal} es inalcanzable.
- 2) Eliminar de la lista L cada punto k que cumpla las siguientes condiciones:
  - (a) ($X_k^L \; \epsilon \; BL$) or ($X_k^L \; \epsilon \; BR$)
  - (b) $d(X_k^L, X_{robot}) > d(X_{goal}, X_{robot})$
  - (c) $d(X_k^L, P) > 2R$

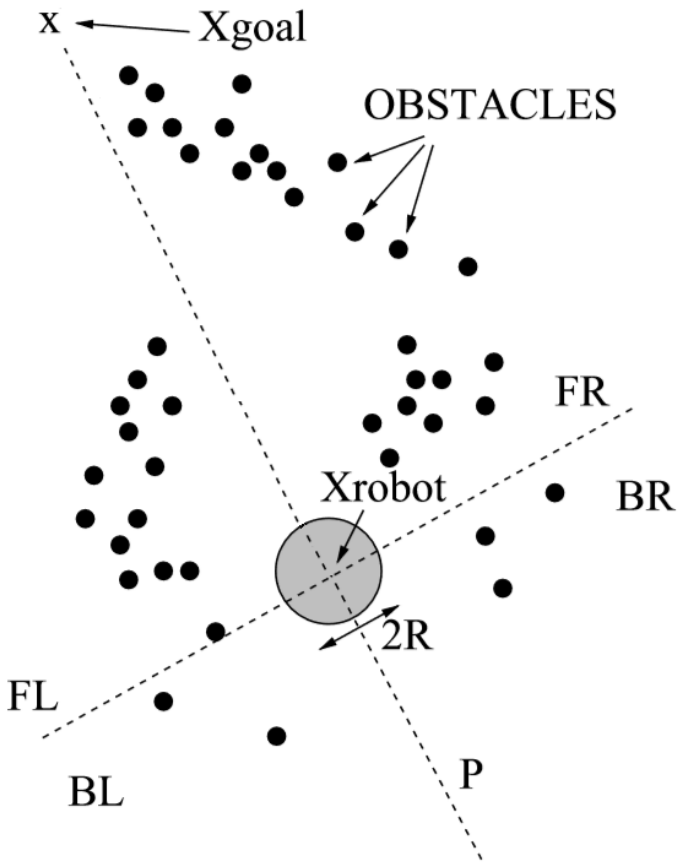


Figura A.1: Representación de las principales variables del algoritmo.

- 3) Para todos los puntos restantes de la lista L, si se cumple que $d(X_j^L, X_k^L) > 2R$ (donde $X_j^L \exists FR$ y $X_k^L \exists FL$) entonces X_{goal} podrá ser alcanzado. En caso contrario, el robot no será capaz de llegar a X_{goal}.

Una vez vistos los pasos por los cuales se compone el algoritmo, pasemos a ver paso por paso cómo actúa realmente:

- 1) Se lleva a cabo un inflado de los obstáculos con el radio del robot tal y como se puede observar en la figura A.2. Posteriormente, se comprueba que el punto objetivo no entre en colisión con ninguna de las esferas creadas.
- 2) Posteriormente se eliminan todos los puntos pertenecientes a obstáculos que se encuentren fuera del rectángulo con anchura $4R$ formado por X_{goal} y X_{robot}. En la figura A.3 se puede observar el rectángulo creado. Hay que tener en cuenta que dentro de este rectángulo es donde realmente se está buscando un camino posible hasta el punto objetivo.
- 3) Finalmente, se comprueba si existen colisiones entre los obstáculos inflados pertenecientes a FR y FL. Si no existe colisión entre ningún obstáculo, entonces se puede asegurar que existe un camino navegable hasta el punto objetivo.

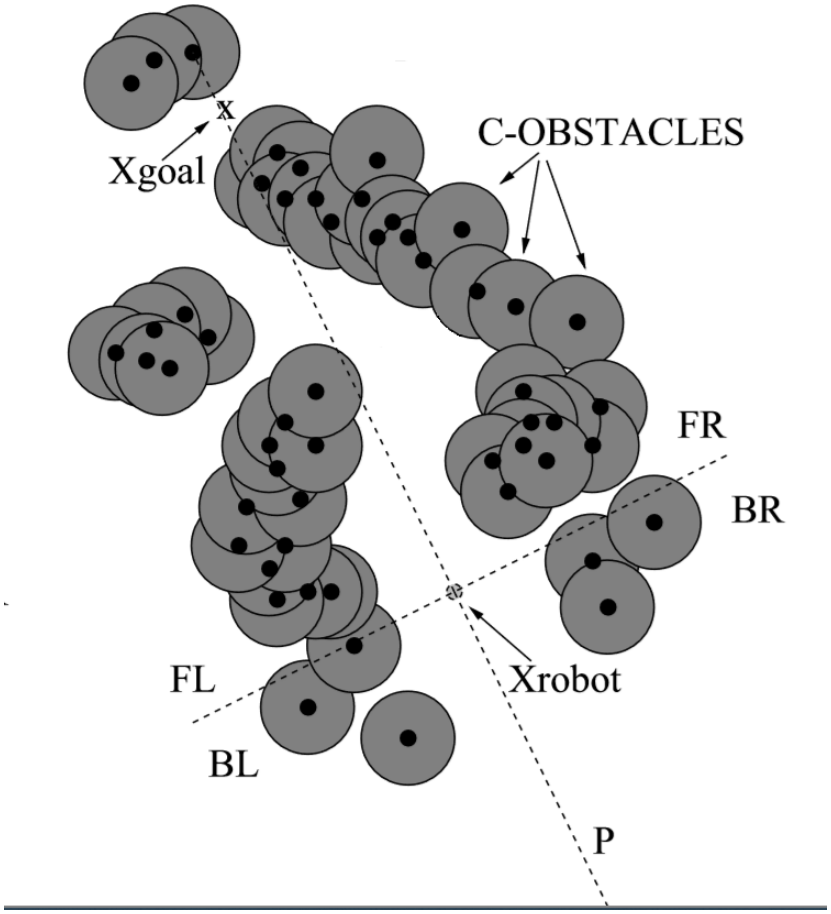


Figura A.2: Representación del inflado de obstáculos.

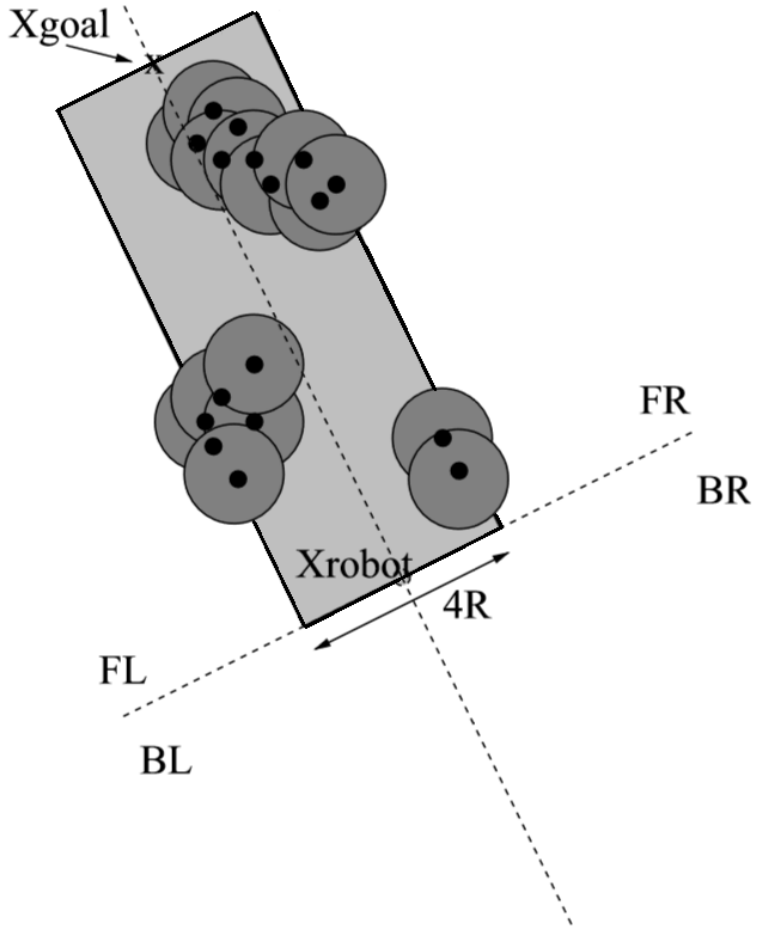


Figura A.3: Rectángulo creado para comprobar la navegabilidad hasta el punto objetivo.

APÉNDICE B

CÓDIGO IMPLEMENTADO EN ROS

B.1 *robot_controller.cpp*

```
/*
* Author: David Arévalo Jiménez
* Year: 2017
* Treball de fi de grau EEIA
* University: Universitat de les Illes Balears (UIB)
*
*/

/*
* Modifier author: Juan Isaac Cifre Izquierdo
* Year: 2018
* Treball de fi de grau EEIA
* University: Universitat de les Illes Balears (UIB)
*
*/

//************************************************//
//************************************************//
//******************** 2017 **********************//
//************************************************//
//************************************************//

#include "ros/ros.h"
#include "geometry_msgs/Twist.h"
#include "nav_msgs/Odometry.h"
#include <sstream>
#include <stdio.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>

using namespace std;

//Global variables
ros::Publisher velocity_publisher; //create a publisher
ros::Subscriber pose_subscriber; //subscribe to Pose
nav_msgs::Odometry car_pose;
geometry_msgs::Twist vel_msg;
geometry_msgs::Twist vel_msg_stop;


double angulos;
double velocidades;

int mode=0,velMax=50,angularMax=50;

// métodos
void rotateREL (double relative_angle, double linear_speed,
   double maxRotVel);
void rotateABS (double angular_speed);
double degrees2radians(double angle_in_degrees);
void poseCallback(const nav_msgs::Odometry::ConstPtr &
   pose_message);
int Server();

//the thread function
void *connection_handler(void *);

//************************************************//
//************************************************//
//************************************************//
//************************************************//
//************************************************//

//************************************************//
//************************************************//
//******************** 2018 **********************//
//************************************************//
//************************************************//

#include "sensor_msgs/LaserScan.h"

#include <string.h>
#include <stdlib.h>
#include <netinet/in.h>


// Se añade la clase algoritmo_cg creada para implementar el
```

```
   algoritmo Closest Gap.
#include </home/isaac/catkin_ws/src/robot_controller/src/
   cg_algorithm.h>


ros::Subscriber laser_subscriber; // Suscripción al láser.

sensor_msgs::LaserScan laser; // Objeto para describir el láser.

// Coordenadas del punto objetivo.
float x_T = 0.0;
float y_T = 0.0;

float R = 0.500; // Radio de robot ATRV.
float Ds; // Distancia de seguridad alrededor del robot.
float k; // Constante k para modificar la influencia de los
   threats.
float v_max; // Velocidad linear máxima a la que puede ir el
   robot.
float w_max; // Velocidad angular máxima a la que puede ir el
   robot.
float Dvs; // Distancia segura de velocidad.


void laser_Callback(const sensor_msgs::LaserScan::ConstPtr &
   laser_msg);

cg_algorithm cg_implementation; // Objeto para poder utilizar la
    clase algoritmo_cg


//***********************************************//
//***********************************************//
//***********************************************//
//***********************************************//
//***********************************************//

int main(int argc, char **argv)
{

  // Initiate new ROS node named "talker"
  ros::init(argc, argv, "robot_controller");
  ros::NodeHandle n;

  //subscribers and publishers
  velocity_publisher = n.advertise<geometry_msgs::Twist>("/atrv/
     motion", 100);

  pose_subscriber = n.subscribe("/atrv/pose", 10, poseCallback);

  // Subscribe to the laser
  laser_subscriber = n.subscribe("/base_scan", 1000,
     laser_Callback);
```

```
  Server();

  return 0;
}

//***********************************************//
//***********************************************//
//******************** 2017 *********************//
//***********************************************//
//***********************************************//

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

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

  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;

  // Posicion

  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;


  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);

}

//**********SERVER*************

int Server(){
```

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

char buf[100];

//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("192.168.1.38");
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);


cg_implementation.initFile();


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
    //pthread_join( sniffer_thread , NULL); // was commented
       before
    puts("Handler assigned");

  }

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

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);

}


//***********************************************//
//***********************************************//
//***********************************************//
//***********************************************//
//***********************************************//


//***********************************************//
//***********************************************//
//******************** 2018 *********************//
//***********************************************//
//***********************************************//

// Función laser_Callback.
// Callback que se encarga de recojer la información que MORSE
   publica
// sobre el láser Hokuyo. Dentro del Callback se guarda la
   información
// necesaria para poder utilizarla en el algoritmo CG.

void laser_Callback(const sensor_msgs::LaserScan::ConstPtr &
   laser_msg){

  // Ángulo mínimo
  laser.angle_min = (laser_msg->angle_min);

  // Ángulo máximo
  laser.angle_max = (laser_msg->angle_max);

  // Rango máximo
  laser.range_max = (laser_msg->range_max);

  // Resolución del láser
  laser.angle_increment = (laser_msg->angle_increment);

  // Capturas (scans) del sensor láser.
  laser.ranges = (laser_msg->ranges);

}
```

```
// Connection_handler creada en 2017 pero modificada en 2018
   para introducir el algoritmo CG

// Función connection_handler.
// Cada vez que se consigue conectar con el cliente se crea un
   nuevo
// thread y se ejecuta esta función.

void *connection_handler(void *socket_desc){

  //Get the socket descriptor

  //*************2017*************/

  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_x=0,e=0;
  float angulo=0,longitud=0;
  memset(client_message, 0, strlen(client_message));

  //*************/

  //*************2018*************/

  // Mensajes que se recibirán a través del socket de la
     aplicación Android.

  //Coordenadas del punto objetivo.
  char x_T_message[10];
  char y_T_message[10];

  char Ds_message[10];
  char k_message[10];
  char Dvs_message[10];
  char v_max_message[10];
  char w_max_message[10];

  ros::Rate loop_rate(50);

  // Velocidad lineal y angular final del robot.
  float vels[2];

  // Variables para poder enviar mensaje a traves de sockets.
  int bufsize = 10000;
  char buffer[bufsize];

  // Posición del robot.
  struct point X_robot;
  X_robot.x = car_pose.pose.pose.position.x;
```

```
X_robot.y = car_pose.pose.pose.position.y;

// Posición del punto objetivo.
struct point X_goal;
X_goal.x = x_T;
X_goal.y = y_T;

// Declarar las tres estructuras para poder realizar el aná
   lisis de gaps.
struct info_scan *scan_forward = (struct info_scan*)malloc(
   sizeof(struct info_scan));
struct info_scan *scan_backward = (struct info_scan*)malloc(
   sizeof(struct info_scan));
struct info_scan *list_of_gaps = (struct info_scan*)malloc(
   sizeof(struct info_scan));

ros::spinOnce();
loop_rate.sleep();

struct config init_config;

init_config.R = R;
init_config.Ds = Ds;
init_config.k = k;
init_config.laser = laser;
init_config.X_goal = X_goal;
init_config.X_robot = X_robot;
init_config.orientation_robot = car_pose.pose.pose.orientation
   .w;
init_config.v_max = v_max;
init_config.w_max = w_max;
init_config.Dvs = Dvs;


// Inicializar algoritmo con todas las variables de la struct
   de tipo config.
cg_implementation.init(init_config);

// Reservar la memoria para las structs.
cg_implementation.reserv_memory(scan_forward, scan_backward,
   list_of_gaps);

// STEP1 DEL ALGORITMO.

cg_implementation.step1(scan_forward, FORWARD);
cg_implementation.step1(scan_backward, BACKWARD);

// STEP2 DEL ALGORITMO.

cg_implementation.step2(list_of_gaps, scan_forward,
   scan_backward);

// Cálculo de la nueva dirección de movimiento.
```

```
cg_implementation.calculate_md(list_of_gaps);

// Método de navegación reactiva.

cg_implementation.navigation_method(vels);

// Crear mensaje que se publicará para el nodo MORSE.

vel_msg = cg_implementation.msg_MORSE(vels);

velocity_publisher.publish(vel_msg);
loop_rate.sleep();

// Crear mensaje que se enviará a través del socket hacia la
   App Android

int longitud_msg = cg_implementation.msg_app(buffer,
   list_of_gaps);

int n_bytes = send(sock, buffer, longitud_msg, 0);


// Liberar la memoria que ocupan las structs.
cg_implementation.liber_memory(scan_forward, scan_backward,
   list_of_gaps);

/************* RECEPCIÓN *************/

while((read_size = recv(sock , client_message , 2000 , 0)) > 0
    )
{

  if(client_message[i_x] == '1'){ // posicion x e y del punto
     objetivo

    i_x=i_x+2;

    //Recibir coordenada X

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

      x_T_message[e] = client_message[i_x];
      e++;
      i_x++;


    }

    x_T = strtof(x_T_message, NULL);

    i_x++;
```

```
e = 0;

//Recibir coordenada Y

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

  y_T_message[e] = client_message[i_x];

  e++;
  i_x++;

}

y_T = strtof(y_T_message, NULL);

i_x++;

e = 0;

// Recibir Ds

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


  Ds_message[e] = client_message[i_x];

  e++;
  i_x++;
}

Ds = strtof(Ds_message, NULL);

i_x++;

e = 0;

// Recibir k

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


  k_message[e] = client_message[i_x];

  e++;
  i_x++;
}

k = strtof(k_message, NULL);

i_x++;

e = 0;
```

```
    // Recibir Dvs

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


      Dvs_message[e] = client_message[i_x];

      e++;
      i_x++;
    }

    Dvs = strtof(Dvs_message, NULL);

    i_x++;

    e = 0;

    // Recibir v_max

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


      v_max_message[e] = client_message[i_x];

      e++;
      i_x++;
    }

    v_max = strtof(v_max_message, NULL);

    i_x++;

    e = 0;

    // Recibir w_max

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


      w_max_message[e] = client_message[i_x];

      e++;
      i_x++;
    }

    w_max = strtof(w_max_message, NULL);



  }else{ // mensaje para que el robot se pueda mover en los
      modos antiguos

    i_x=i_x+2;
```

```
//angulo
while(client_message[i_x]!=' '){
  angulo_message[e]=client_message[i_x];
  e++;
  i_x++;
}

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_x++;
e=0;

while(client_message[i_x]!=' '){
  longitud_message[e]=client_message[i_x];
  i_x++;
  e++;
}

i_x++;

mode_message[0]=client_message[i_x];

e=0;

i_x++;
i_x++;

while(client_message[i_x]!=' '){
  velMax_message[e]=client_message[i_x];
  i_x++;
  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_x++;
      e=0;

      while(client_message[i_x]!=' '){
        angularMax_message[e]=client_message[i_x];
        i_x++;
        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_x=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(connected);
  close(sock);

  pthread_exit(NULL);
  return 0;
}

//*********************************************//
//*********************************************//
//*********************************************//
//*********************************************//
//*********************************************//
```

B.2 *cg_algorithm.h*

```
//  Descripcion: Cabecera (.h) de la clase creada para
   implementar el algoritmo Closest Gap
//  Autor: Juan Isaac Cifre Izquierdo
//  Año : 2018
//  Universitat de les Illes Balears
//  Enginyeria Electrònica Industrial i Automàtica
```

```
#ifndef __CG_ALGORITHM_H_
#define __CG_ALGORITHM_H_

/***************/
/* Librerías   */
/***************/

#include "sensor_msgs/LaserScan.h"
#include "ros/ros.h"
#include "geometry_msgs/Twist.h"
#include "nav_msgs/Odometry.h"
#include <string.h>
#include <iostream>
#include <fstream>

#include <sys/socket.h>
#include <sys/types.h>

#include <stdio.h>      /* printf, scanf, NULL */
#include <stdlib.h>     /* malloc, free, rand */

/*********************************/
/* Variables globales y macros   */
/*********************************/

#define FORWARD 1
#define BACKWARD 0

geometry_msgs::Twist _vel_msg;

const double PI = 3.14159265359;
const double twoPi = 2.0 * 3.14159265359;

/***************/
/* Estructuras  */
/************** */

struct info_scan{

int *points_i;
int *points_j;
float *dist_i;
float *dist_j;
float *dist_gap;
int n_gaps;

};

struct point{

float x;
float y;
```

```
};

struct config{

float R;
float Ds;
float k;
sensor_msgs::LaserScan laser;
struct point X_goal;
struct point X_robot;
float orientation_robot;
float v_max;
float w_max;
float Dvs;

};


class cg_algorithm{

/*******************************/
/* Variables y métodos Private  */
/*******************************/

private:

// Ficheros para poder documentar las pruebas.
std::ofstream myfile1; // Se guarda posición y orientación en
   cada ciclo.
std::ofstream myfile2; // Se guarda todos los puntos detectados
   por el láser en cada ciclo.

bool key_file; // Para saber cuando hay que cerrar el fichero.

float R; // Radio del robot.
float Ds; // Distancia de seguridad.
float Dvs; // Distancia segura de velocidad.
float k; // Constante k
sensor_msgs::LaserScan laser; // láser.
struct point X_goal; // Punto objetivo.
struct point X_robot; // Posición actual del robot.

//Angulo que abarca el sensor.
int field;

//Scans que detecta el láser. Depende del campo de visión y de
   la resolución.
int laser_scans;

float orientation_robot; // Orientación actual del robot.
float v_max; // Velocidad lineal máxima.
float w_max; // Velocidad angular máxima.
```

```
float a_md; // angulo motion direction.

float a_traj; // trayectoria final que seguirá el robot.

int x1; // para saber exactamente que gap ha sido elegido.

int n_obstacles; // Cantidad de obstaculos que detecta el robot.

int n_gaps_aux; // Número de gaps auxiliar para contemplar caso
   especial.

float d_min; // Distancia mínima a la que se encuentra el
   obstaculo más cercano.

// Puntos i y j del gap seleccionado para poder pintarlo bien en
    la App.
float a_gapselected_i;
float a_gapselected_j;

// Ángulos i y j de cada gap.
float *a_gaps_i;
float *a_gaps_j;

double d2r(double angle_in_degrees);
double r2d(double angle_in_radians);

float distBetween2Points(float x1, float y1, float x2, float y2)
   ;
float distBetween2Points(struct point P1, struct point P2);

float minDistBetween2Angles(float angle1, float angle2);
float minDistBetween2AnglesWithSign(float angle1, float angle2);

bool angle_inside_2angles(float angleX, float angleY, float
   angleZ);

void points_xy_gaps(float *xp_gaps_i, float *yp_gaps_i, float *
   xp_gaps_j, float *yp_gaps_j, struct info_scan *list_of_gaps);

bool appendixA_ND(struct point _X_robot, struct point _X_goal,
   struct point *_list_obstacles, int n_obstacles);
bool appendixA_ND_D(struct point _X_robot, struct point _X_goal,
    struct point *_list_obstacles, int n_obstacles);

float f_sat(float a, float b, float x);


float convert02PI (float angle);

float f_proj(float angle);

std::string ConvertFtS (float number);
std::string ConvertItS (int number);
```

```
std::string calculate_xy(struct info_scan *info_scan);

/*******************/
/* Métodos public  */
/*******************/

public:

void init(struct config init_config);
void initFile();

void reserv_memory(struct info_scan *scan_forward, struct
   info_scan *scan_backward, struct info_scan *list_of_gaps);
void liber_memory(struct info_scan *scan_forward, struct
   info_scan *scan_backward, struct info_scan *list_of_gaps);

void step1(struct info_scan *_scan, bool type_scan);
void step2(struct info_scan *list_of_gaps, struct info_scan *
   scan_forward, struct info_scan *scan_backward);

void calculate_md(struct info_scan *list_of_gaps);
void navigation_method(float *velocidades);

geometry_msgs::Twist msg_MORSE(float *velocidades);
int msg_app(char *buffer, struct info_scan *list_of_gaps);

};

#endif
```

B.3 *cg_algorithm.cpp*

```
//  Descripcion: Cabecera (.h) de la clase creada para
   implementar el algoritmo Closest Gap
//  Autor: Juan Isaac Cifre Izquierdo
//  Año : 2018
//  Universitat de les Illes Balears
//  Enginyeria Electrònica Industrial i Automàtica

#ifndef __CG_ALGORITHM_H_
#define __CG_ALGORITHM_H_

/***************/
/* Librerías   */
/***************/

#include <stdio.h>
#include <cmath>
#include <iostream>
```

```
#include <string.h>


#include </home/isaac/catkin_ws/src/robot_controller/src/
    cg_algorithm.h>

using namespace std;
```

```
// Método d2r.
// Permite pasar de grados sexagesimales a radianes.

double cg_algorithm::d2r(double angle_in_degrees){
  return angle_in_degrees *PI /180.0;
}
```

```
// Método r2d.
// Permite pasar de radianes a grados sexagesimales.

double cg_algorithm::r2d(double angle_in_radians){
  return angle_in_radians *180.0 /PI;
}
```

```
// Método distBetween2Points (coordenadas).
// Permite conocer la distancia entre dos puntos indicados
    mediante
// suscoordenadas.

float cg_algorithm::distBetween2Points(float x1, float y1, float
     x2, float y2){

  return sqrt(((x2-x1)*(x2-x1))+((y2-y1)*(y2-y1)));

}
```

```
// Método distBetween2Points (struct)).
// Permite conocer la distancia entre dos puntos indicados
    mediante
// una struct point.

float cg_algorithm::distBetween2Points(struct point P1, struct
    point P2){

  return sqrt(((P2.x-P1.x)*(P2.x-P1.x))+((P2.y-P1.y)*(P2.y-P1.y)
      ));
```

```
}
```

```
// Método minDistBetween2Angles.
// Permite calcular de forma absoluta la mínima distancia
// entre dos ángulos cualesquiera.

float cg_algorithm::minDistBetween2Angles(float angle1, float
    angle2){ // distc y distcc del algoritmo CG

  float y = angle1-angle2;

  float x = abs(y);

  // Se hace el modulo de la abs(resta de los dos angulos) con 2
     pi

  if (x>PI){

    x = twoPi - x;
  }

  return x;

}
```

```
// Método minDistBetween2AnglesWithSign.
// Permite calcular la mínima distancia entre dos ángulos
    cualesquiera
// con signo. Angle 1 es el angulo de partida y Angle 2 el
    angulo de llegada.
// Se calcula el angulo mínimo entre ellos dos partiendo de
    Angle1. Por lo
// tanto, se tiene en cuenta el sentido de giro.

float cg_algorithm::minDistBetween2AnglesWithSign( float angle1,
     float angle2 ){

  float w = minDistBetween2Angles(angle1, angle2);

  float angle1_aux = convert02PI(angle1);
  float angle2_aux = convert02PI(angle2);

  float the_rest_of_the_360 = (2*PI) - w;

  float angle2_aux_sup = angle2_aux + 0.05;
  float angle2_aux_inf = angle2_aux - 0.05;
```

```
  float s; // signo

  if((((angle1_aux + w) >= angle2_aux_inf) && ((angle1_aux + w)
     <= angle2_aux_sup)) || (((angle1_aux - the_rest_of_the_360)
      >= angle2_aux_inf) && ((angle1_aux - the_rest_of_the_360)
     <= angle2_aux_sup))){

    return w;

  }else if((((angle1_aux - w) >= angle2_aux_inf) && ((angle1_aux
      - w) <= angle2_aux_sup)) || (((angle1_aux +
     the_rest_of_the_360) >= angle2_aux_inf) && ((angle1_aux +
     the_rest_of_the_360) <= angle2_aux_sup))){

    return -w;

  }

}
```

```
// Método angle_inside_2angles.
// Permite conocer si un ángulo cualquiera (angle X) se
   encuentra entre
// otros dos ángulos cualesquiera (angleY y angleZ).

bool cg_algorithm::angle_inside_2angles(float angleX, float
   angleY, float angleZ){

  float angleX_aux = convert02PI(angleX);
  float angleY_aux = convert02PI(angleY);
  float angleZ_aux = convert02PI(angleZ);

  float diferencia;

  float angle1;
  float angle2;
  bool angleX_inside = false;

  if(angleY_aux > angleZ_aux){

    if((angleY_aux-angleZ_aux) > PI){

      angle1 = angleY_aux;
      angle2 = angleZ_aux;

    }else{

      angle1 = angleZ_aux;
      angle2 = angleY_aux;
```

```
  }

}else{

  if((angleZ_aux-angleY_aux) > PI){

    angle1 = angleZ_aux;
    angle2 = angleY_aux;

  }else{

    angle1 = angleY_aux;
    angle2 = angleZ_aux;

  }


}

diferencia = minDistBetween2Angles(angleY, angleZ);

if(angle1 + diferencia > twoPi){

  // El 0 está entre ellos dos

  if((angleX_aux >= 0 && angleX_aux <= angle2) || (angleX_aux
     >= angle1 && angleX_aux <= twoPi)){
    // Angulo X está entre a1 y a2
    angleX_inside = true;
  }else{
    // Angulo X NO está entre a1 y a2

  }


}else{

  // El 0 NO está entre ellos dos

  if((angleX_aux >= angle1 && angleX_aux <= angle2)){
    // Angulo X está entre a1 y a2
    angleX_inside = true;
  }else{
    // Angulo X NO está entre a1 y a2Â¡

  }


}


return angleX_inside;
```

```
}
```

```
// Método points_xy_gaps.
// Permite las coordenadas de cada extremo del gap.

void cg_algorithm::points_xy_gaps(float *_xp_gaps_i, float *
    _yp_gaps_i, float *_xp_gaps_j, float *_yp_gaps_j, struct
    info_scan *list_of_gaps){

  for(int i = 0; i < list_of_gaps->n_gaps; i++){

    _xp_gaps_i[i] = list_of_gaps->dist_i[i]*cos((list_of_gaps->
        points_i[i]*laser.angle_increment)+orientation_robot-d2r(
        field/2));
    _yp_gaps_i[i] = list_of_gaps->dist_i[i]*sin((list_of_gaps->
        points_i[i]*laser.angle_increment)+orientation_robot-d2r(
        field/2));

    _xp_gaps_j[i] = list_of_gaps->dist_j[i]*cos((list_of_gaps->
        points_j[i]*laser.angle_increment)+orientation_robot-d2r(
        field/2));
    _yp_gaps_j[i] = list_of_gaps->dist_j[i]*sin((list_of_gaps->
        points_j[i]*laser.angle_increment)+orientation_robot-d2r(
        field/2));

  }
}
```

```
// Método appendixA_ND.
// Aplica el apéndice A del algoritmo Nearness Diagram. Permite
    conocer
// si existe un camino navegable hasta un punto objetivo.

bool cg_algorithm::appendixA_ND(struct point _X_robot, struct
    point _X_goal, struct point *_list_obstacles, int n_obstacles
    ){

  // Paso 0. Dividir el plano en cuatro semiplanos: FL, FR, BL,
      BR (F: FORWARD, B: BACKWARD, R: RIGHT, L: LEFT)
  // Existe una linia llamada P que contiene X_robot y X_goal

  // Lo que se hará será calcular el punto de cada extremo de
      las linias que forman el semiplano.

  struct point _goal_rR; // Coordenadas punto goal respecto al
      robot

  _goal_rR.x = _X_goal.x - _X_robot.x;
```

```
_goal_rR.y = _X_goal.y - _X_robot.y;

struct point origin; // Punto origen. Nos hará falta más
   adelante.
origin.x = 0;
origin.y = 0;

float _a_goal = atan2(_goal_rR.y, _goal_rR.x); // Angulo de
   goal respecto al robot

float W = distBetween2Points(_X_robot, _X_goal); // Distancia
   entre el robot y el punto objetivo

struct point A; // Puntos A, B, C y D que delimitaran las 4
   zonas creadas
struct point B;
struct point C;
struct point D;

A.x = W*cos(_a_goal); // Coordenadas de los puntos A, B, C y D
A.y = W*sin(_a_goal);

B.x = W*cos(_a_goal - PI/2);
B.y = W*sin(_a_goal - PI/2);

C.x = W*cos(_a_goal - PI);
C.y = W*sin(_a_goal - PI);

D.x = W*cos(_a_goal - (3*PI)/2);
D.y = W*sin(_a_goal - (3*PI)/2);

float a_A = atan2(A.y, A.x);
float a_B = atan2(B.y, B.x);
float a_C = atan2(C.y, C.x);
float a_D = atan2(D.y, D.x);

// A partir de ahora tendemos 4 arrays guardando en cada uno
   los puntos que se encuentren respectivamente en FL, FR, BL
   y BR
// Entre A-B -> FR
// Entre B-C -> BR
// Entre C-D -> BL
// Entre D-A -> FL

int c_FR = 0;
int c_FL = 0;
int c_BR = 0;
int c_BL = 0;

struct point points_FR[n_obstacles + 1];
struct point points_FL[n_obstacles + 1];
struct point points_BR[n_obstacles + 1];
struct point points_BL[n_obstacles + 1];
```

```
float angles_FR[n_obstacles + 1];
float angles_FL[n_obstacles + 1];
float angles_BR[n_obstacles + 1];
float angles_BL[n_obstacles + 1];

// Paso 0. Calcular angulo de cada punto y mirar en que zona
   FR,FL,BR,BL se encuentra

float a_points[n_obstacles + 1];

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

  a_points[i] = atan2(_list_obstacles[i].y, _list_obstacles[i
     ].x);

  if(angle_inside_2angles(a_points[i], a_A, a_B) == true){ //
     Comprobar si está entre A y B -> FR

    points_FR[c_FR] = _list_obstacles[i];
    angles_FR[c_FR] = a_points[i];
    c_FR++;

  }else if(angle_inside_2angles(a_points[i], a_B, a_C) == true
     ){ // Comprobar si está entre B y C -> BR

    points_BR[c_BR] = _list_obstacles[i];
    angles_BR[c_BR] = a_points[i];
    c_BR++;

  }else if(angle_inside_2angles(a_points[i], a_C, a_D) == true
     ){ // Comprobar si está entre C y D -> BL

    points_BL[c_BL] = _list_obstacles[i];
    angles_BL[c_BL] = a_points[i];
    c_BL++;

  }else{ // Si no se cumple ninguna anterior significa que est
     á entre D y A -> FL

    points_FL[c_FL] = _list_obstacles[i];
    angles_FL[c_FL] = a_points[i];
    c_FL++;

  }

}

// Paso 1. Si existe algun punto i de la lista de obstaculos
   el cual d(Xgoal,Li) < R entonces Xgoal no se puede alcanzar

bool key_return = true; // true -> se cumple algoritmo; false
   -> no se cumple algoritmo
```

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

  if(distBetween2Points(_list_obstacles[i], _goal_rR) < R){
    key_return = false;
    break;
  }

}


// Paso 2. Eliminar de L cada punto k que:

//       a) Forme parte de BL o BR

//       b) d(Xk, Xrobot) > d(Xgoal, Xrobot)

//       c) d(Xk, P) > 2R


// 2a) Simplemente basta con ignorar BL y BR

// 2b) Hacemos las comprobaciones con FR y FL y vamos
   modificandolos
// y haciendo la copia en otro array

if(key_return == true){

  int c_FR_aux = 0; // Nuevos contadores auxiliares para las
     zonas FR y FL
  int c_FL_aux = 0;

  struct point points_FR_aux[c_FR + 1]; // Nuevas estructuras
     de puntos auxiliares para las zonas FR y FL
  struct point points_FL_aux[c_FL + 1];

  float angles_FR_aux[c_FR + 1];
  float angles_FL_aux[c_FL + 1];

  // Primero para FR

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

    if(distBetween2Points(points_FR[i], origin) <
       distBetween2Points(_X_robot, _X_goal)){
      points_FR_aux[c_FR_aux] = points_FR[i];
      angles_FR_aux[c_FR_aux] = angles_FR[i];
      c_FR_aux++;
    }

  }

  // Segundo para FL
```

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

  if(distBetween2Points(points_FL[i], origin) <
     distBetween2Points(_X_robot, _X_goal)){
    points_FL_aux[c_FL_aux] = points_FL[i];
    angles_FL_aux[c_FL_aux] = angles_FL[i];
    c_FL_aux++;
  }

}

// 2c) Eliminar aquellos puntos cual distancia d(x, P) > R

float pendiente = (_X_goal.y - _X_robot.y)/(_X_goal.x -
   _X_robot.x);
float b = 0;
float perpendicularpendiente = -(1/pendiente);
float perpendicularB;
float x_interseccion, y_interseccion;
float dist_x, dist_y, dist;

int c_FR_aux2 = 0;
int c_FL_aux2 = 0;

struct point points_FR_aux2[c_FR_aux + 1];
struct point points_FL_aux2[c_FL_aux + 1];

float angles_FR_aux2[c_FR_aux + 1];
float angles_FL_aux2[c_FL_aux + 1];

// Primero para FR.

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

  perpendicularB = points_FR_aux[i].y -
     perpendicularpendiente*points_FR_aux[i].x;

  x_interseccion = (perpendicularB-b)/(pendiente-
     perpendicularpendiente);

  y_interseccion = x_interseccion*pendiente + b;

  dist_x = x_interseccion - points_FR_aux[i].x;

  dist_y = y_interseccion - points_FR_aux[i].y;

  dist = sqrt(dist_x*dist_x + dist_y*dist_y);

  if(dist <= R){

    points_FR_aux2[c_FR_aux2] = points_FR_aux[i];
    angles_FR_aux2[c_FR_aux2] = angles_FR_aux[i];
```

```
      c_FR_aux2++;

    }

  }

  // Segundo para FL.

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

    perpendicularB = points_FL_aux[i].y -
       perpendicularpendiente*points_FL_aux[i].x;

    x_interseccion = (perpendicularB-b)/(pendiente-
       perpendicularpendiente);

    y_interseccion = x_interseccion*pendiente + b;

    dist_x = x_interseccion - points_FL_aux[i].x;

    dist_y = y_interseccion - points_FL_aux[i].y;

    dist = sqrt(dist_x*dist_x + dist_y*dist_y);

    if(dist <= R){

      points_FL_aux2[c_FL_aux2] = points_FL_aux[i];
      angles_FL_aux2[c_FL_aux2] = angles_FL_aux[i];

      c_FL_aux2++;

    }

  }

  // Paso 3. Si para todos los puntos que hayan quedado de FR
     y FL,
  // se cumple que la distancia entre ellos sea más grando que
      2R,
  // entonces Xgoal puede ser alcanzado. Si no se cumple para
     todos,
  // entonces Xgoal no puede ser alcanzado.
  // Los puntos a comparar se encuentran en points_FR_aux2 y
     points_FL_aux2

  for(int i = 0; i < c_FR_aux2; i++){

    for(int j = 0; j < c_FL_aux2; j++){

      if(distBetween2Points(points_FR_aux2[i], points_FL_aux2[
         j]) < R){
        key_return = false;
```

```
        }


      }

    }

  }

  return key_return;
}
```

```
// Método appendixA_ND_D.
// Aplica el apéndice A del algoritmo Nearness Diagram. En este
   caso, permite
// conocer si existe un camino DIRECTAMENTE navegable hasta un
   punto objetivo.

bool cg_algorithm::appendixA_ND_D(struct point _X_robot, struct
   point _X_goal, struct point *_list_obstacles, int n_obstacles
   ){

  // Paso 0. Dividir el plano en cuatro semiplanos: FL, FR, BL,
     BR (F: FORWARD, B: BACKWARD, R: RIGHT, L: LEFT)
  // Existe una linia llamada P que contiene X_robot y X_goal

  // Lo que se hará será calcular el punto de cada extremo de
     las linias que forman el semiplano.


  struct point _goal_rR; // Coordenadas punto goal respecto al
     robot

  _goal_rR.x = _X_goal.x - _X_robot.x;
  _goal_rR.y = _X_goal.y - _X_robot.y;

  struct point origin; // Punto origen. Nos hará falta más
     adelante.
  origin.x = 0;
  origin.y = 0;

  float _a_goal = atan2(_goal_rR.y, _goal_rR.x); // Angulo de
     goal respecto al robot

  float W = distBetween2Points(_X_robot, _X_goal); // Distancia
     entre el robot y el punto objetivo

  struct point A; // Puntos A, B, C y D que delimitaran las 4
```

```
    zonas creadas
struct point B;
struct point C;
struct point D;

A.x = W*cos(_a_goal); // Coordenadas de los puntos A, B, C y D
A.y = W*sin(_a_goal);

B.x = W*cos(_a_goal - PI/2);
B.y = W*sin(_a_goal - PI/2);

C.x = W*cos(_a_goal - PI);
C.y = W*sin(_a_goal - PI);

D.x = W*cos(_a_goal - (3*PI)/2);
D.y = W*sin(_a_goal - (3*PI)/2);

float a_A = atan2(A.y, A.x);
float a_B = atan2(B.y, B.x);
float a_C = atan2(C.y, C.x);
float a_D = atan2(D.y, D.x);

// A partir de ahora tendemos 4 arrays guardando en cada uno
    los puntos que se encuentren respectivamente en FL, FR, BL
    y BR
// Entre A-B -> FR
// Entre B-C -> BR
// Entre C-D -> BL
// Entre D-A -> FL

int c_FR = 0;
int c_FL = 0;
int c_BR = 0;
int c_BL = 0;

struct point points_FR[n_obstacles + 1];
struct point points_FL[n_obstacles + 1];
struct point points_BR[n_obstacles + 1];
struct point points_BL[n_obstacles + 1];

float angles_FR[n_obstacles + 1];
float angles_FL[n_obstacles + 1];
float angles_BR[n_obstacles + 1];
float angles_BL[n_obstacles + 1];

// Paso 0. Calcular angulo de cada punto y mirar en que zona
    FR,FL,BR,BL se encuentra

float a_points[n_obstacles +1];

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

  a_points[i] = atan2(_list_obstacles[i].y, _list_obstacles[i
```

```
      ].x);

  if(angle_inside_2angles(a_points[i], a_A, a_B) == true){ //
     Comprobar si está entre A y B -> FR

    points_FR[c_FR] = _list_obstacles[i];
    angles_FR[c_FR] = a_points[i];
    c_FR++;

  }else if(angle_inside_2angles(a_points[i], a_B, a_C) == true
     ){ // Comprobar si está entre B y C -> BR

    points_BR[c_BR] = _list_obstacles[i];
    angles_BR[c_BR] = a_points[i];
    c_BR++;

  }else if(angle_inside_2angles(a_points[i], a_C, a_D) == true
     ){ // Comprobar si está entre C y D -> BL

    points_BL[c_BL] = _list_obstacles[i];
    angles_BL[c_BL] = a_points[i];
    c_BL++;

  }else{ // Si no se cumple ninguna anterior significa que est
     á entre D y A -> FL

    points_FL[c_FL] = _list_obstacles[i];
    angles_FL[c_FL] = a_points[i];
    c_FL++;

  }

}

// Paso 1. Si existe algun punto i de la lista de obstaculos
   el cual d(Xgoal,Li) < R entonces Xgoal no se puede alcanzar

bool key_return = true; // true -> se cumple algoritmo; false
   -> no se cumple algoritmo

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

  if(distBetween2Points(_list_obstacles[i], _goal_rR) < R){
    key_return = false;
    break;
  }

}


// Paso 2. Eliminar de L cada punto k que:

//       a) Forme parte de BL o BR
```

```
//       b) d(Xk, Xrobot) > d(Xgoal, Xrobot)

//       c) d(Xk, P) > 2R


// 2a) Simplemente basta con ignorar BL y BR

// 2b) Hacemos las comprobaciones con FR y FL y vamos
   modificandolos
// y haciendo la copia en otro array

if(key_return == true){

  int c_FR_aux = 0; // Nuevos contadores auxiliares para las
     zonas FR y FL
  int c_FL_aux = 0;

  struct point points_FR_aux[c_FR + 1]; // Nuevas estructuras
     de puntos auxiliares para las zonas FR y FL
  struct point points_FL_aux[c_FL + 1];

  float angles_FR_aux[c_FR + 1];
  float angles_FL_aux[c_FL + 1];

  // Primero para FR

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

    if(distBetween2Points(points_FR[i], origin) <
       distBetween2Points(_X_robot, _X_goal)){
      points_FR_aux[c_FR_aux] = points_FR[i];
      angles_FR_aux[c_FR_aux] = angles_FR[i];
      c_FR_aux++;
    }

  }

  // Segundo para FL

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

    if(distBetween2Points(points_FL[i], origin) <
       distBetween2Points(_X_robot, _X_goal)){
      points_FL_aux[c_FL_aux] = points_FL[i];
      angles_FL_aux[c_FL_aux] = angles_FL[i];
      c_FL_aux++;
    }

  }

  // 2c) Eliminar aquellos puntos cual distancia d(x, P) > R
```

```
float pendiente = (_X_goal.y - _X_robot.y)/(_X_goal.x -
    _X_robot.x);
float b = 0;
float perpendicularpendiente = -(1/pendiente);
float perpendicularB;
float x_interseccion, y_interseccion;
float dist_x, dist_y, dist;

int c_FR_aux2 = 0;
int c_FL_aux2 = 0;

struct point points_FR_aux2[c_FR_aux + 1];
struct point points_FL_aux2[c_FL_aux + 1];

float angles_FR_aux2[c_FR_aux + 1];
float angles_FL_aux2[c_FL_aux + 1];

// Primero para FR.

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

  perpendicularB = points_FR_aux[i].y -
     perpendicularpendiente*points_FR_aux[i].x;

  x_interseccion = (perpendicularB-b)/(pendiente-
     perpendicularpendiente);

  y_interseccion = x_interseccion*pendiente + b;

  dist_x = x_interseccion - points_FR_aux[i].x;

  dist_y = y_interseccion - points_FR_aux[i].y;

  dist = sqrt(dist_x*dist_x + dist_y*dist_y);

  if(dist <= R){

    points_FR_aux2[c_FR_aux2] = points_FR_aux[i];
    angles_FR_aux2[c_FR_aux2] = angles_FR_aux[i];

    c_FR_aux2++;

  }

}

// Segundo para FL.

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

  perpendicularB = points_FL_aux[i].y -
     perpendicularpendiente*points_FL_aux[i].x;
```

```
      x_interseccion = (perpendicularB-b)/(pendiente-
         perpendicularpendiente);

      y_interseccion = x_interseccion*pendiente + b;

      dist_x = x_interseccion - points_FL_aux[i].x;

      dist_y = y_interseccion - points_FL_aux[i].y;

      dist = sqrt(dist_x*dist_x + dist_y*dist_y);

      if(dist <= R){

        points_FL_aux2[c_FL_aux2] = points_FL_aux[i];
        angles_FL_aux2[c_FL_aux2] = angles_FL_aux[i];

        c_FL_aux2++;

      }

    }

    // Paso 3. Si para todos los puntos que hayan quedado de FR
       y FL,
    // se cumple que la distancia entre ellos sea más grando que
        2R,
    // entonces Xgoal puede ser alcanzado. Si no se cumple para
       todos,
    // entonces Xgoal no puede ser alcanzado.
    // Los puntos a comparar se encuentran en points_FR_aux2 y
       points_FL_aux2

    if(c_FR_aux2 > 0 || c_FL_aux2 > 0 ){
      key_return = false;
    }

  }

  return key_return;
}
```

```
// Método f_sat.
// Implementación de la función de saturación f_sat.

float cg_algorithm::f_sat(float a, float b, float x){

  float v_return;

  if(x <= a) {
```

```
    v_return = a;

  }else if((a < x) && (b > x)){

    v_return = x;

  }else{

    v_return = b;

  }

  return v_return;

}
```

```
// Método convert02PI.
// Permite convertir un ángulo cualquiera entre 0 y 2pi.

float cg_algorithm::convert02PI (float angle){

  if(angle < 0){

    angle = twoPi - abs(angle);

  }

  return angle;

}
```

```
// Método f_proj.
// Devuelve la proyección de un ángulo cualquiera.

float cg_algorithm::f_proj(float angle){

  return (minDistBetween2Angles(angle, PI)) - PI;

}
```

```
// Método ConvertFtS.
// Permite convertir un número de Float a String.

std::string cg_algorithm::ConvertFtS (float number){
  std::ostringstream buff;
  buff<<number;
```

```
  return buff.str();
}
```

```
// Método ConvertItS.
// Permite convertir un número de Integer a String.

std::string cg_algorithm::ConvertItS (int number){
  std::ostringstream buff;
  buff<<number;
  return buff.str();
}
```

```
// Método calculate_xy.
// Se encarga de calcular las coordenadas de todos los elementos
    que se
// representan gráficamente en la interfaz de la App Android.

std::string cg_algorithm::calculate_xy(struct info_scan *
   info_scan){

  // Coordenadas correspondientes a todos los puntos detectados
     como obstáculos por el láser.
  float xp[laser_scans+1];
  float yp[laser_scans+1];

  // Coordenadas de cada extremos de cada gap.
  float xp_gaps_i[info_scan->n_gaps+1];
  float yp_gaps_i[info_scan->n_gaps+1];
  float xp_gaps_j[info_scan->n_gaps+1];
  float yp_gaps_j[info_scan->n_gaps+1];

  // Coordenadas correspondientes con la flecha que se dibujará
     indicando hacia dónde se encuentra
  // orientado el robot
  float xp_robot;
  float yp_robot;
  float xp_robot2;
  float yp_robot2;
  float xp_robot3;
  float yp_robot3;


  // Coordenadas correspondientes con la flecha que se dibujará
     indicando hacia dónde se tiene
  // que orientar el robot.
  float xp_a_traj;
  float yp_a_traj;
  float xp_a_traj2;
```

```
float yp_a_traj2;
float xp_a_traj3;
float yp_a_traj3;

// Coordenadas del punto objetivo
float xp_T;
float yp_T;

float dist_T = distBetween2Points(X_robot.x, X_robot.y, X_goal
    .x, X_goal.y);

if(dist_T > laser.range_max){

  xp_T = (X_goal.x - X_robot.x)/(dist_T/(laser.range_max*1.06)
      );
  yp_T = (X_goal.y - X_robot.y)/(dist_T/(laser.range_max*1.06)
      );

}else{

  xp_T = X_goal.x - X_robot.x;
  yp_T = X_goal.y - X_robot.y;

}

xp_a_traj = 3*cos(a_traj);
yp_a_traj = 3*sin(a_traj);

xp_a_traj2 = 3*cos(a_traj+d2r(5));
yp_a_traj2 = 3*sin(a_traj+d2r(5));

xp_a_traj3 = 3*cos(a_traj+d2r(-5));
yp_a_traj3 = 3*sin(a_traj+d2r(-5));

xp_robot = 1.5*cos(orientation_robot);
yp_robot = 1.5*sin(orientation_robot);

xp_robot2 = 1.5*cos(orientation_robot+d2r(15));
yp_robot2 = 1.5*sin(orientation_robot+d2r(15));

xp_robot3 = 1.5*cos(orientation_robot+d2r(-15));
yp_robot3 = 1.5*sin(orientation_robot+d2r(-15));

string c_x = "";
string c_y = "";

int x = 0;

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

  if(laser.ranges[i]<laser.range_max){

    xp[x] = laser.ranges[i]*cos((i*laser.angle_increment)+
```

```
        orientation_robot-d2r(field/2));
    yp[x] = laser.ranges[i]*sin((i*laser.angle_increment)+
        orientation_robot-d2r(field/2));

    x++;

  }

}


points_xy_gaps(xp_gaps_i, yp_gaps_i, xp_gaps_j, yp_gaps_j,
   info_scan);

// Coordenada (x) orientacion robot
// Coordenadas (x) de todos los puntos

c_x = c_x + ConvertFtS(xp_robot) + " " + ConvertFtS(xp_robot2)
    + " " + ConvertFtS(xp_robot3) + " " + "f";


if(x > 0){

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

    if(i == (x - 1)){

      c_x = c_x + ConvertFtS(xp[i]) + " " + "f";

    }else{

      c_x = c_x + ConvertFtS(xp[i]) + " ";

    }

  }

}else{

  c_x = c_x + "f";

}

// Coordenada (y) orientacion robot
// Coordenadas (y) de todos los puntos

c_y = c_y + ConvertFtS(yp_robot) + " " + ConvertFtS(yp_robot2)
    + " " + ConvertFtS(yp_robot3) + " " + "f";

if(x > 0){

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

```
    if(i == (x - 1)){
      c_y = c_y + ConvertFtS(yp[i]) + " " + "f";
    }else{
      c_y = c_y + ConvertFtS(yp[i]) + " ";
    }
  }
}else{
  c_y = c_y + "f";
}
// Coordenadas (x) de los puntos i y j de los gaps
if(info_scan->n_gaps > 0){
  for(int i = 0; i < info_scan->n_gaps; i++){
    if(i == (info_scan->n_gaps - 1)){
      c_x = c_x + ConvertFtS(xp_gaps_i[i]) + " " + "f";
    }else{
      c_x = c_x + ConvertFtS(xp_gaps_i[i]) + " ";
    }
  }

  for(int i = 0; i < info_scan->n_gaps; i++){
    if(i == (info_scan->n_gaps - 1)){
      c_x = c_x + ConvertFtS(xp_gaps_j[i]) + " " + "f";
    }else{
      c_x = c_x + ConvertFtS(xp_gaps_j[i]) + " ";
    }
  }

  // Coordenadas (y) de los puntos i y j de los gaps
  for(int i = 0; i < info_scan->n_gaps; i++){
```

```
    if(i == (info_scan->n_gaps - 1)){

      c_y = c_y + ConvertFtS(yp_gaps_i[i]) + " " + "f";

    }else{

      c_y = c_y + ConvertFtS(yp_gaps_i[i]) + " ";

    }
  }

  for(int i = 0; i < info_scan->n_gaps; i++){

    if(i == (info_scan->n_gaps - 1)){

      c_y = c_y + ConvertFtS(yp_gaps_j[i]) + " " + "f";

    }else{

      c_y = c_y + ConvertFtS(yp_gaps_j[i]) + " ";

    }
  }



}else{
  c_x = c_x + "f";
  c_x = c_x + "f";
  c_y = c_y + "f";
  c_y = c_y + "f";
}


c_x = c_x + ConvertFtS(xp_T) + " " + "f";

c_y = c_y + ConvertFtS(yp_T) + " " + "f";

c_y = c_y + ConvertItS(x1) + " " + "f";

c_y = c_y + ConvertFtS(xp_a_traj) + " " + ConvertFtS(yp_a_traj
    ) + " " + ConvertFtS(xp_a_traj2) + " " + ConvertFtS(
    yp_a_traj2) + " " + ConvertFtS(xp_a_traj3) + " " +
    ConvertFtS(yp_a_traj3) + " " + "f";

c_y = c_y + ConvertFtS(R) + " " + "f";

c_y = c_y + ConvertFtS(Ds+R) + " " + "f";

c_y = c_y + ConvertFtS(laser.range_max) + " " + "fF";
```

```
  return (c_x+c_y);

}
```

```
// Método init.
// Inicialización de todas las variables que el algoritmo
   utiliza.

void cg_algorithm::init(struct config init_config){

  R = init_config.R;
  Ds = init_config.Ds;
  k = init_config.k;
  laser = init_config.laser;
  X_goal = init_config.X_goal;
  X_robot = init_config.X_robot;
  orientation_robot = init_config.orientation_robot;
  v_max = init_config.v_max;
  w_max = init_config.w_max;
  Dvs = init_config.Dvs;

  //Angulo que abarca el sensor
  field = r2d(laser.angle_max-laser.angle_min);

  //posiciones del array. Tomas que ha cogido el laser
  laser_scans = field/r2d(laser.angle_increment);

}
```

```
// Método reserv_memory.
// Se encarga de reservar la memoria que utilizarán todas las
   struct del algoritmo

void cg_algorithm::reserv_memory(struct info_scan *scan_forward,
    struct info_scan *scan_backward, struct info_scan *
   list_of_gaps){

  scan_forward->points_i = (int*) malloc((laser_scans+1)*sizeof(
     int));
  scan_forward->points_j = (int*) malloc((laser_scans+1)*sizeof(
     int));
  scan_forward->dist_i = (float*) malloc((laser_scans+1)*sizeof(
     float));
  scan_forward->dist_j = (float*) malloc((laser_scans+1)*sizeof(
     float));
  scan_forward->dist_gap = (float*) malloc((laser_scans+1)*
     sizeof(float));
```

```
  scan_backward->points_i = (int*) malloc((laser_scans+1)*sizeof
      (int));
  scan_backward->points_j = (int*) malloc((laser_scans+1)*sizeof
      (int));
  scan_backward->dist_i = (float*) malloc((laser_scans+1)*sizeof
      (float));
  scan_backward->dist_j = (float*) malloc((laser_scans+1)*sizeof
      (float));
  scan_backward->dist_gap = (float*) malloc((laser_scans+1)*
      sizeof(float));

  list_of_gaps->points_i = (int*) malloc((laser_scans+1)*sizeof(
      int));
  list_of_gaps->points_j = (int*) malloc((laser_scans+1)*sizeof(
      int));
  list_of_gaps->dist_i = (float*) malloc((laser_scans+1)*sizeof(
      float));
  list_of_gaps->dist_j = (float*) malloc((laser_scans+1)*sizeof(
      float));
  list_of_gaps->dist_gap = (float*) malloc((laser_scans+1)*
      sizeof(float));

}
```

```
// Método liber_memory.
// Se encarga de liberar la memoria que utilizarán todas las
    struct del algoritmo

void cg_algorithm::liber_memory(struct info_scan *scan_forward,
    struct info_scan *scan_backward, struct info_scan *
    list_of_gaps){

  free(scan_forward->points_i);
  free(scan_forward->points_j);
  free(scan_forward->dist_i);
  free(scan_forward->dist_j);
  free(scan_forward->dist_gap);

  free(scan_backward->points_i);
  free(scan_backward->points_j);
  free(scan_backward->dist_i);
  free(scan_backward->dist_j);
  free(scan_backward->dist_gap);

  free(list_of_gaps->points_i);
  free(list_of_gaps->points_j);
  free(list_of_gaps->dist_i);
  free(list_of_gaps->dist_j);
  free(list_of_gaps->dist_gap);
```

```
  free(scan_forward);
  free(scan_backward);
  free(list_of_gaps);

}
```

```
// Método step1.
// Implementación de la primera fase del algoritmo CG. Se
   necesita una
// estrutura del tipo info_scan y un bool indicando si se desea
   realizar
// el barrido forward o el barrido backward. De forma interna
   este método
// ya guarda la información resultante con respecto a cada
   barrido
// que se realice.

void cg_algorithm::step1(struct info_scan *_scan, bool type_scan
   ){

  //posiciones(ángulo) a recorrer en el array para cada scan i.
  int d = ((int)laser_scans*180)/(int)field;

  float dist_ij; //distancia entre puntos. Forward -> dist i-j.
     Backward -> dist j-i
  float dist_ij_min = 10000; // distancia minima que
     encontraremos entre i-j / j-i

  int number_of_gaps = 0; // cantidad de gaps encontrados.

  //Variables auxiliares.
  int i_aux;
  int j_aux;
  int punto_i[laser_scans+1];
  int punto_j[laser_scans+1];
  float dist_i[laser_scans+1];
  float dist_j[laser_scans+1];
  float dist[laser_scans+1];
  float laser_ranges_aux[laser_scans+1]; // Variable auxiliar
     que utilizaremos para hacer la copia de los datos del
     sensor en el scan backward.
  float dist_i_aux;
  float dist_j_aux;
  bool key = false; // Para saber si hay que guardar algun gap o
      no.

  int x=1; // Variable para saber desde donde hay que reanudar
     la busqueda cada vez que se encuentra una discontinuidad.
```

```
int d_aux[laser_scans+1];

for(int i=0; i<=laser_scans; i++){
  punto_i[i]=0;
  punto_j[i]=0;
  dist[i]=0;
}

// Si se desea realizar el barrido backward hay que girar todo
    el scan para que así se aproveche
// el mismo bucle utilizado en el scan forward.

if(type_scan == BACKWARD){

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

    laser_ranges_aux[i] = laser.ranges[laser_scans-i];

  }

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

    laser.ranges[i] = laser_ranges_aux[i];

  }

}

for (int i=0; i<=laser_scans; i=i+x){

  if (laser_scans-i<=d){
    d=(int)laser_scans-i;
  }else{
    d=((int)laser_scans*180)/(int)field;;
  }

  if((laser.ranges[i+1]-laser.ranges[i])>(2*R)){ //
     Discontinuidad de tipo 1.

    for (int j = 1; j <= d; j++){

      if (laser.ranges[i+j] < laser.range_max){

        dist_ij = sqrt((laser.ranges[i]*laser.ranges[i])+(
           laser.ranges[i+j]*laser.ranges[i+j])-(2*laser.
           ranges[i]*laser.ranges[i+j]*cos(j*laser.
           angle_increment)));

        if ((dist_ij <= dist_ij_min)){ // Se ha encontrado la
           segunda parte del gap. Se va actualizando cada vez
           a medida que la distancia se hace pequeña
          // Al llegar realmente a la distancia más corta ya
             no se entrarán más veces y se quedará guardada la
```

```
              distancia
            // más pequeña.
            dist_ij_min = dist_ij;

            j_aux = i+j;
            i_aux = i;
            dist_i_aux=laser.ranges[i];
            dist_j_aux=laser.ranges[i+j];

            key = true;

          }

        }
      }

    }else if((laser.ranges[i+1]==laser.range_max)&&(laser.ranges
       [i]<laser.range_max)){ // Discontinuidad de tipo 2.

      for (int j = 1; j <= d; j++){

        if (laser.ranges[i+j] < laser.range_max){ // Se ha
           encontrado la segunda parte del gap.

          dist_ij = sqrt((laser.ranges[i]*laser.ranges[i])+(
             laser.ranges[i+j]*laser.ranges[i+j])-(2*laser.
             ranges[i]*laser.ranges[i+j]*cos(j*laser.
             angle_increment)));

          dist_ij_min = dist_ij;

          j_aux = i+j;
          i_aux = i;
          dist_i_aux=laser.ranges[i];
          dist_j_aux=laser.ranges[i+j];

          key = true;

          break;

        }
      }

    }

    if(key==true){ // Se guarda el gap

      punto_i[number_of_gaps]=i_aux;
      punto_j[number_of_gaps]=j_aux;
      dist_i[number_of_gaps]=dist_i_aux;
      dist_j[number_of_gaps]=dist_j_aux;
      dist[number_of_gaps] = dist_ij_min;
      number_of_gaps++;
```

```
    d_aux[number_of_gaps] = d;

    x = j_aux+1;

  }else{
    x=1;
  }

  key = false;

  dist_ij_min = 10000;
}


for(int i = 0; i <= laser_scans; i++){
  _scan->points_i[i]=0;
  _scan->points_j[i]=0;
  _scan->dist_i[i]=0;
  _scan->dist_j[i]=0;
  _scan->dist_gap[i]=0;

}

// Se guardan los gaps de forma interna

for(int i = 0; i < number_of_gaps; i++){
  _scan->points_i[i]=punto_i[i];
  _scan->points_j[i]=punto_j[i];
  _scan->dist_i[i]=dist_i[i];
  _scan->dist_j[i]=dist_j[i];
  _scan->dist_gap[i]=dist[i];

}

_scan->n_gaps = number_of_gaps;

// En caso se haber utilizado el barrido backward se vuelve a
   dejar scan en su orden original para que futuros métodos
// puedan utilizarlo sin problemas.

if(type_scan == BACKWARD){

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

    laser_ranges_aux[i] = laser.ranges[laser_scans-i];

  }

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

    laser.ranges[i] = laser_ranges_aux[i];

  }
```

```
  }
}
```

```
// Método step2.
// Implementación de la segunda fase del algoritmo CG. Se
   necesita la
// estructura definitiva donde se guardará la lista de gaps
   definitiva
// (list_of_gaps) y también se necesita la información de los
   dos
// que se tienen que realizar de forma previa (scan_forward y
   scan_backward).

void cg_algorithm::step2(struct info_scan *list_of_gaps, struct
   info_scan *scan_forward, struct info_scan *scan_backward){

  int punto_i_1[laser_scans+1];
  int punto_j_1[laser_scans+1];
  float dist_i_1[laser_scans+1];
  float dist_j_1[laser_scans+1];
  float dist_1[laser_scans+1];

  int punto_i_2[laser_scans+1];
  int punto_j_2[laser_scans+1];
  float dist_i_2[laser_scans+1];
  float dist_j_2[laser_scans+1];
  float dist_2[laser_scans+1];

  int x=0;

  int w1=0;

  if((scan_forward->n_gaps==0)&&(scan_backward->n_gaps==0)){

    ROS_INFO("No se ha encontrado ningun gap");

    list_of_gaps->n_gaps = 0;

  }else{

    // Se giran los indices del scan backward para que coincidan
        con los del scan forward.

    for(int i = 0; i < scan_backward->n_gaps; i++){

      int aux_i=scan_backward->points_i[i];
      float aux_i2 = scan_backward->dist_i[i];

      scan_backward->points_i[i]=laser_scans-scan_backward->
```

```
        points_j[i];
    scan_backward->points_j[i]=laser_scans-aux_i;

    scan_backward->dist_i[i]=scan_backward->dist_j[i];
    scan_backward->dist_j[i]=aux_i2;
  }

  // Se pone toda la información de los gaps (tanto de los
     forward como backward) en un mismo array y se eliminan
     aquellos más pequeños que 2R.

  for(int i = 0; i < scan_forward->n_gaps; i++){

    if(scan_forward->dist_gap[i] > 2*R){

      punto_i_1[x]=scan_forward->points_i[i];
      punto_j_1[x]=scan_forward->points_j[i];
      dist_i_1[x]=scan_forward->dist_i[i];
      dist_j_1[x]=scan_forward->dist_j[i];
      dist_1[x]=scan_forward->dist_gap[i];

      x++;
    }

  }

  for(int i = 0; i < scan_backward->n_gaps; i++){

    if(scan_backward->dist_gap[i]>2*R){

      punto_i_1[x]=scan_backward->points_i[i];
      punto_j_1[x]=scan_backward->points_j[i];
      dist_i_1[x]=scan_backward->dist_i[i];
      dist_j_1[x]=scan_backward->dist_j[i];
      dist_1[x]=scan_backward->dist_gap[i];

      x++;
    }

  }

  // Se eliminan gaps que esten dentro de otros gaps

  bool key;
  float dist_equals[laser_scans+1];
  int c_equals=0;
  bool key_equals = false;

  int punto_i_equal[laser_scans+1];
  int punto_j_equal[laser_scans+1];

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

```
    key=false;
    key_equals = false;

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

      if((j != i)&&((punto_i_1[i]>=punto_i_1[j])&&(punto_j_1[i
          ]<=punto_j_1[j]))){

        key = true;

        if((punto_i_1[i]==punto_i_1[j])&&(punto_j_1[i]==
            punto_j_1[j])){ // Solo en el caso de que sean
            exactamente iguales

          for(int w=0; w < c_equals; w++){

            if((punto_i_1[i]==punto_i_equal[w])&&(punto_j_1[i
                ]==punto_j_equal[w])){

              key_equals = true;
              key=false;


            }

          }

          if(key_equals == false){

            punto_i_equal[c_equals]=punto_i_1[i];
            punto_j_equal[c_equals]=punto_j_1[i];
            c_equals++;
          }

        }


      }

    }

    if(key==false){

      punto_i_2[w1]=punto_i_1[i];
      punto_j_2[w1]=punto_j_1[i];
      dist_i_2[w1]=dist_i_1[i];
      dist_j_2[w1]=dist_j_1[i];
      dist_2[w1]=dist_1[i];

      w1++;
    }
  }
```

```
    // Lista final de gaps

    for(int i = 0; i < w1; i++){

      list_of_gaps->points_i[i]=punto_i_2[i];

      list_of_gaps->points_j[i]=punto_j_2[i];

      list_of_gaps->dist_i[i]=dist_i_2[i];

      list_of_gaps->dist_j[i]=dist_j_2[i];

      list_of_gaps->dist_gap[i]=dist_2[i];

    }

    list_of_gaps->n_gaps = w1;

  }

}
```

```
// Método calculate_md.
// Implementación de la tercera fase del algoritmo CG. Se
   procede a calcular
// la dirección de movimiento (ángulo a_md) que el robot debe
   seguir. Se
// necesita la lista de gaps definitiva para poder realizar este
    paso

void cg_algorithm::calculate_md(struct info_scan *list_of_gaps){

  n_gaps_aux = list_of_gaps->n_gaps;

  struct point origin; // Origen. Nos servirá para realizar
     distintos cálculos ya que nos interesa conocer la distancia
      de algunos puntos respecto al robot.
  origin.x = 0;
  origin.y = 0;

  struct point mid_gap; // Posición punto medio de X gap.
```

```
struct point list_obstacles[laser_scans+1]; // Estructura de
   puntos que nos definirá los obstaculos que el robot está
   viendo.

float a_goal; // ángulo hacia punto objetivo.
float a_cs; // ángulo del closest gap mas cerca del objetivo.
float a_os; // el otro angulo del gap.


bool is_navigable = false; // Comprobaciones para saber si es
   navegable.
bool is_direct_navigable = false;


// Coordenadas de los puntos i e j de cada gap encontrado.
float xp_gaps_i[n_gaps_aux+1];
float yp_gaps_i[n_gaps_aux+1];
float xp_gaps_j[n_gaps_aux+1];
float yp_gaps_j[n_gaps_aux+1];

// Angulos en los que se encuentran cada extremo i e j.

a_gaps_i = (float*) malloc((n_gaps_aux+1)*sizeof(float));
a_gaps_j = (float*) malloc((n_gaps_aux+1)*sizeof(float));

// Mismas variables pero auxiliares, ya que se necesita
   modificar arrays.
// y por lo tanto se necesitan copias

float xp_gaps_i_aux[n_gaps_aux+1];
float yp_gaps_i_aux[n_gaps_aux+1];
float xp_gaps_j_aux[n_gaps_aux+1];
float yp_gaps_j_aux[n_gaps_aux+1];

float xp_gaps_i_aux2[n_gaps_aux+1];
float yp_gaps_i_aux2[n_gaps_aux+1];
float xp_gaps_j_aux2[n_gaps_aux+1];
float yp_gaps_j_aux2[n_gaps_aux+1];

float a_gaps_i_aux[n_gaps_aux+1];
float a_gaps_j_aux[n_gaps_aux+1];

float a_gaps_i_aux2[n_gaps_aux+1];
float a_gaps_j_aux2[n_gaps_aux+1];

float a_gapselected_i;
float a_gapselected_j;

int remaining_gaps; // Número de gaps que van quedando a
   medida que los eliminamos por si no cumplen las
   restricciones.

float a_min = 1000; // Variable auxiliar para ir guardando el
```

```
    angulo mínimo.

int x2; // para saber que parte del gap (i o j) está más cerca
     del punto objetivo.
float a_aux; // ángulo auxiliar para realizar cálculo
    intermedio.

float Dcs; // Distancia desde el centro del robot hasta el
    lado más cercano al goal del gap seleccionado.

bool goal_inside = false; // Para poder saber si a_goal se
    encuentra dentro del gap seleccionado.

int cont_aux_z = 0; // Contador auxiliar para poder hacer
    correctamente la copia de gaps en
// otro array por si se da el caso de que el array
    seleccionado no es navegable

float a_scs; // Angulo que se va a asignar en primera
    instancia a a_md si se da el caso de
// que el gap seleccionado sea amplio.

float a_mid; // Angulo que se va a asignar en primera
    instancia a a_md si se da el caso de
// que el gap seleccionado sea estrecho.

float Dns; // Distancia hasta el lado del gap seleccionado mas
     cercano al robot.
float Dns1; // Dns auxiliar para calcular la distancia hasta
    la parte i.
float Dns2; // Dns auxiliar para calcular la distancia hasta
    la parte j.

float w;   // Anchura real del gap que el robot estará viendo.
float alpha; // Ángulo de corrección que se le aplica a a_md.
float fi;  // Ángulo mínimo para que el robot quepa por el gap
    .
float beta;  // Beta siempre valdrá el doble de fi.

bool key_direct = false;

d_min = 1000; // Distancia mínima a la que se encuentra el
    obstaculo más cercano.

a_goal = atan2((X_goal.y-X_robot.y),(X_goal.x-X_robot.x)); //
    Angulo goal respecto a la posición del robot.

// Crear lista de obstaculos

n_obstacles = 0;

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

```
    if(laser.ranges[i] < laser.range_max){

      myfile2 << laser.ranges[i] << " "; // Sacamos los puntos
         detectados por el láser en el caso de que no de el
         rango máximo del sensor.

      list_obstacles[n_obstacles].x =  laser.ranges[i]*cos((i*
         laser.angle_increment)+orientation_robot/*car_pose.pose
         .pose.orientation.w*/-d2r(field/2));
      list_obstacles[n_obstacles].y =  laser.ranges[i]*sin((i*
         laser.angle_increment)+orientation_robot/*car_pose.pose
         .pose.orientation.w*/-d2r(field/2));

      n_obstacles++;

    }else{

      myfile2 << 0 << " "; // Si devuelve el rango máximo del lá
         ser ponemos un 0.

    }


  }

  myfile2 << "\n"; // Salto de línea en el fichero.

  if(n_obstacles > 0){

    is_direct_navigable = appendixA_ND_D(X_robot, X_goal,
       list_obstacles, n_obstacles);

    if(is_direct_navigable == false){

      if(n_gaps_aux > 0){

        // Calcular puntos x e y de cada uno de los dos extremos
            de los gaps.

        points_xy_gaps(xp_gaps_i, yp_gaps_i, xp_gaps_j,
           yp_gaps_j, list_of_gaps);

        // Calcular angulo en el cual se encuentra cada extremo
           de cada gap.

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

          a_gaps_i[i] = atan2(yp_gaps_i[i], xp_gaps_i[i]);
          a_gaps_j[i] = atan2(yp_gaps_j[i], xp_gaps_j[i]);

        }

        // A partir de este punto vamos a encontrar que gap más
```

```
        cercano al a_goal es navegable.

      // Primero copiamos los gaps en otra cadena para poder
         modificarlos mejor. Copiamos tanto las coordenadas
         como los ángulos.

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

        xp_gaps_i_aux[i] = xp_gaps_i[i];
        yp_gaps_i_aux[i] = yp_gaps_i[i];
        xp_gaps_j_aux[i] = xp_gaps_j[i];
        yp_gaps_j_aux[i] = yp_gaps_j[i];

        a_gaps_i_aux[i] = a_gaps_i[i];
        a_gaps_j_aux[i] = a_gaps_j[i];

      }

      remaining_gaps = n_gaps_aux;

      while(remaining_gaps > 0){

        is_navigable = false;
        a_min = 1000;

        //*****************************************
        //*****************************************
        //Paso 1. BUSCAMOS EL GAP MÁS CERCANO A A_GOAL
        //*****************************************
        //*****************************************

        // Buscamos el lado del gap que esté más cerca de
           a_goal. Cuando encontremos este lado ya sabremos
           cual es el gap.

        // Primero recorremos los puntos i.
        for(int i = 0; i < remaining_gaps; i++){

          a_aux = (minDistBetween2Angles(a_goal, a_gaps_i_aux[
             i]));

          if(a_aux < a_min){
            a_min = a_aux;
            x1 = i;
            x2 = 1;

          }
        }

        // Despues los puntos j.
        for(int i = 0; i < remaining_gaps; i++){

          a_aux = (minDistBetween2Angles(a_goal, a_gaps_j_aux[
```

```
            i]));

          if(a_aux < a_min){
            a_min = a_aux;
            x1 = i;
            x2 = 2;

          }
        }

        // Se guardan angulos del gap selected para poder
           representarlos en Android.

        a_gapselected_i = a_gaps_i_aux[x1];
        a_gapselected_j = a_gaps_j_aux[x1];

        if(x2 == 1){ // La parte i es la que esta mas cerca
           del goal.

          a_cs = a_gaps_i_aux[x1];
          a_os = a_gaps_j_aux[x1];

          Dcs = sqrt((xp_gaps_i_aux[x1]*xp_gaps_i_aux[x1]) + (
             yp_gaps_i_aux[x1]*yp_gaps_i_aux[x1]));

        }else{ // La parte j es la que esta mas cerca del goal
           .

          a_cs = a_gaps_j_aux[x1];
          a_os = a_gaps_i_aux[x1];

          Dcs = sqrt((xp_gaps_j_aux[x1]*xp_gaps_j_aux[x1]) + (
             yp_gaps_j_aux[x1]*yp_gaps_j_aux[x1]));

        }


        //****************************
        //****************************
        //Paso 2. COMPROBAMOS SI EL GAP
           SELECCIONADO ES NAVEGABLE
        //****************************
        //****************************

        // Dentro de este paso hay dos pasos incluidos.
           Primero hay que comprobar a ver si
        // a_goal se encuentra dentro del gap. En este caso
           habrá que comprobar si hay un
        // camino viable hasta el punto objetivo. La otra opci
           ón es que a_goal no se
        // encuentre dentro de el gap y por lo tanto habrá que
            comprobar si existe un
        // camino viable hasta el centro del gap.
```

```
        // Primero comprobamos a ver si a_goal se encuentra
           dentro del gap.

        goal_inside = angle_inside_2angles(a_goal, a_cs, a_os)
           ;

        // Si a_goal está dentro del gap miramos a ver si
           existe camino navegable hasta el punto objetivo.
        if(goal_inside == true){

          is_navigable = appendixA_ND(X_robot, X_goal,
             list_obstacles, n_obstacles);

          if(is_navigable == true){

            key_direct = true;

          }else{

            goal_inside = false;

          }

        }


        //Si a_goal NO está dentro del gap miramos a ver si
           existe camino navegable hasta el centro del gap.
        if(goal_inside == false){

          // Calculamos coordenadas del punto medio del gap
             seleccionado.
          // Notar que necesitamos estas coordenadas en modo
             absoluto ya
          // que al método appendixA_ND hay que pasarle las
             estructuras
          // de puntos en modo absoluto y el ya lo calculará
             en función al robot.

          mid_gap.x = X_robot.x + (xp_gaps_i_aux[x1] +
             xp_gaps_j_aux[x1])/2;
          mid_gap.y = X_robot.y + (yp_gaps_i_aux[x1] +
             yp_gaps_j_aux[x1])/2;

          is_navigable = appendixA_ND(X_robot, mid_gap,
             list_obstacles, n_obstacles);

        }

        if(is_navigable == true){

          break;
```

```
        }else{

          cont_aux_z = 0;

          // Hacemos la copia de los gaps que tienen que
             volver a ser tratados
          // y eliminamos el gap seleccionado que no es
             navegable.

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

            if(i != x1){ // No hay que copiar el gap no
               navegable.

              xp_gaps_i_aux2[cont_aux_z] = xp_gaps_i_aux[i];
              yp_gaps_i_aux2[cont_aux_z] = yp_gaps_i_aux[i];
              xp_gaps_j_aux2[cont_aux_z] = xp_gaps_j_aux[i];
              yp_gaps_j_aux2[cont_aux_z] = yp_gaps_j_aux[i];

              a_gaps_i_aux2[cont_aux_z] = a_gaps_i_aux[i];
              a_gaps_j_aux2[cont_aux_z] = a_gaps_j_aux[i];

              cont_aux_z++;
            }


          }

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

            xp_gaps_i_aux[i] = xp_gaps_i_aux2[i];
            yp_gaps_i_aux[i] = yp_gaps_i_aux2[i];
            xp_gaps_j_aux[i] = xp_gaps_j_aux2[i];
            yp_gaps_j_aux[i] = yp_gaps_j_aux2[i];

            a_gaps_i_aux[i] = a_gaps_i_aux2[i];
            a_gaps_j_aux[i] = a_gaps_j_aux2[i];

          }

          remaining_gaps--;

        }

      }

      // a_scs = a_cs +- arcsin((R+Ds)/Dcs).
      // a_cs ya calculado.
      // R radio robot.
      // Ds distancia de seguridad.
      // Dcs distancia hasta obstaculo que se encuentra en
         a_cs.
```

```
// Comprobamos si a_cs esta a la derecha o a la
   izquierda de a_os.

if(x2 == 1){ // si x2 == 1 entonces a_cs está a la
   derecha.

  if((R+Ds) > Dcs){

    a_scs = a_cs + asin(1);

  }else{

    a_scs = a_cs + asin((R+Ds)/(Dcs));

  }

  a_mid = a_cs + minDistBetween2Angles(a_cs, a_os)/2;

}else{

  if((R+Ds) > Dcs){

    a_scs = a_cs - asin(1);

  }else{

    a_scs = a_cs - asin((R+Ds)/(Dcs));

  }

  a_mid = a_cs - minDistBetween2Angles(a_cs, a_os)/2;

}

// Ahora hay que elegir el valor de a_md.

//      | a_goal, if a_rs <= a_goal <= a_ls
// a_md = | a_mid, if dist(a_cs, a_mid) < dist(a_cs,
   a_scs)
//        | a_scs, otherwise

if((angle_inside_2angles(a_goal, a_gaps_i_aux[x1],
   a_gaps_j_aux[x1]) == true) && (key_direct == true)){

  a_md = a_goal;

}else if(minDistBetween2Angles(a_cs, a_mid) <
   minDistBetween2Angles(a_cs, a_scs)){

  a_md = a_mid; // gap is narrow.

}else{
```

```
        a_md = a_scs; // gap is wide.

      }

      // Dns es la distancia desde el robot hasta la parte del
          gap mas cercana a él.
      // Dns1 parte derecha del gap.
      // Dns2 parte izquierda del gap.

      Dns1 = sqrt((xp_gaps_i_aux[x1]*xp_gaps_i_aux[x1]) + (
         yp_gaps_i_aux[x1]*yp_gaps_i_aux[x1]));
      Dns2 = sqrt((xp_gaps_j_aux[x1]*xp_gaps_j_aux[x1]) + (
         yp_gaps_j_aux[x1]*yp_gaps_j_aux[x1]));

      if(Dns1 < Dns2){

        Dns = Dns1;

      }else{

        Dns = Dns2;

      }


      fi = asin(R/Dns);

      beta = 2*fi;

      w = minDistBetween2Angles(a_cs, a_os);

      // alpha = sat[0,beta](beta-w).

      // funcion sat:
      //                | a, if x<= a
      // sat[a,b](x) = | x, if a < x < b
      //                | b, if x >= b

      //alpha = f_sat(float a, float b, float x).

      alpha = f_sat(0.0, beta, (beta - w));

      if(alpha != 0){ // Si alpha=0 no hace falta modificar
         a_md.

        if(Dns2 < Dns1){

          a_md =  a_md - alpha;

        }else{

          a_md =  a_md + alpha;
```

```
          }

        }

      }else{ // Si no hay gaps se avanza hasta el punto objetivo
          .

        a_md = a_goal;

      }

    }else{ // Caso en el que hay un camino directo y navegable
       hasta goal.

      a_md = a_goal;

    }

  }else{ // Caso en el que no hay obstáculos.

    a_md = a_goal;

  }

}
```

```
// Método navigation_method.
// Implementación de la cuarta fase del algoritmo CG. Se procede
    a aplicar
// el método de navegación reactiva. Se debe introducir un array
    que contenga
// las variables de velocidad lineal y angular. El método
   calculará el
// valor de estas velocidades y lo guardará en la memoria.

void cg_algorithm::navigation_method(float *velocidades){

  float threat[laser_scans + 1]; // Lista de threats (amenazas)
     que se encuentran alrededor del robot.

  float angles_obstacles[n_obstacles];
  float angles_left[n_obstacles];
  float angles_right[n_obstacles];

  float weight_left[n_obstacles];
  float weight_right[n_obstacles];

  float deflections_left[n_obstacles];
```

```
float deflections_right[n_obstacles];


struct point origin; // Punto origen.
origin.x = 0;
origin.y = 0;

struct point list_obstacles[laser_scans+1]; // Estructura de
   puntos que define los obstaculos que el robot está viendo.

// Crear lista de obstaculos.

n_obstacles = 0;

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

  if(laser.ranges[i] < laser.range_max){

    myfile2 << laser.ranges[i] << " ";

    list_obstacles[n_obstacles].x =  laser.ranges[i]*cos((i*
       laser.angle_increment)+orientation_robot-d2r(field/2));
    list_obstacles[n_obstacles].y =  laser.ranges[i]*sin((i*
       laser.angle_increment)+orientation_robot-d2r(field/2));

    n_obstacles++;

  }else{

    myfile2 << 0 << " ";

  }


}

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

  angles_obstacles[i] = atan2(list_obstacles[i].y,
     list_obstacles[i].x);

  threat[i] = f_sat(0.0, 1.0, (((R+Ds) - (distBetween2Points(
     origin, list_obstacles[i])))/(R+Ds)));

}

// Ahora dividimos el espacio en dos regiones. Una a la
   izquierda de a_md y la
// otra a la derecha de a_md

struct point A;
struct point B;
struct point C;
```

```
struct point D;

A.x = 3*cos(a_md);
A.y = 3*sin(a_md);

B.x = 3*cos(a_md - PI/2);
B.y = 3*sin(a_md - PI/2);

C.x = 3*cos(a_md - PI);
C.y = 3*sin(a_md - PI);

D.x = 3*cos(a_md - (3*PI)/2);
D.y = 3*sin(a_md - (3*PI)/2);

float a_A = atan2(A.y, A.x);
float a_B = atan2(B.y, B.x);
float a_C = atan2(C.y, C.x);
float a_D = atan2(D.y, D.x);

// A partir de ahora tendemos 4 arrays guardando en cada uno
   los puntos que se encuentren respectivamente en FL, FR, BL
   y BR.
// Entre A-B -> FR
// Entre B-C -> BR
// Entre C-D -> BL
// Entre D-A -> FL

int c_RIGHT = 0;
int c_LEFT = 0;

struct point points_LEFT[n_obstacles];
struct point points_RIGHT[n_obstacles];

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

  if(distBetween2Points(origin, list_obstacles[i]) <= (R+Ds)){
      // INSIDE DS. IMPORTANT

    if(angle_inside_2angles(angles_obstacles[i], a_A, a_B) ==
       true){ // Comprobar si está entre A y B -> FR

      points_RIGHT[c_RIGHT] = list_obstacles[i];
      weight_right[c_RIGHT] = 1/(pow(1-threat[i], k));
      angles_right[c_RIGHT] = angles_obstacles[i];
      deflections_right[c_RIGHT] = threat[i]*f_proj(
         minDistBetween2Angles(angles_obstacles[i] + PI, a_md)
         );
      c_RIGHT++;

    }else if(angle_inside_2angles(angles_obstacles[i], a_B,
       a_C) == true){ // Comprobar si está entre B y C -> BR

      points_RIGHT[c_RIGHT] = list_obstacles[i];
```

```
      weight_right[c_RIGHT] = 1/(pow(1-threat[i], k));
      angles_right[c_RIGHT] = angles_obstacles[i];
      deflections_right[c_RIGHT] = threat[i]*f_proj(
         minDistBetween2Angles(angles_obstacles[i] + PI, a_md)
         );
      c_RIGHT++;

    }else if(angle_inside_2angles(angles_obstacles[i], a_C,
       a_D) == true){ // Comprobar si está entre C y D -> BL

      points_LEFT[c_LEFT] = list_obstacles[i];
      weight_left[c_LEFT] = 1/(pow(1-threat[i], k));
      angles_left[c_LEFT] = angles_obstacles[i];
      deflections_left[c_LEFT] = threat[i]*f_proj(
         minDistBetween2Angles(angles_obstacles[i] + PI, a_md)
         );
      c_LEFT++;

    }else{ // Si no se cumple ninguna anterior significa que
       está entre D y A -> FL

      points_LEFT[c_LEFT] = list_obstacles[i];
      weight_left[c_LEFT] = 1/(pow(1-threat[i], k));
      angles_left[c_LEFT] = angles_obstacles[i];
      deflections_left[c_LEFT] = threat[i]*f_proj(
         minDistBetween2Angles(angles_obstacles[i] + PI, a_md)
         );
      c_LEFT++;

    }

  }

}

// Ahora toca definir los pesos totales de los obstaculos de
   la parte izquierda y derecha.

float total_weight_left = 0;
float total_weight_right = 0;

float total_deflections_left = 0;
float total_deflections_right = 0;

float net_deflection;

int number_total_obstacles_inside_ds;


for(int i = 0; i < c_LEFT; i++){
  total_weight_left = total_weight_left + weight_left[i];
}
```

```
for(int i = 0; i < c_RIGHT; i++){
  total_weight_right = total_weight_right + weight_right[i];
}

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

  total_deflections_left = total_deflections_left + (
      weight_left[i]/total_weight_left)*(deflections_left[i]);

}

for(int i = 0; i < c_RIGHT; i++){
  total_deflections_right = total_deflections_right + (
      weight_right[i]/total_weight_right)*(deflections_right[i
      ]);
}

number_total_obstacles_inside_ds = (c_LEFT+c_RIGHT);

if((number_total_obstacles_inside_ds > 0) && (c_LEFT > 0)){
  total_deflections_left = -(total_deflections_left*
      number_total_obstacles_inside_ds)/(c_LEFT);
}

if((number_total_obstacles_inside_ds > 0) && (c_RIGHT > 0)){
  total_deflections_right = (total_deflections_right*
      number_total_obstacles_inside_ds)/(c_RIGHT);
}

if(c_RIGHT>0 || c_LEFT>0){

  net_deflection = ((total_weight_left*total_deflections_left)
      +(total_weight_right*total_deflections_right))/(
      total_weight_left+total_weight_right);

  a_traj = a_md - net_deflection;

}else{
  a_traj = a_md;
}

for(int i = 0; i < n_obstacles; i++){
  if((distBetween2Points(origin, list_obstacles[i]) - R) <
      d_min){
    d_min = distBetween2Points(origin, list_obstacles[i]) - R;
  }
}

float a_traj_aux = minDistBetween2AnglesWithSign(
   orientation_robot, a_traj);

float v_limit = sqrt(1 - f_sat(0.0, 1.0, ((Dvs-d_min)/Dvs))) *
    v_max;
```

```
  velocidades[1] = f_sat(0.0, 1.0, ((PI - 4*abs(a_traj_aux))/(PI
     ))) * v_limit;

  velocidades[2] = f_sat(-1.0, 1.0, (2*a_traj_aux)/(PI)) * w_max
     ;

  // Se tiene que calcular el numero del gap selected ya que
     puede que no se corresponda con la lista de gaps original.

  for(int w = 0; w < n_gaps_aux; w++){

    if((a_gapselected_i == a_gaps_i[w]) && (a_gapselected_j ==
       a_gaps_j[w])){
      x1 = w;
    }

  }

  free(a_gaps_i);
  free(a_gaps_j);

}
```

```
// Método msg_MORSE.
// Creación del mensaje que se debe enviar al nodo MORSE.

geometry_msgs::Twist cg_algorithm::msg_MORSE(float *velocidades)
    {

  // En este punto se ha decidido guardar en el fichero la
     posición en la que se encuentra el robot.

  myfile1 << X_robot.x << " " << X_robot.y << " " << r2d(
     convert02PI(orientation_robot));
  myfile1 << "\n";

  if(distBetween2Points(X_robot, X_goal) > 0.25){

    _vel_msg.linear.x = velocidades[1];
    _vel_msg.linear.y = 0;
    _vel_msg.linear.z = 0;

    _vel_msg.angular.x = 0;
    _vel_msg.angular.y = 0;
    _vel_msg.angular.z = velocidades[2];

    key_file = true;

  }else{
```

```
    _vel_msg.linear.x = 0;
    _vel_msg.linear.y = 0;
    _vel_msg.linear.z = 0;

    _vel_msg.angular.x = 0;
    _vel_msg.angular.y = 0;
    _vel_msg.angular.z = 0;

    if(key_file == true){

      myfile1.close();
      myfile2.close();

    }

    key_file = false;

  }

  return _vel_msg;

}
```

```
// Método msg_app.
// Creación del mensaje que se debe enviar a Android.

int cg_algorithm::msg_app(char *buffer, struct info_scan *
    list_of_gaps){

  string points_xy;

  points_xy = calculate_xy(list_of_gaps);

  int num_cabecera = points_xy.length() + 2 + ConvertFtS(
      points_xy.length()).length();

  points_xy = ConvertFtS(num_cabecera) + " " + "f" + points_xy;

  strcpy(buffer, points_xy.c_str());

  return points_xy.length();
}
```

APÉNDICE C

CÓDIGO IMPLEMENTADO EN PYTHON

C.1 *entorno.py*

```
from morse.builder import *

# Land robot
atrv = ATRV()
#pr2 = BasePR2()

# Add a keyboard controller to move the robot with arrow keys
keyboard = Keyboard()
atrv.append(keyboard)

odom = Odometry()
odom.add_interface('ros', topic = "/atrv/pose")
atrv.append(odom)

atrv.motion = MotionVW()
atrv.motion.add_interface('ros', topic="/atrv/motion")
atrv.motion.properties(ControlType = 'Position')

atrv.append(atrv.motion)

#laser Hokuyo
laser = Hokuyo()
laser.properties(Visible_arc = True)
laser.properties(laser_range = 5.0)
laser.properties(resolution = 1.0)
laser.properties(scan_window = 360.0)
laser.frequency(10)
laser.add_interface('ros', topic='/base_scan')
laser.translate(0, 0, 0.5)
```

```
atrv.append(laser)


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

#Cambiar el escenario que se quiere utilizar antes de crear el
   entorno
env = Environment('/home/isaac/Documentos/Escenario_1')
```

APÉNDICE D

CÓDIGO IMPLEMENTADO EN ANDROID

D.1 *Client.java*

```
package com.example.david.tfg;

import java.io.BufferedReader;
import java.io.ByteArrayOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.PrintStream;
import java.io.UnsupportedEncodingException;
import java.net.Socket;
import java.net.InetSocketAddress;
import java.net.UnknownHostException;
import android.os.AsyncTask;
import android.util.Log;
import android.widget.TextView;

public class Client extends AsyncTask<Void, Void, String> {

String dstAddress;
int dstPort;
String response = "";
String received = "";
int n_bytes = 10000;
int bytes_recibidos;
int mode;
```

```
public Client(String addr, int port, int _mode) {
  dstAddress = addr;
  dstPort = port;
  mode = _mode;

}

@Override
protected String doInBackground(Void... arg0) {

Socket socket = null;

if(HomeActivity.i!=5 || mode == 1){ //enviar

  try {

    socket = new Socket(dstAddress, dstPort);

    PrintStream output = new PrintStream(socket.getOutputStream
        ());

    if(mode == 1){

      response = "1" + " " + OrientationVisualisationFragment.x
          + " " + OrientationVisualisationFragment.y + " " +
          OrientationVisualisationFragment.Ds + " " +
          OrientationVisualisationFragment.k + " " +
          OrientationVisualisationFragment.Dvs + " " +
          OrientationVisualisationFragment.v_max + " " +
          OrientationVisualisationFragment.w_max + " ";


    } else {

      response = "2" + " " + String.format("%.6f", HomeActivity.
          angulo) + " " + String.format("%.6f", HomeActivity.
          longitud) + " " +
      HomeActivity.mode + " " + OrientationVisualisationFragment
          .MaxLinealVel + " " +
      OrientationVisualisationFragment.MaxLinealAngle + " " +
          HomeActivity.altura + " " +
      OrientationVisualisationFragment.alturaMax + " " +
          OrientationVisualisationFragment.areaMax + " ";

    }

    output.print(response);

    output.flush();//limpia el mensaje
    output.close();

  } catch (UnknownHostException e) {
    e.printStackTrace();
```

```
    response = "UnknownHostException: " + e.toString();
  } catch (IOException e) {
    e.printStackTrace();
    response = "IOException: " + e.toString();
  } finally {
    if (socket != null) {
      try {
        socket.close();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }

} else { // recibir

  InputStream input = null;

  try {
    socket = new Socket(dstAddress, dstPort);
  } catch (IOException e) {
    e.printStackTrace();
  }


  try {
    input = socket.getInputStream();
  } catch (IOException e) {
    e.printStackTrace();
  }


  byte[] lenBytes = new byte[n_bytes];


  try {
    bytes_recibidos = input.read(lenBytes, 0, n_bytes);
  } catch (IOException e) {
    e.printStackTrace();
  }


  received = new String(lenBytes).trim();

  if (socket != null) {
    try {
      socket.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
```

```
}

return received;

}


@Override
protected void onPostExecute(String result) {
}
}
```

D.2 *CubeRenderer.java*

```
package com.example.david.tfg;

import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;
import javax.microedition.khronos.opengles.GL11;

import com.example.david.tfg.GLText.GLText;
import com.example.david.tfg.orientationProvider.
   OrientationProvider;
import com.example.david.tfg.representation.MatrixF4x4;
import com.example.david.tfg.representation.Quaternion;
import com.example.david.tfg.representation.Vector3f;
import com.example.david.tfg.representation.Vector4f;

import android.content.Context;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.opengl.GLSurfaceView;
import android.opengl.GLU;
import android.opengl.GLUtils;
import android.os.Handler;
import android.util.Log;
import java.util.concurrent.TimeUnit;

import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;

import android.content.Context;
import android.opengl.GLSurfaceView;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.TimeoutException;

/**
* Class that implements the rendering of a Point with the
   current rotation of the device that is provided by a
```

```
* OrientationProvider
*
* @author David Arévalo Jiménez
*
*/

/*
Modifier Author : Juan Isaac Cifre Izquierdo

Year : 2018
*/

public class CubeRenderer implements GLSurfaceView.Renderer {
/**
* The colour-cube that is drawn repeatedly
*/
private Circle mPoint;
private Circle mCircle0;
private Circle mCircle1;
private Circle mCircle2;
private Circle mCircle3;
private Circle mCircle4;
private Cuadrado mCuadrado;
private Lines mLines;


/**
* The current provider of the device orientation.
*/
private OrientationProvider orientationProvider = null;
private Quaternion quaternion = new Quaternion();
private MatrixF4x4 matrixRot = new MatrixF4x4();
private Vector4f vectorOrigen = new Vector4f();
private Vector4f vectorPointMoved = new Vector4f();
private Vector4f vectorDiference = new Vector4f();
private float angulo;
private float angulo2D;
private float moduloA;
private float moduloB;
private float moduloC;
private float longitud;
float x;
float y;
float z;
float w;
private int port;
private String ip;

private float naranjavistoso[]= new float[]{0.968f, 0.294f,
    0.086f, 1.0f}; //AZULFUERTE;
private float azulturquesa[]= new float[]{0.086f, 0.611f, 0.611f
    , 1.0f}; //AZULTURQUESA;
private float azul[]= new float[]{0.231f, 0.352f, 0.847f, 1.0f};
```

```
    //AZUL;
private float lila[]= new float[]{0.792f,0.266f , 0.631f, 1.0f};
    //LILA;
private float amarillo[]= new float[]{1.0f, 1.0f, 0.0f, 1.0f};
   //AMARILLO;
private float verdeLight[]= new float[]{0.565f, 0.933f, 0.565f,
   1.0f};//VERDE CLARO
private float verde[]= new float[]{0.0f, 1.0f, 0.0f, 1.0f};//
   VERDE
private float verdeDark[]= new float[]{0.000f, 0.392f, 0.000f,
   1.0f};//VERDE OSCURO
private float tomate[]= new float[]{1.0f, 0.388f, 0.278f, 1.0f};
    //TOMATE
private float naranja[]= new float[]{1.0f, 0.271f, 0.0f, 1.0f};
   //NARANJA
private float rojo[]= new float[]{1.0f, 0.0f, 0.0f, 1.0f};//ROJO

private Context context;

// For print the gaps

private Circle mPoint2;
private Circle mPoint3;
private Circle mPoint4;
private Circle mPoint5;
private Circle mPoint6;

private Lines mLines2;
private Lines mLines3;
private Lines mLines4;
private Circle mCircle5;

private Triangle triangle1;

private GLText glText;
private int width = 100; // Updated to the Current Width +
   Height in onSurfaceChanged()
private int height = 100;

//Coordenadas x e y para dibujar obstaculos
private float xp[]= new float[361];
private float yp[]= new float[361];

// Coordenadas x e y para dibujar orientacion robot
private float x_robot[]= new float[3];
private float y_robot[]= new float[3];

private float c_a_md[] = new float[6];
private float R;
private float Ds;
private float range_max;

// Coordenadas x e y de los gaps
```

```
private float x_gaps_i[]= new float[20];
private float x_gaps_j[]= new float[20];
private float y_gaps_i[]= new float[20];
private float y_gaps_j[]= new float[20];

//Contadores para poder controlar el dibujo de los gaps y los
   obstaculos
private int cont_all_points_aux = 0;
private int cont_gaps_points_i_aux = 0;
private int cont_gaps_points_j_aux = 0;
private int cont_no_gaps_i = 0;
private int cont_no_gaps_j = 0;
private int cont_no_obst = 0;

// Coordenadas x e y del punto objetivo que recibimos del nodo.
   Estan calculadas respecto al robot y no al centro del mapa
private float x_T_mod = 0;
private float y_T_mod = 0;
private int gap_selected = 0;

int cont_1 = 0; // veces correcto recibido
int cont_2 = 0; // veces incorrecto recibido

float scale(float n){

  return ((1.5f * n) / (range_max));
}

/**
* Initialises a new CubeRenderer
*/

public CubeRenderer(Context context) {
  this.context = context;
  //colors= new float[]{1.0f, 1.0f, 0.0f, 1.0f}; //AMARILLO
  mPoint = new Circle(0.05f,1,amarillo);
  //colors= new float[]{0.0f, 1.0f, 0.0f, 1.0f};//VERDE LIMA
  mCircle0 = new Circle(0.02f,0,verdeLight);
  //colors= new float[]{0.196f, 0.804f, 0.196f, 1.0f};//VERDE
     OSCURO
  mCircle1 = new Circle(0.5f,0,verde);
  //colors= new float[]{1.0f, 0.388f, 0.278f, 1.0f}; //TOMATE
  mCircle2 = new Circle(1f,0,verdeDark);
  //colors= new float[]{1.0f, 0.271f, 0.0f, 1.0f}; //NARANJA
  mCircle3 = new Circle(1.5f,0,naranja);
  //colors= new float[]{1.0f, 0.0f, 0.0f, 1.0f};//ROJO
  mCircle4 = new Circle(2f,0,rojo);

  mLines = new Lines();

  mCuadrado = new Cuadrado();

  mPoint2 = new Circle(0.009f,1,rojo);
```

```
  mPoint4 = new Circle(0.05f,1, amarillo);
}

/**
* Sets the orientationProvider of this renderer. Use this method
    to change which sensor fusion should be currently
* used for rendering the cube. Simply exchange it with another
   orientationProvider and the cube will be rendered
* with another approach.
*
* @param orientationProvider The new orientation provider that
   delivers the current orientation of the device
*/
public void setOrientationProvider(OrientationProvider
   orientationProvider) {
  this.orientationProvider = orientationProvider;
}

@Override
public void onSurfaceCreated(GL10 gl, EGLConfig config) {
  // dither is enabled by default, we don't need it
  gl.glDisable(GL10.GL_DITHER);

  mCuadrado.loadGLTexture(gl, this.context);

  // clear screen in black
  gl.glClearColor(0, 0, 0, 1);
  // Configura el buffer de profundidad
  gl.glClearDepthf(1.0f);
  // Modo de renderizado de la profundidad
  gl.glEnable(GL10.GL_DEPTH_TEST);
  // Función de comparación de la profundidad
  gl.glDepthFunc(GL10.GL_LEQUAL);
  // Cómo se calcula la perspectiva
  gl.glHint(GL10.GL_PERSPECTIVE_CORRECTION_HINT, GL10.GL_NICEST)
     ;

  port=OrientationVisualisationFragment.port;
  ip=OrientationVisualisationFragment.ip;

}


//Perform the actual rendering of the cube for each frame

// @param  The surface on which the cube should be rendered


public void onDrawFrame(GL10 gl) {
// clear screen

gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
```

```
// set-up modelview matrix
gl.glMatrixMode(GL10.GL_MODELVIEW);
gl.glLoadIdentity();
gl.glTranslatef(0.0f, 0.0f, -3.0f);//coloca el cubo a una
   distancia 3 de la pantalla

if (HomeActivity.i != 5) {

  if (HomeActivity.i != 0) {
    mCircle0.draw(gl);
    mCircle1.draw(gl);
    mCircle2.draw(gl);
    mCircle3.draw(gl);
    mCircle4.draw(gl);
    mLines.draw(gl);
  }
  if (orientationProvider != null) {


    // Get the rotation from the current orientationProvider as
       quaternion
    orientationProvider.getQuaternion(quaternion);
    orientationProvider.getRotationMatrix(matrixRot);

    vectorOrigen.setXYZW(0, 0, -2.5f, 1);
    vectorPointMoved.setXYZW(0, 0, -2.5f, 1);

    matrixRot.multiplyVector4fByMatrix(vectorPointMoved);

    if (!screenTouched) {

      x = vectorPointMoved.getX();
      y = vectorPointMoved.getY();
      z = vectorPointMoved.getZ();
      w = vectorPointMoved.getW();

      moduloA = vectorPointMoved.modulo(); //nuevo punto
      moduloB = vectorOrigen.modulo();//origen
      vectorPointMoved.subtract(vectorOrigen, vectorDiference);
      moduloC = vectorDiference.modulo(); //distancia entre
         ambos puntos

      if (!HomeActivity.horizontalRotation && vectorPointMoved.
         getX() < -0.5 && vectorPointMoved.getY() < 0.5 &&
      vectorPointMoved.getY() > -0.5 && vectorPointMoved.getZ()
         < 0) {
        Connect.setMyBoolean(true);
      }

      Log.v("CoordenadasXYZ", "x=" + vectorPointMoved.getX() + "
          y=" + vectorPointMoved.getY() + "z=" +
         vectorPointMoved.getZ() + "w=" + vectorPointMoved.getW
         () + '\n');
```

```
    if (HomeActivity.horizontalRotation) {

      if (HomeActivity.i == 0) {
        gl.glRotatef((float) (-2.0f * Math.acos(quaternion.
            getW()) * 180.0f / Math.PI), quaternion.getX(),
            quaternion.getY(), -quaternion.getZ());
      }

      if (HomeActivity.i == 1 || HomeActivity.i == 2 ||
          HomeActivity.i == 3) {
        if (HomeActivity.i == 1 || HomeActivity.i == 2) {
          angulo = (float) Math.acos((moduloA * moduloA +
              moduloB * moduloB - moduloC * moduloC) / (2 *
              moduloA * moduloB)); //teorema del coseno
          longitud = vectorOrigen.modulo() * angulo; //arco
          longitud = OrientationVisualisationFragment.
              sensibility * longitud;
          angulo = (float) Math.atan2(vectorPointMoved.getY(),
              vectorPointMoved.getX());

        } else {
          angulo = (float) (-SensorSelectionActivity.
              angleTouch + Math.PI);
          longitud = (float) SensorSelectionActivity.
              powerTouch * 2 / 100;
        }

        Log.v("Touch", "angle: " + angulo + " " + "power: " +
            longitud);

        if (longitud > 2) {
          longitud = 2;
        }

        gl.glTranslatef((float) (longitud * Math.cos(angulo)),
            (float) (longitud * Math.sin(angulo)), 0.0f); //
            rota a 2 del centro

      }
      //Adjust angle to server
      if (HomeActivity.i == 1 || HomeActivity.i == 2) {

        //cambiar valores de angulos de (-3.14 - 0 - 3.14) a
            (0 - 6.28) y girar
        angulo = (float) (angulo + Math.PI * 2.5);
        if (angulo > Math.PI * 2) {
          angulo = (float) (angulo - Math.PI * 2);
        }
        Log.v("CoordenadasA", "yaw= " + angulo + '\n');

      } else { //joystick
        //girar sistema 90Â° en sentido horario
```

```
        angulo = (float) (angulo + Math.PI * 0.5);
        if (angulo > Math.PI * 2) {
          angulo = (float) (angulo - Math.PI * 2);
        }
        Log.v("CoordenadasA", "yaw= " + angulo + '\n');
      }
      Log.v("messages", "angulo= " + String.format("%.6f",
          angulo/*180/Math.PI*/) + "long= " + String.format("
          %.6f", longitud) + '\n'); //angulos de 0 a 6.28
      if (longitud == 0) {
        angulo = 0;
      }
    }
  } else if (HomeActivity.horizontalRotation) {

    OrientationVisualisationFragment.vectorNewOrigen.setXYZW(x
        , y, z, w);

    moduloA = vectorPointMoved.modulo(); //nuevo punto
    moduloB = OrientationVisualisationFragment.vectorNewOrigen
        .modulo();//origen
    vectorPointMoved.subtract(OrientationVisualisationFragment
        .vectorNewOrigen, vectorDiference);
    moduloC = vectorDiference.modulo(); //distancia entre
        ambos puntos

    Log.v("Coordenadas", "x=" + vectorOrigen.getX() + " y=" +
        vectorOrigen.getY() + "z=" + vectorOrigen.getZ() + "w="
         + vectorOrigen.getW() + '\n');

    if (HomeActivity.i == 0) {
      gl.glRotatef((float) (-2.0f * Math.acos(quaternion.getW
          ()) * 180.0f / Math.PI), quaternion.getX(),
          quaternion.getY(), -quaternion.getZ());
    }
    if (HomeActivity.i == 1 || HomeActivity.i == 2) {
      angulo = (float) Math.acos((moduloA * moduloA + moduloB
          * moduloB - moduloC * moduloC) / (2 * moduloA *
          moduloB)); //teorema del coseno
      longitud = vectorOrigen.modulo() * angulo;//arco
      longitud = OrientationVisualisationFragment.sensibility
          * longitud;

      angulo2D = (float) Math.atan2((vectorDiference.getY()),
          (vectorDiference.getX()));
      if (angulo2D >= Math.PI) {
        vectorPointMoved.setZ(-vectorPointMoved.getZ());
      }
      if (longitud > 2) {
        longitud = 2;
        gl.glTranslatef((float) (longitud * Math.cos(angulo2D)
            ), (float) (longitud * Math.sin(angulo2D)), 0.0f);
            //rota a 1 del centro
```

```
        } else {
          gl.glTranslatef((float) (longitud * Math.cos(angulo2D)
              ), (float) (longitud * Math.sin(angulo2D)), 0.0f);
        }
      }
      if (HomeActivity.i == 1 || HomeActivity.i == 2) {

        //cambiar valores de angulos de (-3.14 - 0 - 3.14) a (0
            - 6.28)
        angulo2D = (float) (angulo2D + Math.PI * 2.5);

        if (angulo2D > Math.PI * 2) {
          angulo2D = (float) (angulo2D - Math.PI * 2);
        }

      } else {
        //girar sistema 90Â° en sentido horario
        angulo2D = (float) (angulo2D + Math.PI * 0.5);
        if (angulo2D > Math.PI * 2) {
          angulo2D = (float) (angulo2D - Math.PI * 2);
        }
      }
      Log.v("messages2D", "angulo2D= " + String.format("%.6f",
          angulo2D/*180/Math.PI*/) + "long= " + String.format("
          %.6f", longitud) + '\n'); //angulo2Ds de 0 a 6.28
      if (longitud == 0) {
        angulo2D = 0;
      }
      angulo = angulo2D;
    }
  }


  // draw our object
  gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
  gl.glEnableClientState(GL10.GL_COLOR_ARRAY);

  if ((HomeActivity.i == 0)) {
    mCuadrado.draw(gl);
  } else {
    mPoint.draw(gl);
  }
  HomeActivity.angulo = angulo;//global variables to avoid
      client problems
  HomeActivity.longitud = longitud;

  if (HomeActivity.i == 1 || HomeActivity.i == 3) {
    Client myClient = new Client(ip, port, 2);//192.168.1.X 8080
    myClient.execute();
  }
}

//****************** 2018 ********************//
```

```
// Closest Gap Mode

if (HomeActivity.i == 5) {

  // Guardamos las coordenadas del punto objetivo y las enviamos

  Client myClient2 = new Client(ip, port, 1);

  myClient2.execute();

  // Esperamos a que acabe de enviarlo

  try {
    myClient2.get();
  } catch (InterruptedException e) {
    e.printStackTrace();
  } catch (ExecutionException e) {
    e.printStackTrace();
  }

  //Creamos nuevo cliente para recibir el mensaje del nodo

  Client myClient = new Client(ip, port, 2);


  myClient.execute();

  char c[] = new char[0];

  try {
    c = myClient.get().toCharArray();
  } catch (InterruptedException e) {
    e.printStackTrace();
  } catch (ExecutionException e) {
    e.printStackTrace();
  }

  int i = 0;
  int x = 0;
  int n_points = 0;
  int cont_f = 0;

  int cont_all_points = 0;
  int cont_gaps_points_i = 0;
  int cont_gaps_points_j = 0;

  int _bytes_recibidos;

  boolean bad_message = false;

  String w = "";
  char[] c_aux = new char[20];
```

```
while (c[i] != 'F') {

  while (c[i] != 'f') {

    if (c[i] != ' ') {

      c_aux[x] = c[i];

      x++;

    } else {

      for (int j = 0; j < x; j++) {
        w = w + c_aux[j];
      }

      switch (cont_f) {
        case 0:
        _bytes_recibidos = Integer.parseInt(w);

        if(_bytes_recibidos != myClient.bytes_recibidos){
          bad_message = true;
        }

        break;

        case 1:
        x_robot[n_points] = Float.parseFloat(w);
        break;
        case 2:
        xp[n_points] = Float.parseFloat(w);
        cont_all_points++;
        cont_all_points_aux = cont_all_points;

        break;
        case 3:
        x_gaps_i[n_points] = Float.parseFloat(w);
        cont_gaps_points_i++;
        cont_gaps_points_i_aux = cont_gaps_points_i;
        break;
        case 4:
        x_gaps_j[n_points] = Float.parseFloat(w);
        cont_gaps_points_j++;
        cont_gaps_points_j_aux = cont_gaps_points_j;
        break;

        case 5:
        x_T_mod = Float.parseFloat(w);
        break;
        case 6:
        y_robot[n_points] = Float.parseFloat(w);
        break;
        case 7:
```

```
        yp[n_points] = Float.parseFloat(w);
        break;
        case 8:
        y_gaps_i[n_points] = Float.parseFloat(w);
        break;
        case 9:
        y_gaps_j[n_points] = Float.parseFloat(w);
        break;
        case 10:
        y_T_mod = Float.parseFloat(w);
        break;
        case 11:
        gap_selected = Integer.parseInt(w);
        break;
        case 12:
        c_a_md[n_points] = Float.parseFloat(w);
        break;
        case 13:
        R = Float.parseFloat(w);
        break;
        case 14:
        Ds = Float.parseFloat(w);
        break;
        case 15:
        range_max = Float.parseFloat(w);
        break;
      }

      n_points++;
      x = 0;
      w = "";
    }

    i++;

    if(bad_message == true){
      break;
    }


  }

  if(bad_message == true){
    break;
  }

  if (c[i] == 'f') {
    cont_f++;
  }

  i++;
  x = 0;
  n_points = 0;
```

```
    w = "";

  }

  if(bad_message == true){

    cont_1++;

  }else{

    cont_2++;

  }


  if (cont_all_points == 0) {
    cont_no_obst++;
    if (cont_no_obst == 100) {
      cont_all_points_aux = 0;
    }
  } else {
    cont_no_obst = 0;
  }

  if (cont_gaps_points_i == 0) {
    cont_no_gaps_i++;
    if (cont_no_gaps_i == 100) {
      cont_gaps_points_i_aux = 0;
    }
  } else {
    cont_no_gaps_i = 0;
  }


  //Pintamos los gaps

  for (int z = 0; z < cont_gaps_points_i_aux; z++) {

    if (z == gap_selected) {
      triangle1 = new Triangle(scale(x_gaps_i[z]), scale(
          y_gaps_i[z]), scale(x_gaps_j[z]), scale(y_gaps_j[z]),
          0.0f, 0.0f);

      gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);

      triangle1.draw(gl, verde);

      gl.glDisableClientState(GL10.GL_VERTEX_ARRAY);
    } else {

      triangle1 = new Triangle(scale(x_gaps_i[z]), scale(
          y_gaps_i[z]), scale(x_gaps_j[z]), scale(y_gaps_j[z]),
          0.0f, 0.0f);
```

```
    gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);

    triangle1.draw(gl, azul);

    gl.glDisableClientState(GL10.GL_VERTEX_ARRAY);

  }
}

//Pintamos todos los puntos de los obstaculos detectados
for (int z = 0; z < cont_all_points_aux; z++) {

  gl.glTranslatef(scale(xp[z]), scale(yp[z]), 0);
  mPoint2.draw(gl);
  gl.glTranslatef(-scale(xp[z]), -scale(yp[z]), 0);

}

mPoint3 = new Circle(scale(R), 0, rojo);
mPoint5 = new Circle(scale(Ds), 0, naranjavistoso);

// Pintamos el circulo que representa el robot
gl.glTranslatef(0.0f, 0.0f, 0.0f);
mPoint5.draw(gl);
gl.glTranslatef(0.0f, 0.0f, 0.0f);
mPoint3.draw(gl);

// Imprimir direccion del robot como una flecha
mLines2 = new Lines(scale(x_robot[0]), scale(y_robot[0]), 0,
   0, 0, 0, 0, 0); // tronco de la flecha
mLines2.draw(gl, 1.0f, 1.0f, 0.0f, 1.0f);

//triangulo de la punta de la flecha
triangle1 = new Triangle(scale(x_robot[0]), scale(y_robot[0]),
    scale(x_robot[1]) * 0.75f, scale(y_robot[1]) * 0.75f,
   scale(x_robot[2]) * 0.75f, scale(y_robot[2]) * 0.75f);

//Pintamos triangulo. Siempre que pintemos triangulos se tiene
    que activar y desactivar glEnableClientState
gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
triangle1.draw(gl, amarillo);
gl.glDisableClientState(GL10.GL_VERTEX_ARRAY);

// Imprimir las flechas que indican las coordenadas
mLines3 = new Lines(0.0f, 10.0f, 0.0f, -10.0f, 2.0f, 0.0f,
   -2.0f, 0.0f);
mLines3.draw(gl, 0.086f, 0.611f, 0.611f, 1.0f);

// Imprimir el rango de vision del robot como un circulo
mCircle5 = new Circle(scale(range_max), 0, azulturquesa);

mCircle5.draw(gl);
```

```
  // Imprimir direccion del robot como una flecha
  mLines4 = new Lines(scale(c_a_md[0]), scale(c_a_md[1]), 0, 0,
      0, 0, 0, 0); // tronco de la flecha
  mLines4.draw(gl, 0.792f,0.266f , 0.631f, 1.0f);

  //triangulo de la punta de la flecha
  triangle1 = new Triangle(scale(c_a_md[0]), scale(c_a_md[1]),
      scale(c_a_md[2]) * 0.80f, scale(c_a_md[3]) * 0.80f, scale(
      c_a_md[4]) * 0.80f, scale(c_a_md[5]) * 0.80f);

  //Pintamos triangulo. Siempre que pintemos triangulos se tiene
       que activar y desactivar glEnableClientState
  gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
  triangle1.draw(gl, lila);
  gl.glDisableClientState(GL10.GL_VERTEX_ARRAY);

  // Escalamos y pintamos el punto objetivo. Siempre se pinta en
       funcion de la posicion del robot
  gl.glTranslatef(scale(x_T_mod), scale(y_T_mod), 0);
  mPoint4.draw(gl);
  gl.glTranslatef(-scale(x_T_mod), -scale(y_T_mod), 0);
}
}


/**
* Update view-port with the new surface
*
* @param gl the surface
* @param width new width
* @param height new height
*/
public void onSurfaceChanged(GL10 gl, int width, int height) {



  if(height == 0) {             //Prevent A Divide By Zero By
    height = 1;             //Making Height Equal One
  }

  gl.glViewport(0, 0, width, height);   //Reset The Current
      Viewport
  gl.glMatrixMode(GL10.GL_PROJECTION);  //Select The Projection
      Matrix
  gl.glLoadIdentity();          //Reset The Projection Matrix

  //Calculate The Aspect Ratio Of The Window
  GLU.gluPerspective(gl, 90.0f, (float)width / (float)height,
      0.1f, 100.0f);

  gl.glMatrixMode(GL10.GL_MODELVIEW);   //Select The Modelview
      Matrix
```

```
  gl.glLoadIdentity();          //Reset The Modelview Matrix
}

/**
* Flag indicating whether you want to view inside out, or
    outside in
*/
private boolean screenTouched = false;

/**
* Toggles whether the cube will be shown inside-out or outside
    in.
*/
public void togglescreenTouched() {
  this.screenTouched = !screenTouched;
}

}
```

D.3 *Circle.java*

```
package com.example.david.tfg;

import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.nio.FloatBuffer;
import javax.microedition.khronos.opengles.GL10;

public class Circle {

  // Circle variables
  int circlePoints = 40;
  float radius = 1.5f;
  float center_x = 0.0f;
  float center_y = 0.0f;
  int types;
  float colors[];

  // Outer vertices of the circle i.e. excluding the center_x,
     center_y
  int circumferencePoints = circlePoints-1;

  // Circle vertices and buffer variables
  int vertices = 0;
  float circleVertices[] = new float[circlePoints*2];
  private FloatBuffer toiletBuff; // 4 bytes per float

  // Color values
  private float rgbaValues[] = {
    1f,     0f,  0f,     0f,
```

```
    0f,  1f,  0f,  0f,
    0f,     0f,  1f,     0f,
  };

  private FloatBuffer colorBuff;

  public Circle(float radio, int type, float[] color)
  {
    radius=radio;
    types=type;
    colors=color;
    // The initial buffer values
    circleVertices[vertices++] = center_x;
    circleVertices[vertices++] = center_y;

    // Set circle vertices values
    for (int i = 0; i < circumferencePoints; i++)
    {
      float percent = (i / (float) (circumferencePoints - 1));
      float radians = (float) (percent * 2 * Math.PI);

      // Vertex position
      float outer_x = (float) (center_x + radius * Math.cos(
          radians));
      float outer_y = (float) (center_y + radius * Math.sin(
          radians));

      circleVertices[vertices++] = outer_x;
      circleVertices[vertices++] = outer_y;
    }

    // Float buffer short has four bytes
    ByteBuffer toiletByteBuff = ByteBuffer
    .allocateDirect(circleVertices.length * 4);

    // Garbage collector won't throw this away
    toiletByteBuff.order(ByteOrder.nativeOrder());
    toiletBuff = toiletByteBuff.asFloatBuffer();
    toiletBuff.put(circleVertices);
    toiletBuff.position(0);
  }

  // Draw methods
  public void draw(GL10 gl) {

    gl.glDisable(GL10.GL_TEXTURE_2D);
    gl.glEnableClientState(GL10.GL_COLOR_ARRAY);
    // Establecemos el color del triángulo en modo RGBA
    gl.glColor4f(colors[0], colors[1], colors[2], colors[3]);

    // Dibujamos al revés que las agujas del reloj
    gl.glFrontFace(GL10.GL_CCW);
    // Indicamos el nÂº de coordenadas (3), el tipo de datos de
```

```
        la
    // matriz (float), la separación en la matriz de los vé
       rtices
    // (0) y el buffer con los vértices
    gl.glVertexPointer(2, GL10.GL_FLOAT, 0, toiletBuff);
    // Indicamos al motor OpenGL que le hemos pasado una matriz
       de
    // vértices
    gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
    gl.glDisableClientState(GL10.GL_COLOR_ARRAY);

    // Dibujamos la superficie
    if(types==0){
      // Draw hollow circle
      gl.glDrawArrays(GL10.GL_LINE_LOOP, 1, circumferencePoints)
         ;
    } else {
      // Draw circle as filled shape
      gl.glDrawArrays(GL10.GL_TRIANGLE_FAN, 1,
         circumferencePoints);// 1 para no empezar en el centro
    }

    // Desactivamos el buffer de los vértices
    gl.glDisableClientState(GL10.GL_VERTEX_ARRAY);
    gl.glDisableClientState(GL10.GL_COLOR_ARRAY); //super
       importante
  }
}
```

D.4 *Lines.java*

```
package com.example.david.tfg;

import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.nio.FloatBuffer;

import javax.microedition.khronos.opengles.GL10;

import static javax.microedition.khronos.opengles.GL10.
   GL_COLOR_BUFFER_BIT;
import static javax.microedition.khronos.opengles.GL10.
   GL_DEPTH_BUFFER_BIT;

public class Lines {
  // Indica si deseamos colorear el triángulo
  private boolean conColor;
  // Buffer de tipo float que usamos para pasar
  // a la librería OpenGL los vértices del triángulo
  private FloatBuffer bufferVertices;
```

```
// Vértices del triángulo
private float vertices[] =
{
  0f, 2, 0.0f, // Arriba
  0.0f, -2, 0.0f, // Abajo
  -2, 0.0f, 0.0f, // Abajo izquierda
  2, 0.0f, 0.0f, // Abajo derecha
};


// Constructor del triángulo de David el cual emplea los
   vertices definidos arriba
public Lines() {
  // Definimos el buffer con los vértices del polígono.
  // Un número float tiene 4 bytes de longitud, así que
  // multiplicaremos x 4 el número de vértices.
  ByteBuffer byteBuf=ByteBuffer.allocateDirect(vertices.length
     *4);
  // Establecemos el orden de los bytes en el buffer con el
     valor
  // nativo (es algo así como indicar cómo se leen los bytes
     de
  // izq a dcha o al revés).
  byteBuf.order(ByteOrder.nativeOrder());
  // Asignamos el nuevo buffer al buffer de esta clase
  bufferVertices = byteBuf.asFloatBuffer();
  // Introducimos los vértices en el buffer
  bufferVertices.put(vertices);
  // Movemos la posición del buffer al inicio
  bufferVertices.position(0);
  // Guardamos si es necesario colorear el polígono
}

// Constructor para dibujar hasta 2 lineas. Hay que pasarle
   coordenada x e y de cada extremo de la linea
public Lines(float x1, float y1, float x2, float y2, float x3,
    float y3, float x4, float y4) {

  float vertices2[] =
  {
    x1, y1, 0.0f, // L1
    x2, y2, 0.0f, // L1
    x3, y3, 0.0f, // L2
    x4, y4, 0.0f, // L2
  };

  // Definimos el buffer con los vértices del polígono.
  // Un número float tiene 4 bytes de longitud, así que
  // multiplicaremos x 4 el número de vértices.
  ByteBuffer byteBuf=ByteBuffer.allocateDirect(vertices.length
     *4);
  // Establecemos el orden de los bytes en el buffer con el
     valor
```

```
    // nativo (es algo así como indicar cómo se leen los bytes
       de
    // izq a dcha o al revés).
    byteBuf.order(ByteOrder.nativeOrder());
    // Asignamos el nuevo buffer al buffer de esta clase
    bufferVertices = byteBuf.asFloatBuffer();
    // Introducimos los vértices en el buffer
    bufferVertices.put(vertices2);
    // Movemos la posición del buffer al inicio
    bufferVertices.position(0);
    // Guardamos si es necesario colorear el polígono
  }



  // Método que invoca el Renderer cuando debe dibujar el triá
     ngulo
  public void draw(GL10 gl) {
    // Dibujamos al revés que las agujas del reloj

    gl.glColor4f(0.5f, 0.5f, 1.0f, 1.0f);
    gl.glFrontFace(GL10.GL_CCW);
    // Indicamos el nÂº de coordenadas (3), el tipo de datos de
       la
    // matriz (float), la separación en la matriz de los vé
       rtices
    // (0) y el buffer con los vértices
    gl.glVertexPointer(3, GL10.GL_FLOAT, 0, bufferVertices);
    // Indicamos al motor OpenGL que le hemos pasado una matriz
       de
    // vértices
    gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);

    // Dibujamos la superficie mediante la matriz en el modo
    // triángulo
    gl.glDrawArrays(GL10.GL_LINES, 0, vertices.length/3 );
    // Desactivamos el buffer de los vértices
    gl.glDisableClientState(GL10.GL_VERTEX_ARRAY);

    gl.glDisableClientState(GL10.GL_COLOR_ARRAY); //super
       importante
  }

  public void draw(GL10 gl, float red, float green, float blue,
     float alpha) {
    // Dibujamos al revés que las agujas del reloj

    gl.glColor4f(red, green, blue, alpha);
    gl.glFrontFace(GL10.GL_CCW);
    // Indicamos el nÂº de coordenadas (3), el tipo de datos de
       la
    // matriz (float), la separación en la matriz de los vé
       rtices
```

```
    // (0) y el buffer con los vértices
    gl.glVertexPointer(3, GL10.GL_FLOAT, 0, bufferVertices);
    // Indicamos al motor OpenGL que le hemos pasado una matriz
       de
    // vértices
    gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);

    // Dibujamos la superficie mediante la matriz en el modo
    // triángulo
    gl.glDrawArrays(GL10.GL_LINES, 0, vertices.length/3 );
    // Desactivamos el buffer de los vértices
    gl.glDisableClientState(GL10.GL_VERTEX_ARRAY);

    gl.glDisableClientState(GL10.GL_COLOR_ARRAY); //super
       importante
  }
} // end clase
```

D.5 *Triangle.java*

```
package com.example.david.tfg;


/**
* Created by isaac on 28/03/18.
*/

import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.nio.FloatBuffer;

import javax.microedition.khronos.opengles.GL10;
import javax.microedition.khronos.opengles.GL11;

public class Triangle {
  private FloatBuffer mFVertexBuffer;
  private ByteBuffer mIndexBuffer;

  public Triangle(float x1, float y1, float x2, float y2, float
     x3, float y3) {

    float vertices[] = {
      x1, y1, 0.0f,
      x2, y2, 0.0f,
      x3, y3, 0.0f
    };

    byte indices[] = { 0, 1, 2 };

    mFVertexBuffer = makeFloatBuffer(vertices);
```

```
    mIndexBuffer = ByteBuffer.allocateDirect(indices.length);
    mIndexBuffer.put(indices);
    mIndexBuffer.position(0);
  }

  public void draw(GL10 gl, float[] color) {
    gl.glColor4f(color[0], color[1], color[2], color[3]);
    gl.glVertexPointer(3, GL11.GL_FLOAT, 0, mFVertexBuffer);
    gl.glDrawElements(GL11.GL_TRIANGLES, 3, GL11.
        GL_UNSIGNED_BYTE, mIndexBuffer);
  }

  private static FloatBuffer makeFloatBuffer(float[] arr) {
    ByteBuffer bb = ByteBuffer.allocateDirect(arr.length * 4);
    bb.order(ByteOrder.nativeOrder());
    FloatBuffer fb = bb.asFloatBuffer();
    fb.put(arr);
    fb.position(0);
    return fb;
  }
}
```

D.6 *OrientationVisualizationFragment.java*

```
package com.example.david.tfg;

import com.example.david.tfg.orientationProvider.
   AccelerometerCompassProvider;
import com.example.david.tfg.orientationProvider.
   CalibratedGyroscopeProvider;
import com.example.david.tfg.orientationProvider.
   GravityCompassProvider;
import com.example.david.tfg.orientationProvider.
   ImprovedOrientationSensor1Provider;
import com.example.david.tfg.orientationProvider.
   ImprovedOrientationSensor2Provider;
import com.example.david.tfg.orientationProvider.
   OrientationProvider;
import com.example.david.tfg.orientationProvider.
   RotationVectorProvider;
import com.example.david.tfg.representation.MatrixF4x4;
import com.example.david.tfg.representation.Quaternion;
import com.example.david.tfg.representation.Vector4f;

import android.content.Context;
import android.content.SharedPreferences;
import android.content.res.Resources;
import android.graphics.Bitmap;
```

```
import android.graphics.BitmapFactory;
import android.graphics.drawable.Drawable;
import android.hardware.SensorManager;
import android.opengl.GLSurfaceView;
import android.os.Bundle;
import android.preference.PreferenceManager;
import android.support.v4.app.Fragment;
import android.util.Log;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.view.View.OnLongClickListener;

import com.example.david.tfg.R;

/**
* A fragment that contains the same visualisation for different
   orientation providers
*/
public class OrientationVisualisationFragment extends Fragment {
  /**
  * The surface that will be drawn upon
  */
  private GLSurfaceView mGLSurfaceView;
  /**
  * The class that renders the cube
  */
  private CubeRenderer mRenderer;
  /**
  * The current orientation provider that delivers device
     orientation.
  */
  private OrientationProvider currentOrientationProvider;

  /**
  * The fragment argument representing the section number for
     this
  * fragment.
  */
  public static String ARG_SECTION_NUMBER = "section_number";
  public static Bitmap b ;
  public static Vector4f vectorNewOrigen = new Vector4f();

  static public int SensorSelected=1;
  static public float sensibility=0.75f;
  static public String ip;
  static public int port;
  static public int MaxLinealVel,MaxLinealAngle,alturaMax,
     areaMax,language;

  static public String _x;
  static public String _y;
```

```
static public String _Ds;
static public String _k;
static public String _Dvs;
static public String _v_max;
static public String _w_max;

static public float Ds;
static public float k;
static public float Dvs;
static public float x;
static public float y;
static public float v_max;
static public float w_max;



@Override
public void onResume() {
  // Ideally a game should implement onResume() and onPause()
  // to take appropriate action when the activity looses focus
  super.onResume();
  currentOrientationProvider.start();
  mGLSurfaceView.onResume();
}

@Override
public void onPause() {
  // Ideally a game should implement onResume() and onPause()
  // to take appropriate action when the activity looses focus
  super.onPause();
  currentOrientationProvider.stop();
  mGLSurfaceView.onPause();
}

@Override
public View onCreateView(LayoutInflater inflater, ViewGroup
    container, Bundle savedInstanceState) {
  language=HomeActivity.language;
  // Initialise the orientationProvider
  vectorNewOrigen.setXYZW(0,0,-2.5f,1);

  //Menu Settings
  PreferenceManager.setDefaultValues(getActivity(), R.xml.
      settings, false);
  SharedPreferences sharedPreferences = PreferenceManager.
      getDefaultSharedPreferences(getActivity());
  HomeActivity.compass = sharedPreferences.getBoolean("compass
      ", true);

  SharedPreferences sharedPreferencesIP = PreferenceManager.
      getDefaultSharedPreferences(getActivity());
  ip = sharedPreferencesIP.getString("ip", "10.0.2.15");
```

```
SharedPreferences sharedPreferencesPort = PreferenceManager.
    getDefaultSharedPreferences(getActivity());
port = Integer.parseInt(sharedPreferencesPort.getString("
    port", "8080"));

SharedPreferences sharedPreferencesMaxLinealVel =
    PreferenceManager.getDefaultSharedPreferences(getActivity
    ());
MaxLinealVel = Integer.parseInt(
    sharedPreferencesMaxLinealVel.getString("maxVel", "50"));

SharedPreferences sharedPreferencesMaxLinealAngle =
    PreferenceManager.getDefaultSharedPreferences(getActivity
    ());
MaxLinealAngle = Integer.parseInt(
    sharedPreferencesMaxLinealAngle.getString("maxAngle", "50
    "));

SharedPreferences sharedPreferencesMaxAltura =
    PreferenceManager.getDefaultSharedPreferences(getActivity
    ());
alturaMax = Integer.parseInt(sharedPreferencesMaxAltura.
    getString("maxAltura", "20"));

SharedPreferences sharedPreferencesMaxArea =
    PreferenceManager.getDefaultSharedPreferences(getActivity
    ());
areaMax = Integer.parseInt(sharedPreferencesMaxArea.
    getString("maxArea", "7"));

SharedPreferences sharedPreferencesLanguage =
    PreferenceManager.getDefaultSharedPreferences(getActivity
    ());
language = Integer.parseInt(sharedPreferencesLanguage.
    getString("language", "1"));

SharedPreferences sharedPreferencesX = PreferenceManager.
    getDefaultSharedPreferences(getActivity());
//x = Float.parseFloat(sharedPreferencesX.getString("x",
    "0.0f"));

_x = sharedPreferencesX.getString("x", "0.0");


SharedPreferences sharedPreferencesY = PreferenceManager.
    getDefaultSharedPreferences(getActivity());

_y = sharedPreferencesY.getString("y", "0.0");

_Ds = sharedPreferencesY.getString("Ds", "0.0");
_k = sharedPreferencesY.getString("k", "0.0");
_Dvs = sharedPreferencesY.getString("Dvs", "0.0");
_v_max = sharedPreferencesY.getString("v_max", "0.0");
```

```
_w_max = sharedPreferencesY.getString("w_max", "0.0");

//Sensor Selection
switch (getArguments().getInt(ARG_SECTION_NUMBER)) {
  case 1:
  currentOrientationProvider = new
     ImprovedOrientationSensor1Provider((SensorManager)
     getActivity()
  .getSystemService(SensorSelectionActivity.SENSOR_SERVICE))
     ;

  break;
  case 2:
  currentOrientationProvider = new
     ImprovedOrientationSensor2Provider((SensorManager)
     getActivity()
  .getSystemService(SensorSelectionActivity.SENSOR_SERVICE))
     ;
  break;
  case 3:
  currentOrientationProvider = new RotationVectorProvider((
     SensorManager) getActivity().getSystemService(
  SensorSelectionActivity.SENSOR_SERVICE));
  break;
  case 4:
  currentOrientationProvider = new
     CalibratedGyroscopeProvider((SensorManager) getActivity
     ()
  .getSystemService(SensorSelectionActivity.SENSOR_SERVICE))
     ;
  break;
  case 5:
  currentOrientationProvider = new GravityCompassProvider((
     SensorManager) getActivity().getSystemService(
  SensorSelectionActivity.SENSOR_SERVICE));
  break;
  case 6:
  currentOrientationProvider = new
     AccelerometerCompassProvider((SensorManager)
     getActivity()
  .getSystemService(SensorSelectionActivity.SENSOR_SERVICE))
     ;
  break;
  default:
  break;
}

// Create our Preview view and set it as the content of our
   Activity
mRenderer = new CubeRenderer(getContext());
mRenderer.setOrientationProvider(currentOrientationProvider)
   ;
```

```
    mGLSurfaceView = new GLSurfaceView(getActivity());
    mGLSurfaceView.setEGLConfigChooser(8, 8, 8, 8, 16, 0);
    mGLSurfaceView.setRenderer(mRenderer);

    mGLSurfaceView.setOnLongClickListener(new
       OnLongClickListener() {

      @Override
      public boolean onLongClick(View v) {
        if(HomeActivity.i!=3){
          mRenderer.togglescreenTouched();
        }
        return true;
      }
    });
    return mGLSurfaceView;
  }
}
```

D.7 *SensorSelectionActivity.java*

```
package com.example.david.tfg;

import java.util.Locale;

import android.app.AlertDialog;
import android.content.Context;
import android.content.DialogInterface;
import android.content.Intent;
import android.content.res.Configuration;
import android.graphics.Color;
import android.graphics.drawable.Drawable;
import android.hardware.SensorManager;
import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.support.v4.app.FragmentActivity;
import android.support.v4.app.FragmentManager;
import android.support.v4.app.FragmentPagerAdapter;
import android.support.v4.view.ViewPager;
import android.text.InputType;
import android.util.Log;
import android.view.Display;
import android.view.Gravity;
import android.view.KeyEvent;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.view.Surface;
import android.view.View;
import android.view.WindowManager;
```

```
import android.widget.ImageView;
import android.widget.LinearLayout;
import android.widget.RelativeLayout;
import android.widget.Toast;

import com.afollestad.materialdialogs.MaterialDialog;
import com.crystal.crystalrangeseekbar.interfaces.
    OnSeekbarChangeListener;
import com.erz.joysticklibrary.*;
import com.erz.joysticklibrary.JoyStick;

/**
* The main activity where the user can select which sensor-
    fusion he wants to try out
*
* @author David Arévalo Jiménez
*
*/
public class SensorSelectionActivity extends FragmentActivity
    implements com.erz.joysticklibrary.JoyStick.JoyStickListener{

  /**
  * The {@link android.support.v4.view.PagerAdapter} that will
      provide
  * fragments for each of the sections. We use a {@link android.
      support.v4.app.FragmentPagerAdapter} derivative,
  * which
  * will keep every loaded fragment in memory. If this becomes
      too memory
  * intensive, it may be best to switch to a {@link android.
      support.v4.app.FragmentStatePagerAdapter}.
  */
  SectionsPagerAdapter mSectionsPagerAdapter;

  /**
  * The {@link ViewPager} that will host the section contents.
  */
  ViewPager mViewPager;
  static public double angleTouch=0;
  static public double powerTouch=0;

  private Compass compass;

  static public boolean backPressed = false;


  @Override
  protected void onCreate(Bundle savedInstanceState) {

    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_sensor_selection);
    //setContentView(R.layout.layout_cg);
```

```
// Create the adapter that will return a fragment for each
    of the three
// primary sections of the app.
mSectionsPagerAdapter = new SectionsPagerAdapter(
    getSupportFragmentManager());

// Set up the ViewPager with the sections adapter.
mViewPager = (ViewPager) findViewById(R.id.pager);
mViewPager.setAdapter(mSectionsPagerAdapter);

compass = new Compass(this);
compass.arrowView = (ImageView) findViewById(R.id.
    main_image_hands);

LinearLayout rotation = (LinearLayout) findViewById(R.id.
    rotation); //
rotation.setVisibility(View.GONE);
RelativeLayout rotation2 = (RelativeLayout) findViewById(R.
    id.rotation2);

com.erz.joysticklibrary.JoyStick joy1 = (com.erz.
    joysticklibrary.JoyStick) findViewById(R.id.joy1);
com.crystal.crystalrangeseekbar.widgets.CrystalSeekbar
    UpDown = (com.crystal.crystalrangeseekbar.widgets.
    CrystalSeekbar) findViewById(R.id.UpDown);

UpDown.setOnSeekbarChangeListener(new
    OnSeekbarChangeListener() {
  @Override
  public void valueChanged(Number minValue) {
    HomeActivity.altura=Integer.parseInt(String.valueOf(
        minValue));
  }
});

joy1.setListener(this);
joy1.setPadColor(Color.parseColor("#55ffffff"));
joy1.setButtonColor(Color.parseColor("#55ff0000"));

Connect.addMyBooleanListener(new com.example.david.tfg.
    ConnectionBooleanChangedListener5() {
  @Override
  public void OnMyBooleanChanged() {
    if (Connect.getMyBoolean()) {
      LinearLayout rotation = (LinearLayout) findViewById(R.
          id.rotation);
      rotation.setVisibility(View.GONE);
    }
  }
});
if(HomeActivity.compass){
  rotation2.setVisibility(View.VISIBLE);
} else {
```

```
      rotation2.setVisibility(View.GONE);
    }
    if(HomeActivity.seekbar){
      UpDown.setVisibility(View.VISIBLE);
    } else {
      UpDown.setVisibility(View.GONE);
    }
    if(HomeActivity.joystick){
      joy1.setVisibility(View.VISIBLE);
      //joy2.setVisibility(View.VISIBLE);
    } else {
      joy1.setVisibility(View.GONE);
      //joy2.setVisibility(View.GONE);
    }

    // Check if device has a hardware gyroscope
    SensorChecker checker = new HardwareChecker((SensorManager)
        getSystemService(SENSOR_SERVICE));
    if(!checker.IsGyroscopeAvailable()) {
      // If a gyroscope is unavailable, display a warning.
      displayHardwareMissingWarning();
    }
  }
  @Override
  public void onBackPressed() {

    backPressed = true;

    super.onBackPressed();
    finish();
  }

  private void displayHardwareMissingWarning() {
    AlertDialog ad = new AlertDialog.Builder(this).create();
    ad.setCancelable(false); // This blocks the 'BACK' button
    ad.setTitle(getResources().getString(R.string.
        gyroscope_missing));
    ad.setMessage(getResources().getString(R.string.
        gyroscope_missing_message));
    ad.setButton(DialogInterface.BUTTON_NEUTRAL, getResources().
        getString(R.string.OK), new DialogInterface.
        OnClickListener() {
      @Override
      public void onClick(DialogInterface dialog, int which) {
        dialog.dismiss();
      }
    });
    ad.show();
  }

  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu; this adds items to the action bar if it
```

```
        is present.

    if(HomeActivity.i==5){
      getMenuInflater().inflate(R.menu.menu_cg, menu);
    }else if(HomeActivity.i==1 || HomeActivity.i==3){
      getMenuInflater().inflate(R.menu.sensor_selection, menu);
    } else {
      getMenuInflater().inflate(R.menu.menu_home, menu);
    }
    return true;
  }

  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    if(HomeActivity.i==5){

      switch (item.getItemId())
      {
        case R.id.action_settings2:
        showSettings();
        default:
        break;
      }
    } else if(HomeActivity.i==1 || HomeActivity.i==3){
      switch (item.getItemId())
      {
        case R.id.absolute:
        if(HomeActivity.altura==0){
          HomeActivity.mode=0;
        } else {
          Toast toast = Toast.makeText(getApplicationContext(),
              "Set height to 0 to change mode", Toast.
              LENGTH_SHORT);
          toast.show();
        }
        break;
        case R.id.relative:
        if(HomeActivity.altura==0){
          HomeActivity.mode=1;
        } else {
          Toast toast = Toast.makeText(getApplicationContext(),
              "Set height to 0 to change mode", Toast.
              LENGTH_SHORT);
          toast.show();
        }
        break;
        case R.id.position:
        if(HomeActivity.altura==0){
          HomeActivity.mode=2;
        } else {
          Toast toast = Toast.makeText(getApplicationContext(),
              "Set height to 0 to change mode", Toast.
              LENGTH_SHORT);
```

```
          toast.show();
        }
        break;
        default:
        break;
      }
    }

    return false;
}

@Override
public void onMove(JoyStick joyStick, double angle, double
    power, int direction) {
    switch (joyStick.getId()) {
      case R.id.joy1:
      //gameView.move(angle, power);
      angleTouch=angle;
      powerTouch=power;
      break;
    }
}

@Override
public void onTap() {

}

@Override
public void onDoubleTap() {

}
@Override
protected void onStart() {
    super.onStart();
    //Log.d(TAG, "start compass");
    compass.start();
}

@Override
protected void onPause() {
    super.onPause();
    compass.stop();
}

@Override
protected void onResume() {
    super.onResume();
    compass.start();
}

@Override
protected void onStop() {
```

```
    super.onStop();
    //Log.d(TAG, "stop compass");
    compass.stop();
  }

  /**
  * A {@link FragmentPagerAdapter} that returns a fragment
     corresponding to
  * one of the sections/tabs/pages.
  */
  public class SectionsPagerAdapter extends FragmentPagerAdapter
      {

    /**
    * Initialises a new sectionPagerAdapter
    *
    * @param fm the fragment Manager
    */
    public SectionsPagerAdapter(FragmentManager fm) {
      super(fm);
    }

    @Override
    public Fragment getItem(int position) {
      // getItem is called to instantiate the fragment for the
         given page.
      // Return a DummySectionFragment (defined as a static
         inner class
      // below) with the page number as its lone argument.
      Fragment fragment = new OrientationVisualisationFragment()
         ;
      Bundle args = new Bundle();
      if(getCount()==1){
        position=OrientationVisualisationFragment.SensorSelected
           -1;//cambiar los sensores ajustes
      }
      args.putInt(OrientationVisualisationFragment.
         ARG_SECTION_NUMBER, position + 1);
      fragment.setArguments(args);
      return fragment;
    }

    @Override
    public int getCount() {
      // Show 6 total pages.
      return HomeActivity.num_fused_sensors;
    }

    @Override
    public CharSequence getPageTitle(int position) {
      Locale l = Locale.getDefault();

      if(getCount()==1){
```

```
        position=OrientationVisualisationFragment.SensorSelected
            -1; //cambiar los sensores ajustes
      }
      switch (position) {
        case 0:
        return getString(R.string.title_section1).toUpperCase(l)
            ;
        case 1:
        return getString(R.string.title_section2).toUpperCase(l)
            ;
        case 2:
        return getString(R.string.title_section3).toUpperCase(l)
            ;
        case 3:
        return getString(R.string.title_section4).toUpperCase(l)
            ;
        case 4:
        return getString(R.string.title_section5).toUpperCase(l)
            ;
        case 5:
        return getString(R.string.title_section6).toUpperCase(l)
            ;
      }
      return null;
    }
  }

  private void showSettings()
  {

    Intent i = new Intent(this, Settings.class);
    startActivity(i);
  }




}
```

D.8 *Settings.java*

```
package com.example.david.tfg;

/**
* Created by David on 03/07/2017.
*/

import android.content.Context;
import android.content.Intent;
import android.content.SharedPreferences;
```

```
import android.content.res.Configuration;
import android.os.Bundle;
import android.preference.CheckBoxPreference;
import android.preference.EditTextPreference;
import android.preference.ListPreference;
import android.preference.Preference;
import android.preference.PreferenceActivity;
import android.preference.PreferenceFragment;
import android.util.Log;

import com.example.david.tfg.R;

import java.util.Locale;

public class Settings extends PreferenceActivity {

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    getFragmentManager().beginTransaction().replace(android.R.id
       .content, new SettingsFragment()).commit();

  }

  public static class SettingsFragment extends
     PreferenceFragment
  {
    @Override
    public void onCreate(final Bundle savedInstanceState) {


      if (HomeActivity.i == 5) {

        super.onCreate(savedInstanceState);
        addPreferencesFromResource(R.xml.settings_cg);

        final EditTextPreference x = (EditTextPreference)
           findPreference("x");
        x.setOnPreferenceChangeListener(new Preference.
           OnPreferenceChangeListener() {
          @Override
          public boolean onPreferenceChange(Preference
             preference, Object newValue) {
            String editValue = (String) newValue;
            OrientationVisualisationFragment._x = editValue;
            OrientationVisualisationFragment.x = Float.
               parseFloat(editValue);
            return true;
          }
        });

        final EditTextPreference y = (EditTextPreference)
           findPreference("y");
```

```
        y.setOnPreferenceChangeListener(new Preference.
            OnPreferenceChangeListener() {
          @Override
          public boolean onPreferenceChange(Preference
              preference, Object newValue) {
            String editValue = (String) newValue;
            OrientationVisualisationFragment._y = editValue;
            OrientationVisualisationFragment.y = Float.
                parseFloat(editValue);
            return true;
          }
        });

        final EditTextPreference Ds = (EditTextPreference)
            findPreference("Ds");
        Ds.setOnPreferenceChangeListener(new Preference.
            OnPreferenceChangeListener() {
          @Override
          public boolean onPreferenceChange(Preference
              preference, Object newValue) {
            String editValue = (String) newValue;
            OrientationVisualisationFragment._Ds = editValue;
            OrientationVisualisationFragment.Ds = Float.
                parseFloat(editValue);
            return true;
          }
        });

        final EditTextPreference k = (EditTextPreference)
            findPreference("k");
        k.setOnPreferenceChangeListener(new Preference.
            OnPreferenceChangeListener() {
          @Override
          public boolean onPreferenceChange(Preference
              preference, Object newValue) {
            String editValue = (String) newValue;
            OrientationVisualisationFragment._k = editValue;
            OrientationVisualisationFragment.k = Float.
                parseFloat(editValue);
            return true;
          }
        });

        final EditTextPreference Dvs = (EditTextPreference)
            findPreference("Dvs");
        Dvs.setOnPreferenceChangeListener(new Preference.
            OnPreferenceChangeListener() {
          @Override
          public boolean onPreferenceChange(Preference
              preference, Object newValue) {
            String editValue = (String) newValue;
            OrientationVisualisationFragment._Dvs = editValue;
            OrientationVisualisationFragment.Dvs = Float.
```

```
                parseFloat(editValue);
            return true;
          }
        });

        final EditTextPreference v_max = (EditTextPreference)
            findPreference("v_max");
        v_max.setOnPreferenceChangeListener(new Preference.
            OnPreferenceChangeListener() {
          @Override
          public boolean onPreferenceChange(Preference
              preference, Object newValue) {
            String editValue = (String) newValue;
            OrientationVisualisationFragment._v_max = editValue;
            OrientationVisualisationFragment.v_max = Float.
                parseFloat(editValue);
            return true;
          }
        });

        final EditTextPreference w_max = (EditTextPreference)
            findPreference("w_max");
        w_max.setOnPreferenceChangeListener(new Preference.
            OnPreferenceChangeListener() {
          @Override
          public boolean onPreferenceChange(Preference
              preference, Object newValue) {
            String editValue = (String) newValue;
            OrientationVisualisationFragment._w_max = editValue;
            OrientationVisualisationFragment.w_max = Float.
                parseFloat(editValue);
            return true;
          }
        });


      } else if(HomeActivity.i == 10){


        super.onCreate(savedInstanceState);
        addPreferencesFromResource(R.xml.settings);

        final ListPreference listPreference = (ListPreference)
            findPreference("SensorFusion");
        listPreference.setOnPreferenceChangeListener(new
            Preference.OnPreferenceChangeListener() {
          @Override
          public boolean onPreferenceChange(Preference
              preference, Object newValue) {
            String listValue = (String) newValue;
            OrientationVisualisationFragment.SensorSelected =
                Integer.parseInt(listValue);
            return true;
```

```
        }
      });
      final ListPreference languagePreference = (
         ListPreference) findPreference("language");
      languagePreference.setOnPreferenceChangeListener(new
         Preference.OnPreferenceChangeListener() {
        @Override
        public boolean onPreferenceChange(Preference
           preference, Object newValue) {
          String listValue = (String) newValue;
          OrientationVisualisationFragment.language = Integer.
             parseInt(listValue);
          Locale localizacion;
          switch (OrientationVisualisationFragment.language) {
            case 1:
            localizacion = new Locale("en", "EN");
            break;
            case 2:
            localizacion = new Locale("es", "ES");
            break;
            case 3:
            localizacion = new Locale("ca", "CA");
            break;
            default:
            localizacion = new Locale("en", "EN");
            break;
          }
          Locale.setDefault(localizacion);
          Configuration config = new Configuration();
          config.locale = localizacion;
          getActivity().getBaseContext().getResources().
             updateConfiguration(config, getActivity().
             getBaseContext().getResources().getDisplayMetrics
             ());
          final SharedPreferences prefs = getActivity().
             getSharedPreferences("SettingsDroneControl",
             Context.MODE_PRIVATE);

          SharedPreferences.Editor editor = prefs.edit();
          editor.putInt("language",
             OrientationVisualisationFragment.language);
          editor.commit();

          refresh();
          return true;
        }
      });
      final EditTextPreference sensibility = (
         EditTextPreference) findPreference("sense");
      sensibility.setOnPreferenceChangeListener(new Preference
         .OnPreferenceChangeListener() {
        @Override
        public boolean onPreferenceChange(Preference
```

```
            preference, Object newValue) {
          String editValue = (String) newValue;
          OrientationVisualisationFragment.sensibility = Float
              .parseFloat(editValue + 'f');
          return true;
        }
      });
      final EditTextPreference IPaddress = (EditTextPreference
          ) findPreference("ip");
      IPaddress.setOnPreferenceChangeListener(new Preference.
          OnPreferenceChangeListener() {
        @Override
        public boolean onPreferenceChange(Preference
            preference, Object newValue) {
          String editValue = (String) newValue;
          OrientationVisualisationFragment.ip = editValue;
          return true;
        }
      });
      final EditTextPreference Port = (EditTextPreference)
          findPreference("port");
      Port.setOnPreferenceChangeListener(new Preference.
          OnPreferenceChangeListener() {
        @Override
        public boolean onPreferenceChange(Preference
            preference, Object newValue) {
          String editValue = (String) newValue;
          OrientationVisualisationFragment.port = Integer.
              parseInt(editValue);
          return true;
        }
      });
      final EditTextPreference MaxVel = (EditTextPreference)
          findPreference("maxVel");
      MaxVel.setOnPreferenceChangeListener(new Preference.
          OnPreferenceChangeListener() {
        @Override
        public boolean onPreferenceChange(Preference
            preference, Object newValue) {
          String editValue = (String) newValue;
          OrientationVisualisationFragment.MaxLinealVel =
              Integer.parseInt(editValue);
          return true;
        }
      });
      final EditTextPreference MaxHeight = (EditTextPreference
          ) findPreference("maxAltura");
      MaxHeight.setOnPreferenceChangeListener(new Preference.
          OnPreferenceChangeListener() {
        @Override
        public boolean onPreferenceChange(Preference
            preference, Object newValue) {
          String editValue = (String) newValue;
```

```
            OrientationVisualisationFragment.alturaMax = Integer
                .parseInt(editValue);
            return true;
          }
        });
        final EditTextPreference MaxArea = (EditTextPreference)
            findPreference("maxArea");
        MaxArea.setOnPreferenceChangeListener(new Preference.
            OnPreferenceChangeListener() {
          @Override
          public boolean onPreferenceChange(Preference
              preference, Object newValue) {
            String editValue = (String) newValue;
            OrientationVisualisationFragment.areaMax = Integer.
                parseInt(editValue);
            return true;
          }
        });
        final CheckBoxPreference checkboxpreference = (
            CheckBoxPreference) findPreference("compass");
        checkboxpreference.setOnPreferenceChangeListener(new
            Preference.OnPreferenceChangeListener() {
          @Override
          public boolean onPreferenceChange(Preference
              preference, Object newValue) {
            HomeActivity.compass = Boolean.valueOf(newValue.
                toString());
            return true;
          }
        });

      }
    }
  }

  public static void refresh(){
    Intent intent=new Intent();
    intent.setClass(HomeActivity.home, HomeActivity.home.
        getClass());
    HomeActivity.home.finish();
    HomeActivity.home.startActivity(intent);
  }

}
```

D.9 *menu_cg.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    >
```

```
  <item
    android:id="@+id/action_settings2"
    android:title="Coordenadas">
  </item>

</menu>
```

D.10 *settings_cg.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/
   res/android">

  <PreferenceCategory android:title="Insertar coordenadas 'x' e
     'y' del punto objetivo (en metros)." >

    <EditTextPreference

      android:defaultValue="0.0"
      android:dialogMessage="Nota: Se pueden introducir valores
         en coma flotante,
      se truncará hasta los dos primeros decimales."
      android:dialogTitle="Coordenada X"
      android:inputType="textNoSuggestions"

      android:key="x"
      android:title="Coordenada 'x'" />

    <EditTextPreference
      android:defaultValue="0.0"
      android:dialogMessage="Nota: Se pueden introducir valores
         en coma flotante,
      se truncará hasta los dos primeros decimales."
      android:dialogTitle="Coordenada Y"
      android:inputType="textNoSuggestions"
      android:key="y"
      android:title="Coordenada 'y'" />

    </PreferenceCategory>

  <PreferenceCategory android:title="Insertar distancia de
     seguridad Ds.">

    <EditTextPreference

      android:defaultValue="0.0"
      android:dialogMessage="Nota: Se debe insertar en %. Por
         ejemplo, si se inserta un 50% Ds será un 50% más grande
          que el radio del robot"
```

```
    android:dialogTitle="Distancia de seguridad"
    android:inputType="textNoSuggestions"

    android:key="Ds"
    android:title="Ds" />
</PreferenceCategory>

<PreferenceCategory android:title="Insertar constante k.">

  <EditTextPreference

    android:defaultValue="0.0"
    android:dialogMessage="Nota: Se debe insertar un valor
       entre 1 y 3."
    android:dialogTitle="Constante k"
    android:inputType="textNoSuggestions"

    android:key="k"
    android:title="k" />
</PreferenceCategory>

<PreferenceCategory android:title="Insertar distancia segura
   de velocidad Dvs.">

  <EditTextPreference

    android:defaultValue="0.0"
    android:dialogTitle="Distancia segura de velocidad"
    android:inputType="textNoSuggestions"

    android:key="Dvs"
    android:title="Dvs" />
</PreferenceCategory>

<PreferenceCategory android:title="Insertar valores de
   velocidad lineal y angular máximas.">

  <EditTextPreference

    android:defaultValue="0.0"
    android:dialogTitle="Velocidad lineal máxima"
    android:inputType="textNoSuggestions"

    android:key="v_max"
    android:title="v_max" />

  <EditTextPreference

    android:defaultValue="0.0"
    android:dialogTitle="Velocidad angular máxima"
    android:inputType="textNoSuggestions"

    android:key="w_max"
```

```
      android:title="w_max" />
  </PreferenceCategory>



</PreferenceScreen>
```

BIBLIOGRAFÍA

[1] 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. 1.2, 2, 2.3, 2.5, 4.2, 4.2.1, 4.2.5

[2] M. L. Enrique Sucar, "Robótica inteligente." [Online]. Available: https://ccc.inaoep.mx/~esucar/Clases-rob/clase06-programa.ppt 1.3

[3] D. Arévalo Jiménez, "Desarrollo de una interfaz de teleoperación de robots móviles para dispositivos android," 2017. 1.4, 3.1.2, 3.1.3, 3.2, 3.4.1, 3.4.3, 5.1

[4] Wikipedia, "Android studio." [Online]. Available: https://es.wikipedia.org/wiki/Android_Studio 1.5.1

[5] ——, "Sistema operativo robótico." [Online]. Available: https://es.wikipedia.org/wiki/Sistema_Operativo_Rob%C3%B3tico 1.5.2

[6] ——, "Matlab." [Online]. Available: https://es.wikipedia.org/wiki/MATLAB 1.5.3

[7] I. Javier Minguez, Associate Member and I. Luis Montano, Member, "Nearness diagram (nd) navigation: Collision avoidance in troublesome scenarios," *IEEE TRANSACTIONS ON ROBOTICS AND AUTOMATION, VOL. 20, NO. 1*, FEBRUARY 2004. 2.1, A

[8] J. W. Durham and F. Bullo, "Smooth nearness-diagram navigation," *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Nice, France, Sept, 22-26, 2008. 2.1, 2.2, 2.5, 2.5

[9] Wikipedia, "Socket de internet." [Online]. Available: https://es.wikipedia.org/wiki/Socket_de_Internet 3.1.3

[10] L. Pakus, "Ejemplos de c/c++ sencillos: Intersección recta- circunferencia." [Online]. Available: http://lordpakus.blogspot.com.es/2013/09/ejemplos-de-cc-sencillos-interseccion.html 3.3.1