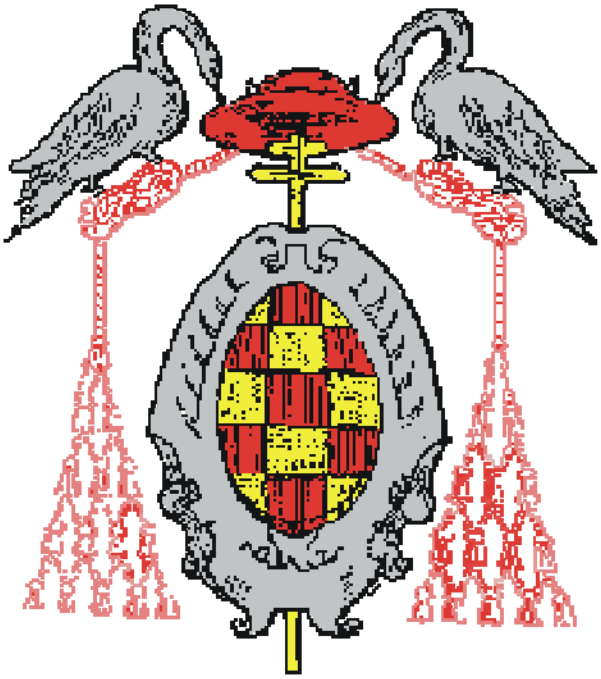
UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

INGENIERÍA TÉCNICA INDUSTRIAL
ESPECIALIDAD EN ELECTRÓNICA INDUSTRIAL

Trabajo Fin de Carrera



Generación de trayectorias con Player para la navegación de un robot móvil en espacios inteligentes

Inés Sanz Alonso

Septiembre de 2008

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

INGENIERÍA TÉCNICA INDUSTRIAL
ESPECIALIDAD EN ELECTRÓNICA INDUSTRIAL

Trabajo Fin de Carrera

Generación de trayectorias con Player para la navegación de un robot móvil en espacios inteligentes

Autora: Inés Sanz Alonso
Tutora: Marta Marrón Romera

TRIBUNAL:

Presidente: D. Elena López Guillén

Vocal 1º: D. Daniel Pizarro Pérez

Vocal 2º: D. Marta Marrón Romera

CALIFICACIÓN.....

FECHA.....

A mis padres y hermanos, por haberme dado la oportunidad y apoyarme siempre. A mis abuelos por ser para mi un ejemplo de superación. Y a Javi, Cris y los amigos de la universidad por haberme ayudado tanto.

I. RESUMEN

Este proyecto tiene como objetivo dotar a un robot móvil de las herramientas software adecuadas para que sea capaz de navegar en un entorno con obstáculos. Se pretende utilizar la plataforma Player-Stage para diseñar una serie de comportamientos que rijan los movimientos del robot.

Para el diseño de los mencionados comportamientos, se propone utilizar el algoritmo de planificación global Wavefront, acompañado del planificador local VFH y del filtro de partículas AMCL.

Palabras clave: Software Player-Stage, robot móvil, planificación global y local, navegación.

CAPÍTULO 1

INTRODUCCIÓN

1.1. Motivaciones

El trabajo desarrollado a lo largo de este proyecto se halla englobado en el campo de la robótica móvil.

El mundo de la robótica está en creciente desarrollo por lo extendidas que están sus aplicaciones. Podemos encontrar aplicaciones robóticas en numerosos ámbitos de la vida cotidiana. Por ejemplo, se pueden desarrollar aplicaciones que sirvan de apoyo a personas con movilidad reducida (sillas de ruedas robotizadas), o que faciliten la limpieza de una vivienda (robot-aspiradora).

Es por eso, que en este proyecto se propone implementar una serie de comportamientos para la navegación de robots móviles, que puedan ser aplicables en un futuro a aplicaciones domóticas, de movilidad, etc.

Para el diseño de estos comportamientos conviene utilizar una herramienta software como enlace entre el robot y el ordenador que lo controla. Lo más eficaz es que dicha herramienta sea portable y flexible, para que el código que generemos sea aplicable a diversas plataformas robóticas. En el caso de este proyecto se ha utilizado Player como enlace entre robot y PC.

1.2. Objetivos perseguidos en el trabajo

Como se ha dicho en el apartado anterior, con este proyecto se pretende programar comportamientos para la navegación con robots móviles en espacios conocidos. Se ha buscado diseñar comportamientos que sean útiles a la hora de integrarlos en aplicaciones robóticas más complejas.

Para el desarrollo de los mencionados comportamientos, primero se propone estudiar el funcionamiento del planificador global Wavefront, el planificador local VFH, y el filtro de partículas AMCL dentro de un entorno acotado y con obstáculos.

Partiendo de los algoritmos mencionados se propone una serie de comportamientos para la navegación autónoma o semi-autónoma del robot evitando los obstáculos estáticos o móviles que puedan encontrarse en su trayectoria.

Se han implementado un total de cuatro comportamientos que se describen brevemente a continuación:

- El primer comportamiento que se propone es desplazar el robot de un punto a otro del entorno sin que colisione con ningún obstáculo. A este comportamiento se le denomina de aquí en adelante movimiento básico.

- El siguiente comportamiento diseñado se basa en el primero, consiste en que el robot recorra una serie de puntos introducidos por el usuario, al igual que en el anterior, evitando los obstáculos de su trayectoria. Este comportamiento puede ser útil, por ejemplo, para hacer que el robot se desplace para realizar un conjunto de tareas en diferentes puntos del entorno.

- El tercero de los comportamientos creados consiste en que el robot siga las paredes de la estancia donde se encuentra. Se puede emplear este comportamiento para realizar un reconocimiento del entorno donde se halla.

- Y el último de los comportamientos propuestos consiste en que el robot siga el movimiento de una persona. Este comportamiento podría emplearse para desarrollar una aplicación robótica de ayuda personal.

1.3. Estructura de la memoria

Esta memoria está distribuida en seis capítulos. El contenido de éstos se describe a continuación:

- **Capítulo 1. Introducción**: Capítulo actual, en el se introduce el trabajo que se describe en la memoria de este proyecto.

- **Capítulo 2. Plataforma de navegación**: En este capítulo se describen la plataforma software con la que se va a trabajar (Player/Stage/Gazebo), y las herramientas hardware de las que se dispone para realizar las pruebas oportunas.

- **Capítulo 3. Aplicaciones para la navegación**: Este es el capítulo donde se describe los algoritmos utilizados a la hora de programar las aplicaciones. Se analiza en detalle el funcionamiento de los algoritmos Wavefront y VFH, y del filtro de partículas AMCL, y el de sus respectivos drivers de Player.

- **Capítulo 4. Comportamientos desarrollados**: En este capítulo se presentan los comportamientos que se han desarrollado para el manejo del robot. Se detalla el funcionamiento de cada uno de ellos y se adjunta su flujograma para hacer más fácil la comprensión del programa de control. Además de comentan sus ventajas e inconvenientes.

- **Capítulo 5. Resultados**: En este capítulo se exponen los resultados obtenidos en las diferentes pruebas realizadas con los robots y la plataforma software.

- **Capítulo 6. Conclusiones y trabajos futuros**: En este capítulo se incluyen, por una parte, las conclusiones a las que se llega después del trabajo realizado y, por otra, los trabajos futuros que se proponen siguiendo la misma temática y a la vista de los problemas y resultados alcanzados.

A continuación de la memoria se incluyen el manual de usuario, el pliego de condiciones, el presupuesto, los planos y la bibliografía de este proyecto.

CAPÍTULO 2

PLATAFORMA DE NAVEGACIÓN

Como se explica en el capítulo de introducción. En este capítulo se pasa a explicar los elementos, tanto software como hardware, que se han utilizado en la realización de este proyecto. En cuanto a hardware se van a describir los robots utilizados de la marca Mobile Robots© y el láser de la marca URG©. Y en cuando a software se van a describir las herramientas Player y Stage, incluidas en el Proyecto Player (*The Player Proyect*).

2.1. Elementos Software

Para la puesta en marcha de este proyecto se ha utilizado la plataforma de desarrollo Player/Stage/Gazebo (PSG). Esta plataforma proporcionada por el Proyecto Player está formada por el servidor Player, y los simuladores Stage (2D) y Gazebo (3D), de los cuales sólo se utilizará el primero. [PlayerProyect]

El programa servidor es el encargado de la adquisición de datos de los sensores, y de generar las órdenes para comandar los actuadores. Mientras que el programa cliente es el sistema de control de la aplicación. Stage, por su parte, es el simulador que permite probar las aplicaciones desarrolladas en un entorno ideal, antes de probarlas en el entorno real.

Esta plataforma Player/Stage/Gazebo se encuentra bajo licencia pública de GNU, es decir, es software libre. Además está diseñada para funcionar con robots de diferentes

marcas, y permite escribir los programas de control en varios lenguajes. Esta característica proporciona portabilidad a las aplicaciones que se desarrollan con dicha plataforma.

2.1.1. Player

Player es el servidor al que se conecta el programa de la aplicación diseñada por el usuario para leer sensores o comandar actuadores del robot que se esté manejando.

La herramienta Player constituye una red de servicios para el control de robots. Ejecutándolo sobre un robot, Player sirve de interfaz para comunicar los sensores y actuadores del robot que se quiera manejar con el PC que lo controla, mediante la IP del robot. El programa cliente se comunica con Player mediante un socket TCP, leyendo datos de los sensores, escribiendo comandos en los actuadores, y configurando dispositivos según requiera el caso. [PlayerProyect] En el Manual de usuario de este libro se puede consultar como se realiza esta conexión.

Se puede usar este servidor con una amplia variedad de robots. Aunque la plataforma original para Player es la familia Pioneer 2 de Mobile Robots© (antigua ActivMedia©), se puede usar con otros robots y sensores muy comunes, por ejemplo con robots de las marcas Botrics©, iRobot© o Segway©, y con sensores de las marcas SICK©, Sony© o Canon©. Al estar estructurado en módulos, Player permite dar soporte a nuevo hardware, y, al estar bajo licencia GNU, la comunidad de usuarios y programadores contribuye a ello programando nuevos drivers y aplicaciones adaptados a sus necesidades, pero que también están al alcance de otros usuarios.

Player está diseñado para ser independiente del lenguaje en el que se programan las aplicaciones y de la plataforma sobre la que se trabaja. Un programa cliente puede ser ejecutado en cualquier PC que tenga conexión de red con el robot a controlar, y puede estar escrito en cualquier lenguaje compatible con los sockets TCP que usa Player para conectar con la plataforma robótica. Se suelen encontrar aplicaciones escritas en lenguajes como C, C++, Java, Python, etc. Por otra parte, Player da libertad en cuanto a la estructura de las aplicaciones de control: se puede crear un programa cliente con varias tareas concurrentes gestionadas con un sistema de semáforos, o simplemente, se puede escribir un programa con un sencillo bucle de lectura-computación- acción.

La estructura básica de una aplicación cliente debe seguir las pautas listadas a continuación. De ellas, las instrucciones 3 y 4 se repiten tantas veces como requiera la aplicación programada.

1. Establecer la conexión con el servidor
2. Suscribirse a los dispositivos necesarios para la aplicación
3. Leer datos de los sensores
4. Enviar comandos de actuación

5. Retirar la suscripción a los servicios antes mencionados
6. Retirar la conexión con el servidor

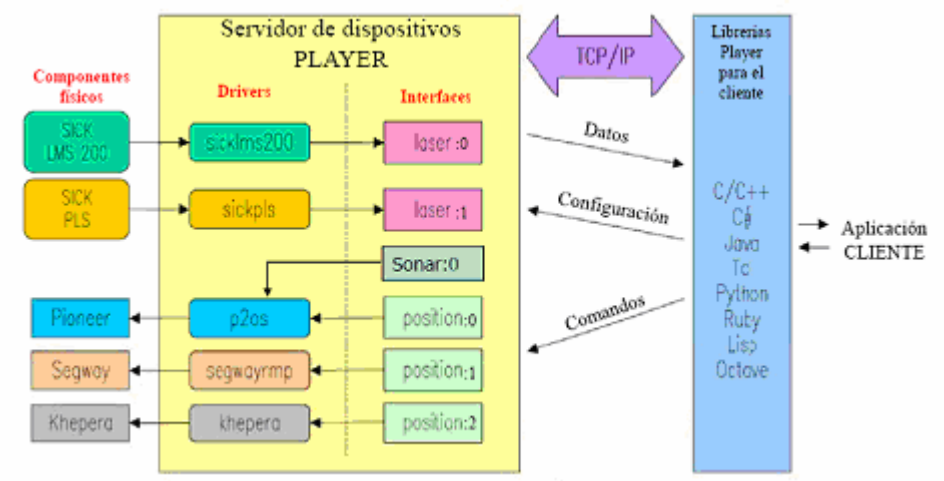


Figura 2. 1 - Modelo de programa Cliente/Servidor de una aplicación utilizando Player [Ocaña08]

En la Figura 2.1 se puede ver como están relacionados los componentes físicos con los drivers de Player y, a su vez, con la aplicación cliente, en un programa básico Cliente/Servidor.

Al margen de su versatilidad, otra de las ventajas de Player es que presenta la misma interfaz para dispositivos de diferentes fabricantes (Figura 2.2). Lo que permite que el mismo programa de control se pueda utilizar en varios tipos de robots. Esta característica de Player resulta muy útil a la hora de utilizar el simulador Stage, porque los programas de control escritos para los robots simulados de Stage son válidos y directamente aplicables para trabajar sobre robots reales.

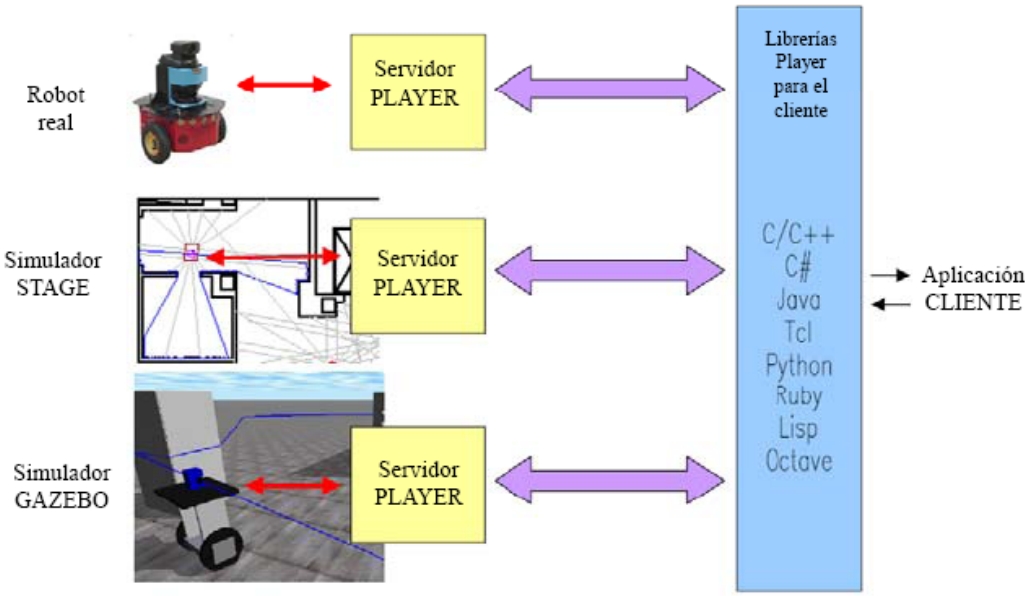


Figura 2. 2 - Abstracción del hardware gracias a que Player presenta la misma interfaz para dispositivos de diferentes marcas [Ocaña08]

Player también trae asociadas otras aplicaciones para monitorizar y controlar el comportamiento del robot con el que se está trabajando: Playerv, Playerjoy, Playernav… De esas aplicaciones asociadas nos ha sido especialmente útil Playerv. Esta herramienta es un programa cliente tipo GUI que visualiza datos de los dispositivos del servidor Player. Al ejecutar esta utilidad se crea una ventana en la que podemos visualizar datos recibidos de los dispositivos y, a su vez, mandar consignas para comandarlos.

Cabe destacar que Player es una herramienta de reciente creación y que aun sigue en vías de construcción. En la página web del Proyecto Player encontramos una invitación a los usuarios a que completen o cambien los contenidos que crean convenientes, y compartan dichas modificaciones con el resto de usuarios.

2.1.2. Stage

La herramienta Stage permite simular el comportamiento de diferentes tipos de robots, así como de sus sensores, dentro de un entorno cualquiera de dos dimensiones. Stage es proveedor de modelos computacionalmente sencillos de muchos dispositivos y, además, emula cada uno de ellos con gran precisión. En la Figura 2.3 vemos un ejemplo de simulación con Stage.

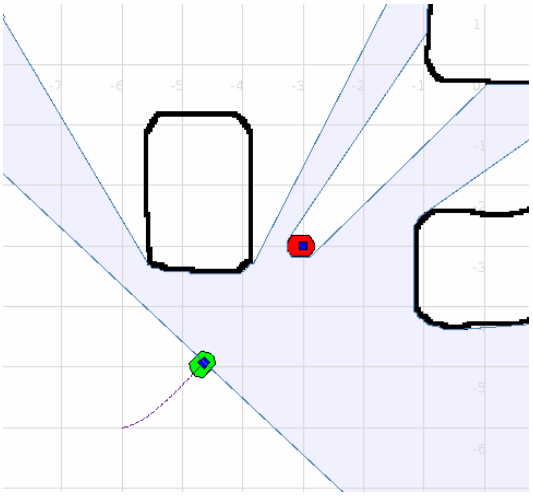


Figura 2. 3 - Ejemplo de simulación con Stage

Existen dos modos de usar Stage: uno como un dispositivo más de Player (*libstageplugin*) y otro como librería independiente en C (*libstage*).

El uso más común de Stage es como *plugin* de Player, en ese caso provee de numerosos dispositivos virtuales para Player. Se escriben los controladores y los algoritmos sensores para los robots como clientes para el servidor Player. No hay diferencias entre los dispositivos reales de un robot y sus simulaciones equivalentes de Stage. Los programas clientes de Player desarrollados usando Stage funcionan casi igual en robots reales, y viceversa, encontrando entre ellos muy pequeños cambios como, por ejemplo, los errores de odometría debidos a deslizamientos de las ruedas. Esto permite simular con gran exactitud las aplicaciones desarrolladas antes de ejecutarlas en un robot físico.

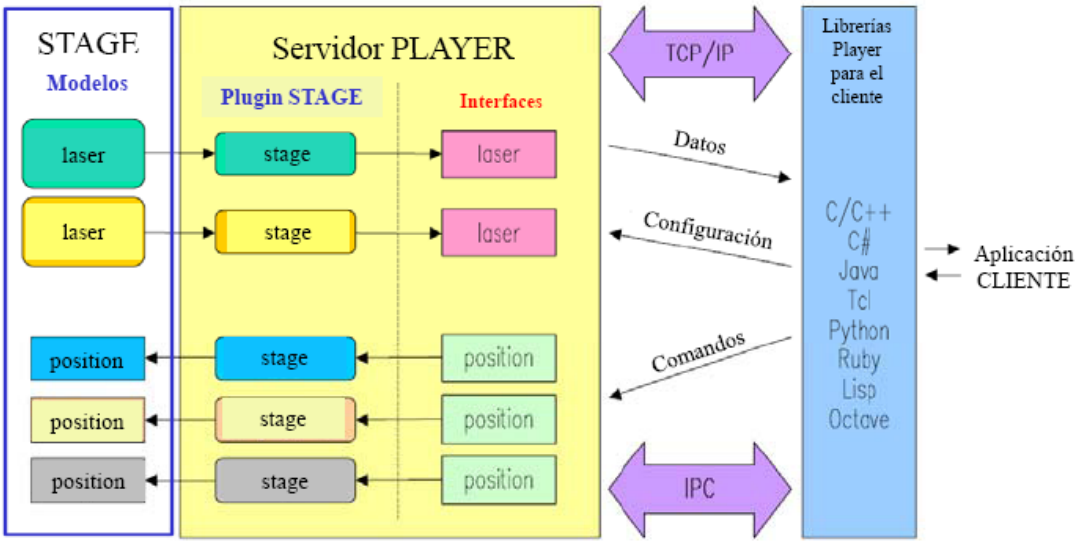


Figura 2. 4 - Uso de Stage como dispositivo de Player (*libstageplugin*) [Ocaña08]

El uso de Stage como librería de C se aplica para incluir una simulación del robot dentro de un programa. Esto resulta útil si Player no es apropiado para las necesidades del programador. [PlayerProyect]

2.2. Elementos Hardware

En este apartado pasamos a describir los elementos hardware que se han utilizado durante el desarrollo de este proyecto: los robots Pionner y el láser Hokuyo. Para comandar los mencionados elementos se utiliza un ordenador portátil con una dristribución de linux (en este caso Ubuntu 7.10), y con conexión a la red local del laboratorio donde se realizan las pruebas.

2.2.1. Robots Pioneer

Se han probado las aplicaciones de control desarrolladas en varios robots de la marca MobileRobots: P3-DX, P3-AT y Amigobot.

El Pioneer3-DX tiene una base de propósito general y tracción diferencial (dos ruedas motrices y una de apoyo o *castor*). Tiene unas dimensiones de 44,5 x 40 x 24,5 cm, y un peso de 9 kg. Puede soportar una carga de hasta 23 kg, y alcanzar una velocidad lineal de hasta 1,2 m/s. Funciona con batería de 12v. Viene con un anillo delantero de 8 sensores de ultrasonidos y, opcionalmente, con un anillo trasero de otros 8 sensores.

El Pioneer3-AT viene con una base todo-terreno y tracción diferencial con cuatro ruedas (*skid-steer*). Sus dimensiones son 50 x 49 x 26 cm y tiene un peso de 12 kg. Es capaz de acarrear una carga de 30 kg y de moverse a una velocidad de hasta 1,2 m/s. Al igual que el anterior funciona con una batería de 12v. Y tiene la posibilidad de contar con un anillo delantero y otro trasero de 8 sensores de ultrasonidos.

Figura 2. 5 - Pioneer3-AT [ActivMedia]

Figura 2. 6 - Pioneer3-DX [ActivMedia]

El Amigobot es un robot de dimensiones más reducidas (33 x 28 x 15 cm) porque está pensado para aplicaciones multi-robot y para su uso docente. Tiene tracción diferencial (dos ruedas motrices y una *castor*) y un peso de 3,6 kg. Puede alcanzar un velocidad de hasta 1 m/s y es capaz de soportar una carga de máxima de 1kg. Éste también funciona con una batería de 12v. Viene con 6 sensores de ultrasonidos delanteros y dos traseros.

Figura 2. 7 – Amigobot [ActivMedia]

De los tres modelos citados en los que se han probado los programas de control desarrollados, el P3-DX se dejó de utilizar porque daba problemas con el algoritmo VFH. Y el P3-AT no funciona bien sobre el suelo pulido del laboratorio donde se realizan las pruebas, porqué está diseñado para su uso en exteriores.

2.2.2. Láser Hokuyo

Este láser URG-04LX es de pequeñas dimensiones (50x50x70 mm) y reducido peso (160 g). Tiene un área de reconocimiento de ± 120º, y es capaz de detectar objetos hasta una distancia de 4m. Funciona con una alimentación de 5v DC, y se conecta al PC con el que se trabaja mediante un puerto USB.

Figura 2. 8 - Laser Hokuyo [URG]

CAPÍTULO 3

APLICACIONES PARA LA NAVEGACIÓN

En este capítulo se van a explicar los algoritmos que se utilizan en las aplicaciones de navegación desarrolladas. Estos algoritmos se manejan en Player a través de las funciones recogidas en las librerías de este servidor, y de los parámetros incluidos en los drivers que implementan el funcionamiento de dichos algoritmos.

También se definirán en este capítulo los conceptos posicionamiento local y global, y por consecuencia, planificación local y global, ya que en este proyecto se usan ambas.

3.1. Sistemas de Posicionamiento Absoluto y Relativo

En este apartado se van a explicar las diferencias entre los sistemas de referencia (o sistemas coordenados) absolutos y relativos y, por consecuencia, las diferencias entre los sistemas de posicionamiento absolutos y relativos.

Un sistema de coordenadas es un conjunto de valores y puntos que permiten definir unívocamente la posición de cualquier punto de un espacio. En robótica móvil se usan habitualmente sistemas de coordenadas cartesianos de dos dimensiones porque los robots sólo pueden moverse dentro de un plano.

Sabiendo eso, se puede definir un sistema de referencia con un sistema de coordenadas y un punto de referencia. Dependiendo de ese punto de referencia tendremos un tipo de sistema de referencia u otro.

En aplicaciones de robótica móvil se considera que es un sistema de referencia local cuando se toma como punto de referencia el centro del robot. Estos sistemas de referencias se utilizan para hacer navegación reactiva. Si se toma como referencia un punto fijo externo al robot, se habla de sistema de referencia global. Estos otros sistemas se utilizan para hacer navegación global.

En la Figura 3.1 se muestra la diferencia entre los dos sistemas de referencia mencionados.

Para transformar un punto de un sistema de referencia a otro hay que seguir un procedimiento matemático que consiste en tomar las coordenadas de ese punto y multiplicarlas por una matriz de rotación y sumarles una matriz de traslación.

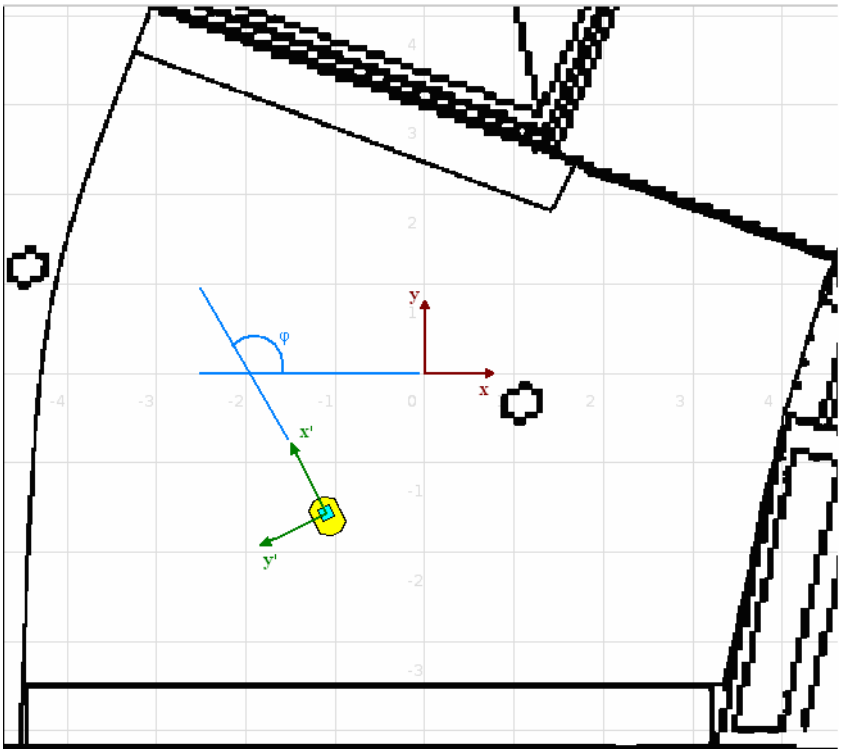


Figura 3. 1 - Sistemas de referencia global (rojo) y local (verde)

A continuación se detalla la ecuación para transformar las coordenadas de un punto de un sistema de referencia local a las correspondientes de un sistema global.

$$\begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} \cos\varphi & -sen\varphi \\ sen\varphi & \cos\varphi \end{pmatrix} \cdot \begin{pmatrix} x' \\ y' \end{pmatrix} + \begin{pmatrix} Xo \\ Yo \end{pmatrix}$$

Siendo: (X, Y) coordenadas en el SR global del punto que estamos transformando.

(Xo, Yo) coordenadas en el SR global del centro del SR local.

(x', y') coordenadas en el SR local del punto que estamos transformando.

φ ángulo diferencia entre los dos sistemas de referencia.

Dependiendo del tipo de sistema de referencia o posicionamiento que se utilice, se obtiene un tipo de navegación u otro. A continuación se explica cada tipo de navegación.

3.1.1. Navegación con posicionamiento relativo

La utilización de un sistema de posicionamiento relativo para navegación, da lugar a lo que se conoce como navegación local.

Como la navegación local se basa en un sistema de posicionamiento relativo, las posiciones que toma el robot a lo largo de sus movimientos están referidas a la posición inicial que tenía el robot al principio de la aplicación que se esté llevando a cabo. Normalmente se hallan las posiciones del robot relativas al inicio con las ecuaciones correspondientes a la odometría del robot, que tienen en cuenta las dimensiones y las características cinemáticas de éste. Este sistema de odometría se basa en sumar incrementos de posición a la posición inicial del robot, esto es lo que se conoce como *dead-reckoning*. Por esto, el gran problema de este método es la acumulación de errores. Para saber más sobre las ecuaciones que rigen la odometría del robot ver [Borenstein96].

En este proyecto, como se utiliza el dispositivo de posición de Player para trabajar con el sistema de odometría, no es necesario manejar las mencionadas ecuaciones de cinemática del robot.

Al trabajar sin un mapa del entorno se considera que este tipo de navegación es reactiva. Esto quiere decir que el robot se desplaza de un punto a otro del plano, utiliza sus sensores externos para detectar obstáculos, y sigue algún protocolo de actuación para evitarlos. Por tanto, se necesita que el robot disponga de al menos algún tipo de sensores externos para conocer qué hay a su alrededor.

Se engloban dentro de este tipo de navegación todos los métodos que incluyen el manejo de la odometría del robot junto con sus sensores, así como algunos algoritmos como VFH ó ND. En este proyecto se usa como navegador local el algoritmo VFH.

3.1.2. Navegación con posicionamiento absoluto

La utilización de un sistema de posicionamiento absoluto para navegación, da lugar a lo que se conoce como navegación global.

Como este tipo de navegación se basa en un sistema de posicionamiento absoluto, las posiciones que toma el robot a lo largo de sus movimientos están referidas a un punto externo a él.

Este tipo de navegación se puede realizar con o sin mapa. Cuando no se dispone de mapa, se dice que se está haciendo una navegación tipo SLAM (*Simultaneous Localization And Mapping*). Esto es que se construye un mapa de un entorno desconocido en el que se encuentra el robot, a la vez que mantiene información de su trayectoria dentro de dicho entorno.

Cuando se trabaja con mapa, se puede hacer con mapas topológicos o con mapas métricos. En los mapas topológicos se representan mediante nodos los lugares del entorno con alguna característica concreta, y con arcos se representan las conexiones de dichos nodos. En los mapas métricos se representa el espacio libre y el espacio ocupado (obstáculos) de forma geométrica. En el caso de este proyecto se trabaja con navegación con mapa métrico.

Haciendo navegación global con mapa, a parte de necesitar el mapa, se necesita un sistema de posicionamiento absoluto o global que es el que da la posición absoluta del robot con el que se trabaja, y un navegador local para que realice la evasión de obstáculos.

En este proyecto se ha utilizado el navegador global Wavefront junto con el localizador global AMCL y el navegador local VFH.

3.2. Descripción de los algoritmos utilizados

Para realizar las aplicaciones de control de robots incluidas en este proyecto se ha utilizado, fundamentalmente, el algoritmo de planificación global Wavefront. El driver de este algoritmo en Player va asociado a los drivers del algoritmos VFH (*Vector Field Histogram*), y AMCL (*Adaptive Monte-Carlo Localization*). El primero le sirve a Wavefront como planificador local, y del segundo toma la posición probable del robot para calcular los movimientos a realizar. En la Figura 3.2 aparecen representadas las comunicaciones entre los citados algoritmos.

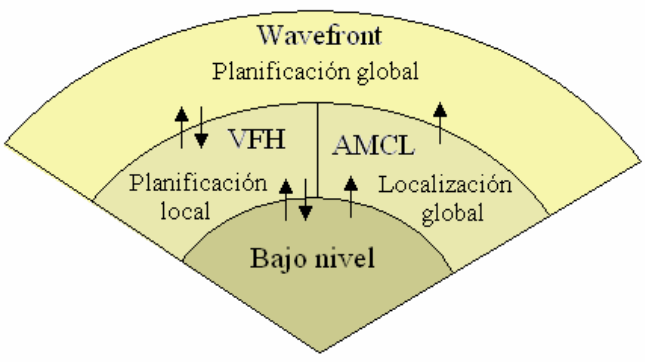


Figura 3. 2 - Relación entre los algoritmos de navegación Wavefront, VFH y AMCL

En los subapartados siguientes se explican con más detalle los algoritmos Wavefront, VFH y AMCL.

3.2.1. Wavefront

Se trata de un algoritmo de planificación global de trayectorias. Dicho algoritmo está basado en la estrategia de expansión de frente de ondas (*WaveFront Expansion*). [Barraquand90]

a) Modo de funcionamiento

Como se explica después en el punto b), este tipo de algoritmo necesita un sistema de posicionamiento global y un mapa del entorno en el que se trabaja. El funcionamiento del algoritmo se basa en los siguientes pasos:

- Configurar un espacio de trabajo en celdas a partir del mapa dado.
- Asignar a cada celda desde el destino un valor acorde con su proximidad al obstáculo más cercano (un valor más alto cuanto mayor sea la proximidad al obstáculo) hasta llegar a la celda origen.
- Trazar la trayectoria siguiendo el menor gradiente desde el origen hasta el destino.

Es decir, cuando se le propone a Wavefront un nuevo destino, el algoritmo asigna valores a todas las celdas en las que ha dividido el mapa dado. La celda destino recibe el valor 0, las celdas contiguas a la celda destino reciben el siguiente valor (el 1), las contiguas a estas últimas reciben el siguiente valor, y así, sucesivamente, se van dando valores a las celdas describiendo círculos concéntricos desde el destino, como un frente de ondas que se desplaza por el agua (de ahí el nombre). Una vez hecho esto, la ruta planificada es la que resulta de seguir el menor gradiente desde la celda origen hasta la celda destino. Para saber en que celda se encuentra necesita el localizador global AMCL.

En la Figura 3.3 se puede ver de que manera se rellenan las celdas del mapa, comenzando en el destino (celda 'G', *goal*), para obtener la trayectoria origen (celda 'S', *start*) – destino.

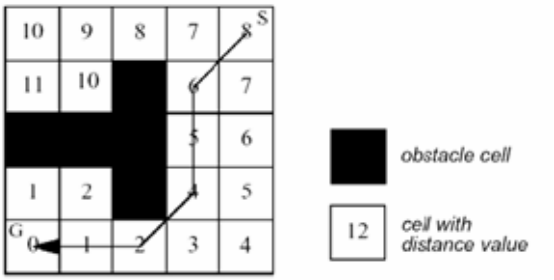


Figura 3. 3 – Ejemplo de planificación con Wavefront [Ocaña08]

En la Figura 3.4 se observa la representación en tres dimensiones del ejemplo que se ha presentado en la Figura 3.3. Para hacer esta representación se ha utilizado un entramado de más celdas de menor tamaño. En esta representación tridimensional se ve claramente que para trazar la trayectoria desde el origen hasta el destino basta con seguir el camino de menor gradiente. Es decir, la ruta trazada hasta el destino es la que seguiría una pelota soltada en el origen.

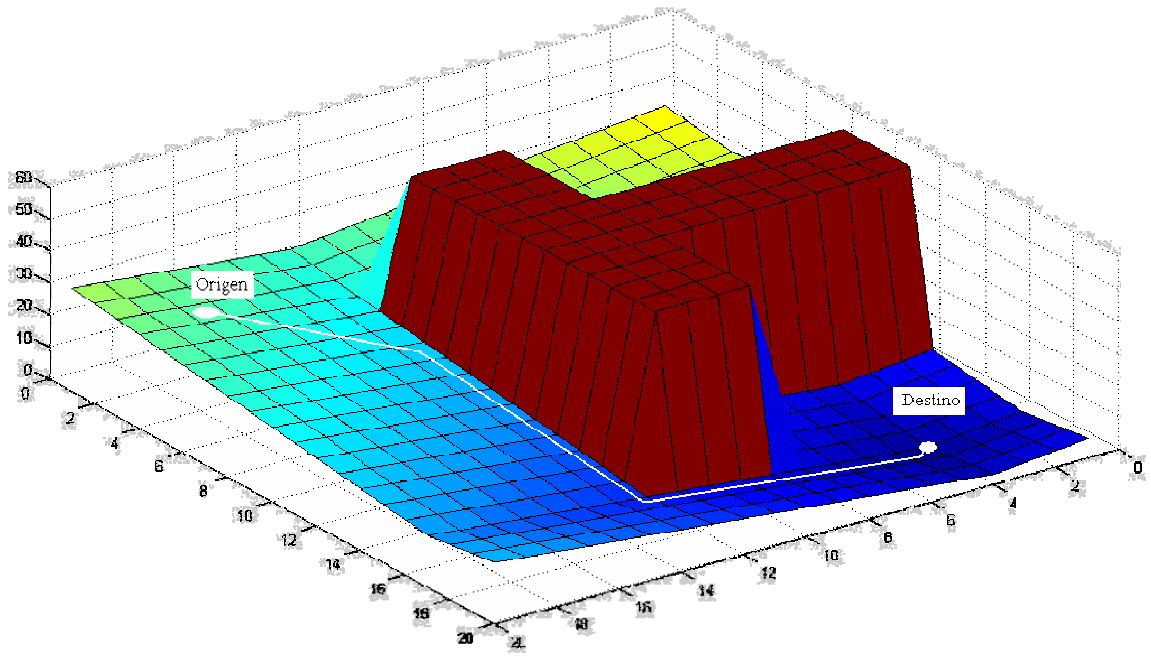


Figura 3. 4 - Representación en 3D de una planificación con Wavefront

Debido al funcionamiento que se ha explicado, el planificador Wavefront busca la ruta formada por el conjunto de líneas rectas más largas que no cruzas ningún obstáculo. Wavefront no genera la ruta completa, si no que genera una serie de puntos intermedios de ésta. Por este motivo es necesario un planificador local para la evasión de los posibles obstáculos móviles (obstáculos que no aparecen en el mapa del que parte Wavefront por no ser conocidos). El planificador local toma los puntos origen y final de esas líneas como sucesivos destinos para el robot. En este proyecto se va a usar como planificador de bajo nivel el algoritmo VFH (Vector Field Histogram) porque está muy estudiado y documentado y cuenta con su implementación en Player, pero podría usarse otro planificador local como por ejemplo ND (*Nearness Diagram*), también implementado en Player o un trazador de rutas de tipo *Spline*.

b) Descripción del driver

El driver del algoritmo Wavefront que incluye Player requiere tres dispositivos y provee uno:

- el mapa (*map*).

- dos dispositivos de posición (*position2d*): uno de entrada y otro de salida.

- Un dispositivo planificador (*planner*)

Requiere el uso del mapa, y de dos dispositivos de posición. El dispositivo de posición de entrada le da a Wavefront la posición en que se encuentra el robot. En este proyecto lo proporcionará AMCL. El dispositivo de posición de salida de Wavefront se le pasa al driver de VFH que, por otra parte, se encarga de la evasión de obstáculos.

A su vez, el dispositivo planificador (*planner*) que provee el driver, que proporciona los puntos intermedios de las rutas que genera el algoritmo. En la Figura3.5 se ve una ventana del simulador Stage en la que están repesentados los puntos intermedios que ha proporcionado el dispositivo *planner* durante una prueba de navegación.

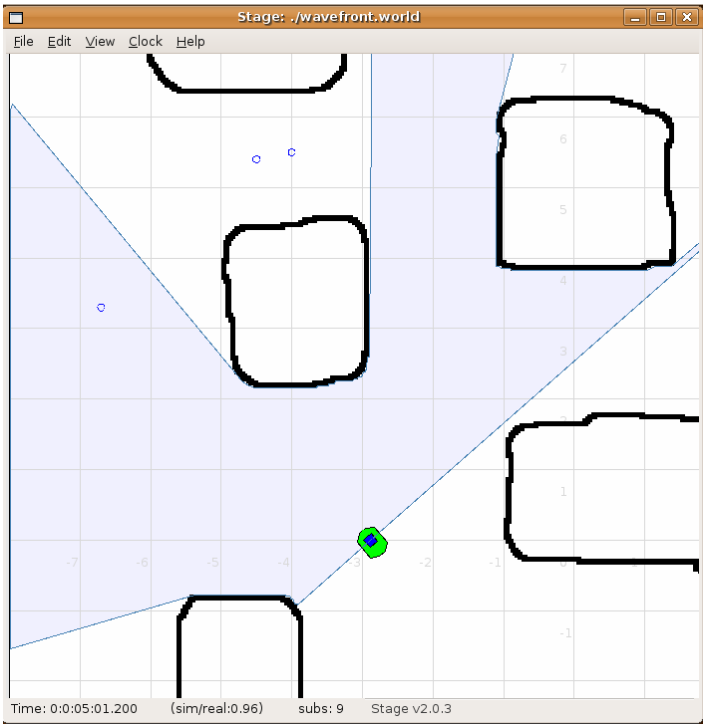


Figura 3. 5 – Ejemplo de puntos intermedios que genera Wavefront

Los dispositivos que se acaban de mencionar aparecen en el archivo de configuración de player (.cfg) de la siguiente manera:

```
driver
(
  name "wavefront"
  provides ["6665:planner:0"]
  requires ["output::6665:position2d:1" "input::6665:position2d:2" "6665:localize:0" "map:0"]
)
```

c) Descripción de los parámetros del driver

El driver de Wavefront en Player tiene, además, una serie de parámetros que se pueden fijar según las necesidades de cada aplicación. Estos parámetros se explican a continuación:

- *safety_dist*: mínima distancia a un obstáculo que se acepta en la planificación de una ruta. Este parámetro debe ser mayor o igual que el correspondiente del sistema de localización de bajo nivel. Valor por defecto: 0,25m.

- *max_radius*: a efectos de planificación, todas las celdas que estén al menos a la distancia marcada por esta cantidad de cualquier obstáculo son igual de buenas. Valor por defecto: 1,0m.

- *dist_penalty*: distancia que se añade a la ruta para que las esquinas de los obstáculos no se recorten al trazar dicha ruta. Valor por defecto: 1,0.

- *distance_epsilon*: distancia de error alrededor del destino aceptada para considerar que el robot ha llegado a éste. Este parámetro debe ser mayor que el correspondiente del sistema de localización de bajo nivel. Valor por defecto: 0,5m.

- *angle_epsilon*: ángulo diferencia entre la orientación del robot y el ángulo se le ha pedido a éste, que se considera aceptable para considerar que ha alcanzado la orientación correcta. Este parámetro debe ser mayor que el correspondiente del sistema de localización de bajo nivel. Por defecto: 10º.

- *replan_dist_thresh*: un cambio en la posición del robot igual o mayor que la distancia que representa este parámetro provoca un replanteamiento de la ruta. Fijar a -1 si no se desea hacer replanteamiento. La solución del replanteamiento es computacionalmente sencilla y puede ayudar mucho en entornos dinámicos. Nótese que no se hacen cambios en el mapa para replantear la ruta. Valor por defecto: 2,0 m.

- *replan_min_time*: tiempo en segundos que espera el planificador Wavefront entre replanteamientos de ruta. Fijar a -1 si no se desea hacer replanteamiento. Valor por defecto: 2,0 s.

- *cspace_file (filename)*: se usa este archivo para configurar el mapa. Cuando se inicia el planificador, si este archivo se puede leer y los datos incluidos en él concuerdan con el mapa actual que se le ha proporcionado al planificador, los datos del *c-space* (*Composition Space*) se leen desde dicho archivo. En caso contrario, los datos del *c-space* son calculados. *C-space* es una aplicación diseñada para facilitar el análisis gráfico y algebraico de datos que representan composiciones coordenadas. En cualquier caso, al parar el planificador, los datos del *c-space* se almacenarán en el archivo que indique este parámetro para su uso siguiente. Los cálculos del *c-space* pueden ser

costosos, por tanto, su almacenamiento nos pude ahorrar mucho tiempo. Por defecto: player.cspace.

- *add_rotational_waypoints*: si no es cero y entre las orientaciones de cada dos puntos intermedios contiguos hay una diferencia mayor a 45º, el planificador añade un punto intermedio entre ambos. Valor por defecto: 1.

3.2.2. VFH

En este apartado se describe el método del Histograma de Campo de Vectores (*Vector Field Histogram* – VFH), creado por Ulrich y Borenstein, que es un algoritmo muy conocido y utilizado para la generación de trayectorias y el replanteamiento de éstas, y también para la evasión de obstáculos mientras el robot se dirige a su destino. (Ver [Ulrich98])

a) Modo de funcionamiento

Este algoritmo se basa en el uso de un histograma cartesiano de dos dimensiones para representar obstáculos. En cada celda de este histograma hay un valor que representa la certeza de que exista un obstáculo en esa posición. Para ello, este algoritmo, al igual que el anterior, también necesita trazar una rejilla en el espacio que capta con su sistema sensor. A diferencia de Wavefront, este algoritmo no usa un mapa, por lo que la rejilla que traza es local, pero en ambos casos la rejilla es métrica.

El funcionamiento de este algoritmo puede resumirse en los siguientes pasos:

- El entorno local del robot se representa en un rejilla de dos dimensiones, en la que los valores de las celdas almacenan la probabilidad de contener un obstáculo, calculada a partir de las mediciones realizadas por el sistema sensor.

- Se reduce el histograma cartesiano a un histograma polar de una dimensión. De este modo se obtiene la probabilidad de encontrar un obstáculo en las direcciones de giro.

- Se buscan todas las rutas disponibles al destino, y se selecciona el que menor función de coste G tenga.

$$G = a \cdot TD + b \cdot WO + c \cdot PD$$

Siendo: TA (*target direction*): alineación del robot con respecto al objetivo

WO (*wheel orientation*): diferencia entre la nueva dirección y la orientación actual de las ruedas.

PD (*previous direction*): diferencia entre la dirección previamente seleccionada y la nueva dirección.

a, b, c: parámetros de ajuste del comportamiento del robot.

Es decir, se busca la dirección de avance mejor ponderando la información que se posee de la dirección actual y la dirección de destino.

VFH emplea una técnica de reducción de datos en dos fases, transformando así los datos en los tres niveles que se explican a continuación.

En el primer nivel se encuentra la "descripción detallada" del entorno del robot. En este nivel el histograma cartesiano de dos dimensiones se actualiza continuamente en tiempo real mediante la información proveniente de los sensores de a bordo del robot.

El nivel intermedio de datos lo construye un "histograma polar" de una dimensión alrededor de la posición istantánea del robot. El histograma contendrá una serie de sectores angulares (*k*) de una anchura determinada. La transformación del "entorno detallado", descrito en el primer nivel, a histograma mencionado, resulta de colocar en cada sector *k* un valor que represente la densidad polar de obstáculos en la dirección correspondiente a dicho sector angular.

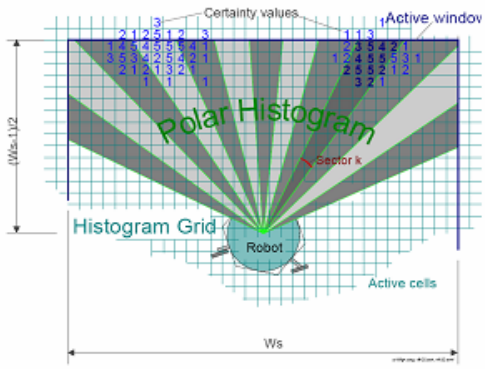


Figura 3. 6 – Transformación del primer nivel de datos a histograma polar en el algoritmo VFH [Borenstein91]

En la Figura 3.6 aparece representada la transformación de la descripción detallada del entorno en el histograma polar.

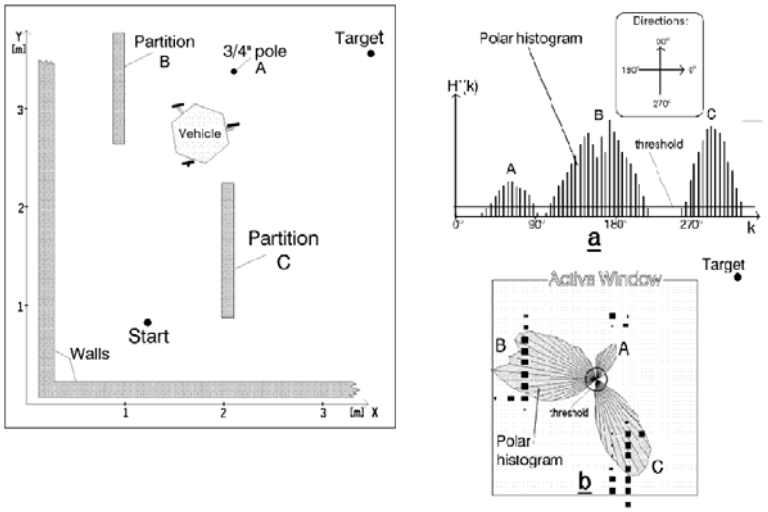


Figura 3. 7 - Nivel intermedio de datos en el algoritmo VFH [Borenstein91]

En la Figura 3.7 aparece representado el nivel intermedio de datos. Y en la Figura 3.8 aparece el diagrama que representa la segunda transformación de datos. En él, aparecen en color verdes los sectores por donde es seguro que el robot circule, y en rojo los que no lo son.

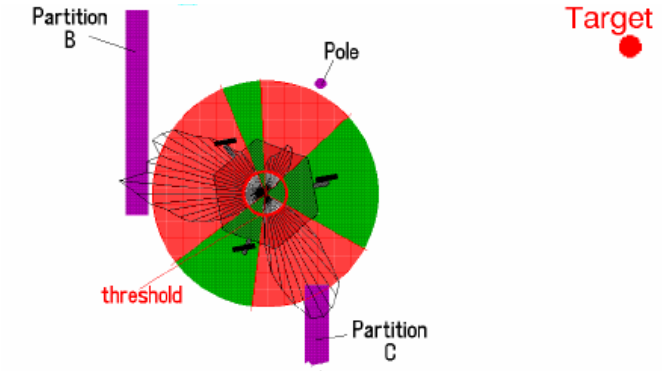


Figura 3. 8 - Diagrama de sectores seguros y peligrosos en el algoritmo VFH [Borenstein91]

El último nivel de la representación de datos es la salida del algoritmo VFH: los valores de referencia para la conducción del vehículo.

b) Descripción del driver

Este algoritmo está implementado en Player donde se comporta como driver que requiere dos dispositivos y provee uno:

- Requiere un dispositivo de posición (*position2d*).
- Y un dispositivo sensor, ya sea láser o de ultrasonidos (*laser* o *sonar*).
- Provee de otro dispositivo de posición (*position2d*).

El dispositivo de posición requerido es el de más bajo nivel, el del robot, que es comandado directamente por VFH (ver Figura 3.2). El driver de sensado que se utilice (láser o de ultrasonidos) servirá para trazar el mapa del entorno de la forma mencionada que permitirá la evitación de obstáculos.

El driver de VFH provee además de otro dispositivo de posición. En el caso de utilizar únicamente este algoritmo para la navegación local, sería este dispositivo el usado para fijar los destinos. Si se usa VFH como sistema de bajo nivel del planificador global Wavefront, este dispositivo de posición será el que esté comunicado con el dispositivo de posición de salida del driver de Wavefront, como se ha explicado en el subapartado anterior. (Ver Figura 3.2)

Los dispositivos que se acaban de mencionar aparecen en el archivo de configuración de player (.cfg) de la siguiente manera:

```
driver
(
  name "vfh"
  provides ["6665:position2d:1"]
  requires ["6665:position2d:0" "6665:laser:0"]
)
```

c) Descripción de los parámetros del driver

Se deben conocer una serie de parámetros asociados a este driver para adaptarlo a nuestras necesidades. Dichos parámetros se detallan a continuación:

- *cell_size*: tamaño de las celda del mapa local de ocupación. Valor por defecto: 0,1m.
- *window_diameter*: fija la dimensión del mapa de ocupación (el mapa consiste de *window_diameter* x *window_diameter* celdas). Valor por defecto: 61. Por tanto, las dimensiones del mapa serán por defecto 6,1 m x 6,1 m (61 x 61 x 0,1 m).
- *sector_angle*: resolución angular del histograma polar en grados. Valor por defecto: 5º.
- *safety_dist_0ms*: distancia mínima a la que se permite al robot acercarse a los obstáculos cuando está parado. Valor por defecto: 0,1m.
- *safety_dist_1ms*: distancia mínima a la que se permite al robot acercarse a los obstáculos cuando lleva una velocidad de 1m/s. Valor por defecto: *safety_dist_0ms*.
- *max_speed*: máxima velocidad permitida para el robot. Valor por defecto: 0,2m/s.
- *max_speed_narrow_opening*: máxima velocidad permitida al robot en espacios estrechos. Valor por defecto: *max_speed*.
- *max_speed_wide_opening*: máxima velocidad permitida al robot en espacios anchos. Valor por defecto: *max_speed*.
- *max_acceleration*: máxima aceleración permitida al robot. Valor por defecto: 0,2m/s^2.
- *min_turnate*: mínima velocidad de giro permitida al robot. Valor por defecto: 10º/s.
- *max_turnate_0ms*: máxima velocidad de giro permitida al robot cuando está parado. Valor por defecto: 40º/s.
- *max_turnate_1ms*: máxima velocidad de giro permitida al robot cuando va a una velocidad de 1m/s. Valor por defecto: *max_turnate_0ms*.

- *min_turn_radius_safety_factor*: este parámetro del driver aun no está explicado en la ayuda, por lo que su uso no queda claro en este proyecto. Valor por defecto: 1.0.

- *free_space_cutoff_0ms*: parámetro adimensional. Cuanto mayor es el valor de este parámetro más se aproxima el robot a los obstáculos antes de evitarlos, mientras esta parado. Valor por defecto: 2000000,0.

- *free_space_cutoff_1ms*: parámetro adimensional. Cuanto mayor es el valor de este parámetro más se aproxima el robot a los obstáculos antes de evitarlos, mientras va a una velocidad de 1m/s. Valor por defecto: *free_space_cutoff_0ms*.

- *obs_cutoff_0ms*: este parámetro del driver aun no está explicado en la ayuda, por lo que su uso no queda claro en este proyecto. Valor por defecto: *free_space_cutoff_0ms*.

- *obs_cutoff_1ms*: este parámetro del driver aun no está explicado en la ayuda, por lo que su uso no queda claro en este proyecto. Valor por defecto: *free_space_cutoff_0ms*.

- *weight_desired_dir*: error de ángulo permitido al robot para dirigirse al destino sin replantear el trayecto. Valor por defecto: 5,0.

- *weight_current_dir*: error de distancia permitido al robot para continuar dirigiéndose hacia el destino. Valor por defecto: 3,0.

- *distance_epsilon*: distancia de error alrededor del destino aceptada para considerar que el robot ha llegado a éste. Valor por defecto: 0,5m.

- *angle_epsilon*: ángulo diferencia entre la orientación del robot y el ángulo se le ha pedido a éste, que se considera aceptable para considerar que ha alcanzado la orientación correcta. Valor por defecto: 10º.

Este driver incluye unas opciones para el escape en caso de parada por choque. En caso de que el dispositivo de posición del robot devuelva una señal de *stall* (indicador de choque del robot), el algoritmo iniciaría un procedimiento de escape. Este procedimiento hace que el robot se mueva, durante un determinado tiempo, hacia delante o hacia atrás mientras gira. Los parámetros que controlan este procedimiento de escape son los siguientes:

- escape_speed: si no es cero, indica la velocidad que se usa mientras se intenta escapar. Valor por defecto: 0,0m/s.

- escape_time: si no es cero, indica el tiempo (en segundos) que durará el intento de escape. Valor por defecto: 0,0s.

- escape_max_turnate: si no es cero, representa la máxima velocidad angular utilizada mientras se intenta escapar. Valor por defecto: 0,0º/s.

3.2.3. AMCL

El algoritmo de Localización Adaptativo de Monte-Carlo (Adaptive Monte-Carlo Localization - AMCL) permite obtener la distribución probabilística de las posibles posiciones del robot utilizando un filtro de partículas (ver [Fox03a]). El filtro de partículas implementado en el driver AMCL es adaptativo porque el número de partículas se ajusta dinámicamente: cuando hay mucha incertidumbre sobre la posición del robot el número de partículas aumenta, y cuando se conoce bien su posición este número disminuye.

a) Modo de funcionamiento

Este algoritmo necesita el mapa del entorno en el que se desea localizar al robot para determinar la probabilidad de de ubicar al robot en cada punto. Cada posible ubicación es representada por una partícula, y la probabilidad de encontrar al robot en esa ubicación se representa con el peso de dicha partícula.

Inicialmente, el algoritmo arranca considerando que el robot puede estar en cualquier punto del mapa. Por este motivo, las partículas se reparten a lo largo de la superficie de éste.

En la posterior etapa de observación o corrección se utilizan los sensores de observación del robot para reconocer su entorno cercano y, con esto, aumentar la probabilidad de ciertas hipótesis, o lo que es lo mismo, aumentar el peso de la partícula que las representa, y disminuir la probabilidad de otras. Esta etapa se basa por tanto en el modelo de observación.

Después, cuando el robot realiza un movimiento, el filtro pasa a la etapa de actuación, en la que se actualiza el valor de las partículas en función del movimiento del robot. Esta etapa está basada por tanto en el modelo de actuación.

En la Figura 3.8 se muestran las fases de la Localización de Monte-Carlo [López08]:

- 3.8.a) Inicialización. Partículas aleatorias de la función de probabilidad de posición (creencia) con el mismo peso.
- 3.8.b) Observación de una puerta. Reponderación de los pesos.
- 3.8.c) Movimiento. Propagación de las partículas.
- 3.8.d) Observación de una puerta. Reponderación de los pesos.
- 3.8.e) Movimiento. Propagación de las partículas.

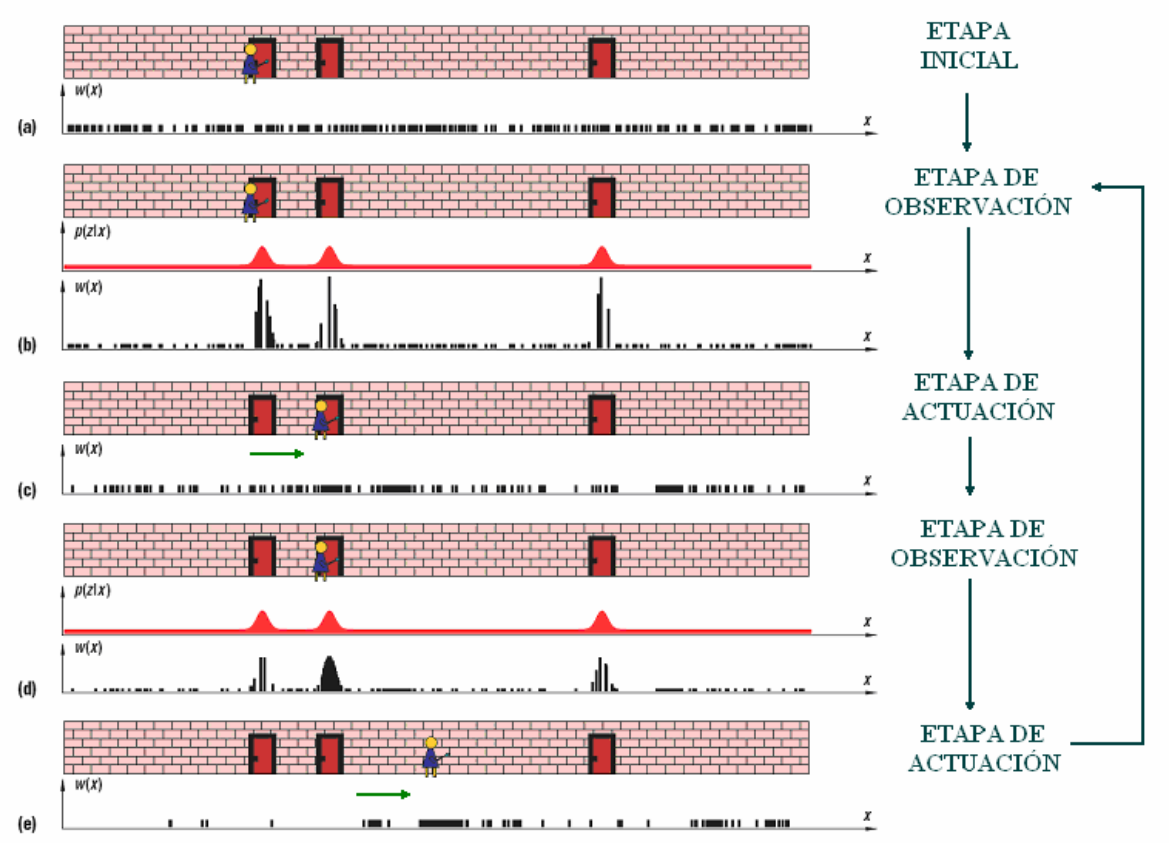


Figura 3. 9 - Ejemplo de funcionamiento de la Localización de Monte-Carlo [Fox03b]

b) Descripción del driver

A partir de lo expuesto anteriormente, el driver de Player que implementa el algoritmo AMCL requiere el uso de los siguientes dispositivos:

- un dispositivo de posición (*position2d*) que se corresponda con el de entrada de odometría del robot, es el que proporciona el modelo de actuación.
- un dispositivo láser (*laser*), es el que propone el modelo de observación.
- el dispositivo mapa (*map*).

A su vez, el driver de AMCL devuelve una serie de dispositivos:

- un dispositivo de localización (*localize*), que proporciona las partículas más probables en las que está el robot con su correspondiente peso.
- un dispositivo de posición (*position2d*) que proporciona el punto más probable en el que se encuentra el robot.

Al usar AMCL como localizador global, este dispositivo de posición que provee es el que se le pasa al driver de Wavefront como entrada. (Ver Figura 3.1)

Los dispositivos que se acaban de mencionar aparecen en el archivo de configuración de player (.cfg) de la siguiente manera:

```
driver
(
  name "amcl"
  provides ["6665:position2d:2" "6665:localize:0"]
  requires ["odometry::6665:position2d:1" "6665:laser:0" "laser::6665:map:0"]
)
```

c) Descripción de los parámetros del driver

Para adaptar el driver de AMCL a las necesidades de cada aplicación basta con modificar los parámetros de éste, que se detallan a continuación.

El primer bloque de parámetros se refiere a características del filtro de partículas:

- *odom_init*: indica si se usa el dispositivo de odometría como sensor de acción (a 1). Si no se usa se pone a 0. Este parámetro está relacionado con el parámetro update_thresh. Valor por defecto: 1.

- *pf_min_samples*: límite inferior del número de muestras a mantener en el filtro de partículas. Valor por defecto: 100.

- *pf_max_samples*: límite superior del número de muestras a mantener en el filtro de partículas. Valor por defecto: 10000.

Los dos siguientes parámetros, *pf_error* y *pf_z*, tienen que ver con que este algoritmo sea adaptativo.

- *pf_err*: parámetro de control para fijar el tamaño del set de partículas. En concreto es el máximo error permitido entre la distribución real y la distribución recortada. Valor por defecto: 0,01.

- *pf_z*: parámetro de control para fijar el tamaño de las partículas. En concreto, este parámetro limita el error dado por *pf_err*. Valor por defecto: 3,0.

- *init_pose*: si no se conoce la posición inicial estimada para el robot, se toma el valor de este parámetro. Por defecto: [0 0 0] (m m rad).

- *init_pose_var*: incertidumbre en la estimación de la posición inicial. Por defecto: [1 1 2π] (m m rad).

- *update_thresh*: mínimo cambio exigido en el sensor de acción (dispositivo de odometría) para forzar una actualización del filtro de partículas. Por defecto: [0,2 π/6] (m rad).

- *odom_drift[0-2]*: fija las tres filas de la matriz de covarianza. Por defecto:

  · *odom_drift[0]* [0,2 0,0 0,0]

  · *odom_drift[1]* [0,0 0,2 0,0]

  · *odom_drift[2]* [0,2 0,0 0,2]

El siguiente conjunto de parámetros se refiere a las características del láser:

- *laser_pose*: posición del sensor láser referido al sistema de coordenadas del robot. Por defecto: [0 0 0].

- *laser_max_beams*: máximo número de lecturas que se usan, de entre el conjunto de éstas. Valor por defecto: 6.

- *laser_range_max*: máximo rango de las lecturas devueltas por el láser. Valor por defecto: 8,192m.

- *laser_range_var*: resolución de las lecturas devueltas por el láser. Valor por defecto: 0,1m.

- *laser_range_bad*: este parámetro del driver aun no está explicado en la ayuda, pero por las pruebas realizadas durante este proyecto, se cree que es el error de medida que comete el láser. Valor por defecto: 0,1.

3.2.4. Detalle de los parámetros de los drivers que se han modificado

Una vez descritos los algoritmos utilizados y sus correspondientes drivers en Player, se detallan a continuación los parámetros específicos que se han modificado de cada driver para adaptarlos a las necesidades de las aplicaciones desarrolladas durante este proyecto.

En el driver del algoritmo Wavefront se han cambiado los valores de los parámetros siguientes:

- *safety_dist* = 0.15 m

- *distance_epsilon* = 0.3 m

- *angle_epsilon* = 20º

De los parámetros anteriores, *safety_dist* se ha disminuido para permitir que el trazado de las rutas más próximo a los obstáculos. El parámetro *distance_epsilon* también se ha disminuido para que la llegada del robot a su destino sea más precisa. Y *angle_epsilon* se ha aumentado para permitir al robot un error un poco superior en cuanto a su orientación.

Para el driver del algoritmo VFH se han cambiado los valores de los parámetros siguientes:

- *safety_dist* = 0.15 m
- *distance_epsilon* = 0.2 m

De los parámetros cambiados en el driver de VFH se ha aumentado el parámetro *safety_dist* para que se realice la evasión de obstáculos antes de que el robot se aproxime demasiado a éstos. Y se ha disminuido el valor del parámetro *distance_epsilon* para que la llegada del robot a su destino sea más precisa.

Y en el driver del algoritmo AMCL se han cambiado los valores de los siguientes parámetros:

- *init_pose_var* = [4m 4m 2π]
- *laser_range_max*: cuando se trabaje con el modelo del robot en Stage fijaremos este parámetro a 8 m. Si trabajamos con el robot real y el láser Hokuyo fijaremos este parámetro en 4 m.

Se ha aumentado el valor del parámetro *init_pose_var* para que el algoritmo tenga en cuenta un conjunto más amplio de posiciones en las que puede estar el robot al iniciar la aplicación. El valor del parámetro *laser_range_max* hay que cambiarlo dependiendo de si trabajamos con el láser implementado en el modelo del robot en Stage, o con el láser Hokuyo, porque el alcance máximo de cada uno de ellos es diferente.

CAPÍTULO 4

COMPORTAMIENTOS DESARROLLADOS PARA EL ROBOT

En este capítulo se pasa a explicar en detalle los comportamientos que se han creado para el robot, y que se mencionaron en la introducción.

Cuando se arranca la aplicación programada, como se explica en el manual de usuario, se abren dos ventanas, una para seleccionar las diferentes opciones de comportamientos desarrollados en este proyecto, y otra que contiene el mapa del espacio inteligente, que es el entorno donde se realizan las pruebas. Se pueden ver dichas ventanas en la Figura 4.1.

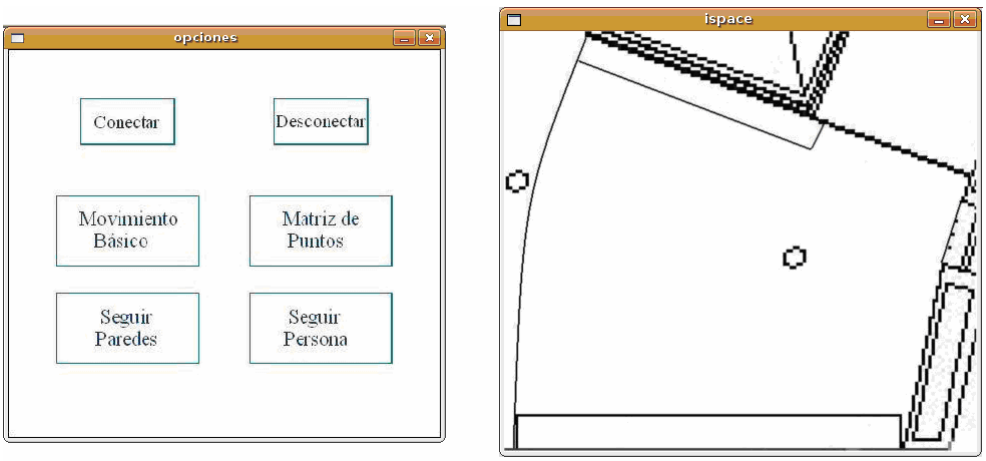


Figura 4. 1 - Ventanas que se abren al ejecutar la aplicación programada en este proyecto

En la ventana de opciones aparecen varios botones: el de “Conectar” crea un cliente y lo suscribe a los dispositivos a utilizar en los comportamientos; el botón “Desconectar” retira la suscripción a dichos dispositivos y destruye la conexión cliente-servidor; los otros cuatro botones se corresponden con los otros tantos comportamientos que se han desarrollado.

Los mencionados comportamientos para el robot son:

1. Movimiento básico.
2. Recorrer un conjunto de puntos para, por ejemplo, hacer que el robot realiza un conjunto de acciones en diferentes lugares del entorno.
3. Seguir las paredes de la estancia para, por ejemplo, realizar un reconocimiento del mapa del entorno.
4. Seguir a una persona u objeto móvil para una posible aplicación de ayuda personal.

El funcionamiento general de la aplicación desarrollada se resume en el flujograma representado en la Figura 4.2.

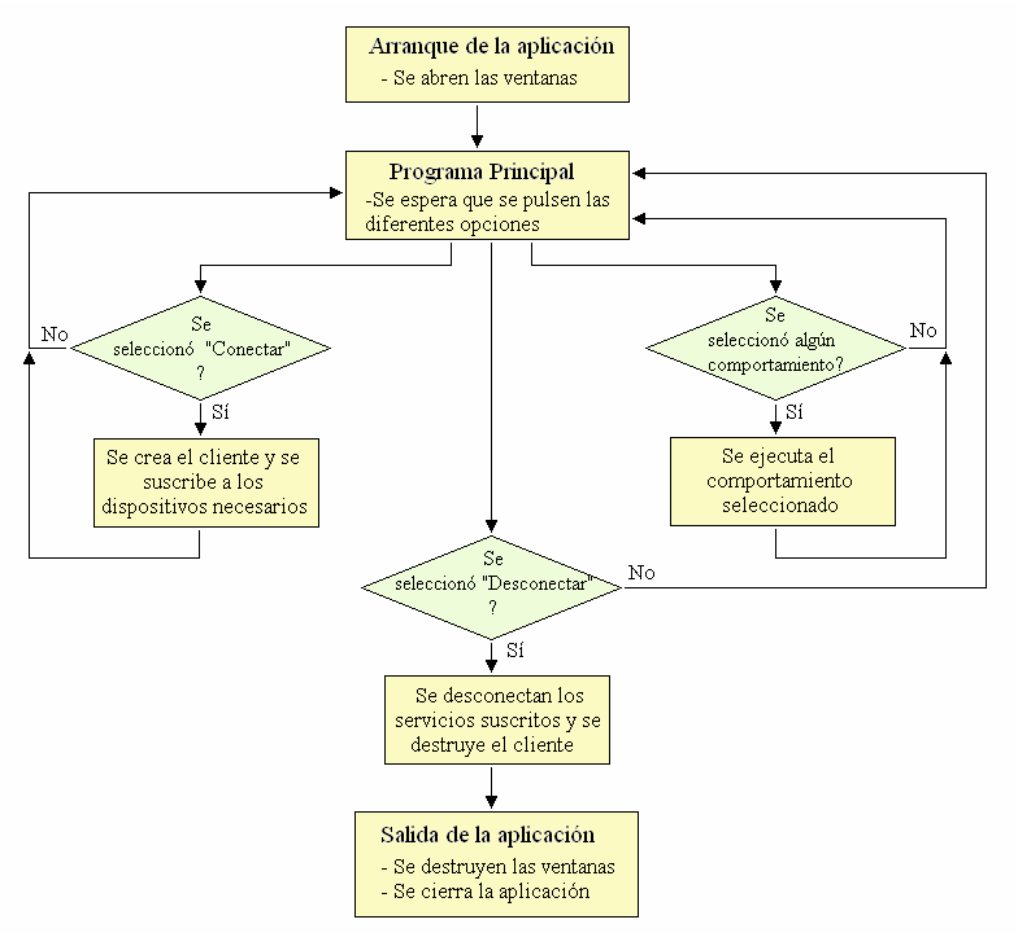


Figura 4. 2 - Flujograma del programa principal

Para que la aplicación desarrollada en este proyecto funcione correctamente, hay que seguir las instrucciones que aparecen en el manual de usuario. Si no se hace así se producen errores que traen como consecuencia la finalización del programa.

Por tanto, para un correcto funcionamiento de la aplicación, hay que pulsar la opción "Conectar" antes de seleccionar cualquier comportamiento, de no ser así se produce un error que conlleva la finalización del programa. De igual manera, no se puede pulsar la opción "Desconectar" si no se ha seleccionado antes la opción "Conectar" porque se produce un error que también hace que se cierre la aplicación.

En el caso de que, al intentar conectar el cliente al servidor ocurriera un error, habría que salir de la aplicación y volver a entrar iniciando de nuevo el proceso de conexión.

El funcionamiento de los comportamientos desarrollados se detalla en los apartados siguientes.

4.1. Movimiento Básico

Este comportamiento consiste en hacer que el robot se desplace desde el punto donde se encuentra (origen), hasta el punto que se le marca como destino en al mapa, utilizando la trayectoria óptima y evitando los posibles obstáculos que se encuentre en su camino. Para que el robot desarrolle este comportamiento se hace uso del algoritmo Wavefront de planificación global, pues se desea moverlo por un entorno conocido.

En la Figura 4.3 se muestra el funcionamiento del comportamiento descrito mediante un flujograma.

Al seleccionar el botón de este comportamiento el programa principal (Figura 4.2) llama a la función que lo controla.

Esta función lo primero que hace es leer la información que proporciona el cliente referente a los dispositivos de posición (*position2d*). Para ello se usa la siguiente función, incluida en las librerías de player:

```
player_client_read (playerc_client_t *cliente);
```

Después de obtener la información, manda un mensaje por la salida estándar (pantalla) informando de las posiciones relativa, dada por la odometría, y absoluta, dada por el localizador AMCL, del robot. Se pide al usuario por la pantalla que seleccione en la ventana del mapa el punto al que quiere llevar al robot, pinchando sobre él, y que seleccione la orientación final deseada para el robot de la misma manera, esto se hace

pinchando sobre las flechas mostradas en la Figura 4.4, que representan el ángulo de llegada del robot.

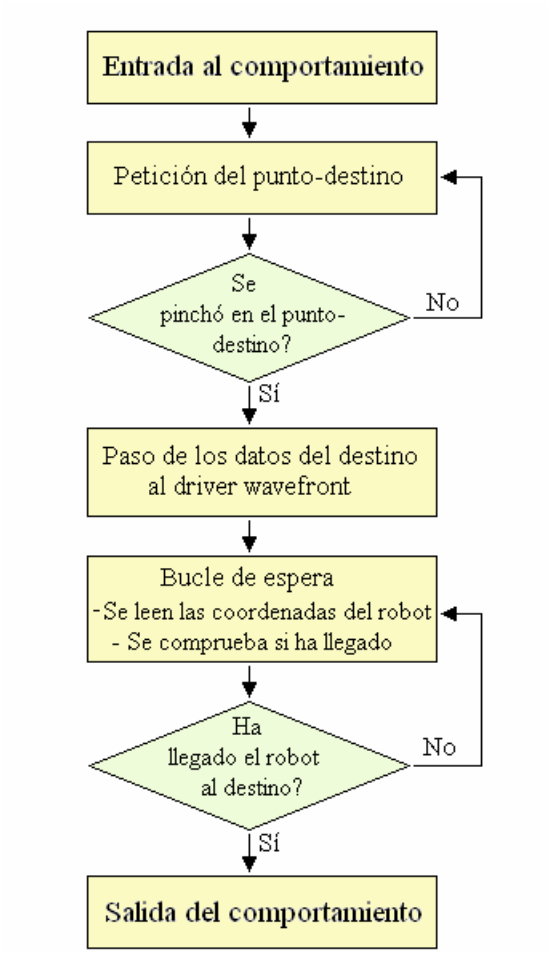


Figura 4. 3 – Flujograma que muestra el funcionamiento del comportamiento "Movimiento Básico"

En la Figura 4.5 se muestra el mensaje que aparece por pantalla informando de la posición local (odometría) y global (AMCL) del robot, y pidiendo que se introduzcan el destino y la orientación.

Figura 4. 4 - Flechas incluidas en la ventana del comportamiento "Movimiento Básico" que se utilizan para fijar la orientación del robot en el destino

Una vez se ha introducido el punto destino y la orientación hay que pasarle esos datos al driver Wavefront, más en concreto al dispositivo *planner* creado al efecto. Para hacer esto se utiliza la función:

```
playerc_planner_set_cmd_pose(playerc_planner_t *dispositivo, double x, double y, double theta);
```

Una vez introducido el destino, el programa entra en un bucle en el que comprueba continuamente la posición del robot hasta que llega al destino. Dicha comprobación se hace calculando la distancia euclidea desde donde se encuentra el robot hasta el destino fijado. Para este cálculo se utilizan las coordenadas que proporciona el dispositivo de posición de salida de AMCL. El bucle es como sigue:

```
while (sqrt (pow (position2d_amcl->px – x_destino,2.0) + pow (position2d_amcl->py – y_destino,2.0)) > error_destino)
{
   playerc_client_read(cliente);
}
```

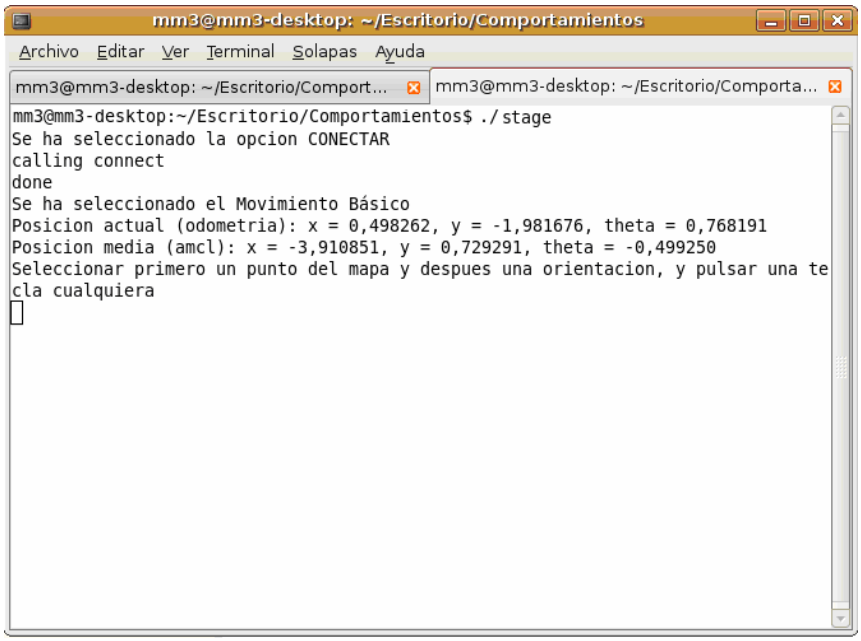


Figura 4. 5 - Mensaje que muestra por pantalla al iniciar el comportamiento "Movimiento Básico"

La función incluida en el bucle, como se ha mencionado antes, sirve para obtener la información de los dispositivos de posición. En este caso es necesaria para actualizar la condición de salida del bucle.

Una vez el robot llega a una distancia suficientemente cercana al destino (aceptable como error), el programa sale de la función de este comportamiento y vuelve al hilo principal. Se acepta un margen de error representado por el parámetro "error_destino".

El recorrido que hace el robot hasta llegar al destino se va dibujando punto a punto en la ventana que contiene el mapa, y en la ventana de Stage, si es que estamos trabajando con el

simulador. Los puntos dibujados en la primera ventana, su ensayo, sólo se ven cuando acaba el comportamiento, ya que mientras éste se desarrolla la ventana queda inactiva.

4.2. Recorrido de un Conjunto de Puntos

Con este comportamiento se pretende que el robot se mueva desde el origen a través de una serie de puntos del mapa que se le marquen en la ventana que se abre al seleccionarlo. Al igual que en el comportamiento anterior, se espera que el robot realice el recorrido marcado sin chocarse con posibles obstáculos móviles o estáticos que no aparezcan en el mapa y esten a lo largo de la trayectoria. Para realizar dichos desplazamientos se utiliza Wavefront para la planificación de trayectorias, junto con VFH que realiza la evasión de obstáculos, y AMCL que hace la localización global.

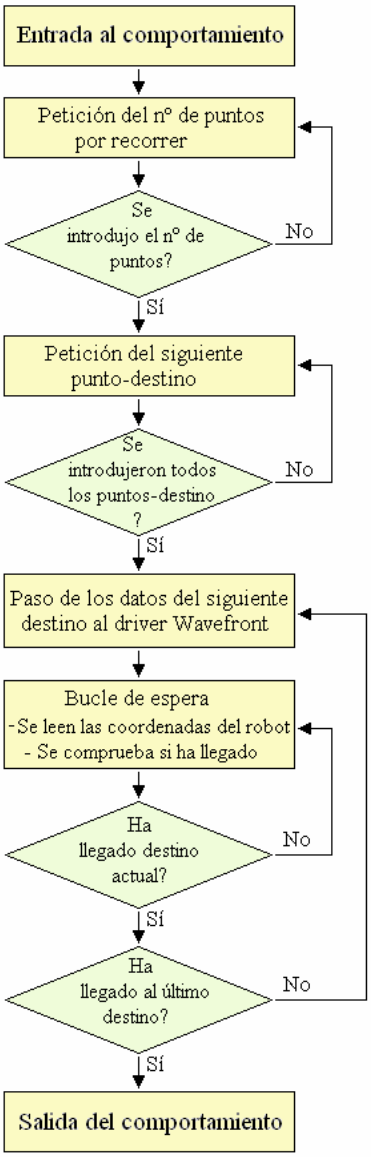


Figura 4. 6 - Flujograma que muestra el funcionamiento del "Recorrido de un Conjunto Puntos"

En la Figura 4.6 se muestra el flujograma que resume el funcionamiento de este recorrido de puntos seleccionados del mapa.

Tras seleccionar esta opción en la ventana de comportamientos, lo primero que hace la función que controla este comportamiento es pedir por pantalla que pinchemos en la ventana del mapa los puntos donde se quiere llevar al robot, que se van almacenando en una matriz. Una vez seleccionados todos los puntos se debe pulsar el botón que aparece en la Figura 4.7, "Fin de la selección de puntos". En este caso no se pide orientación final del robot, ya que ésta se hallará matemáticamente como la pendiente de la recta que lleva del punto origen al de destino de cada trayecto.

Figura 4. 7 - Botón incluido en la ventana del comportamiento "Recorrido de un Conjunto de Puntos", que sirve para indicar que se han introducido todos los puntos deseados

En la Figura 4.8 se muestra el mensaje que aparece por pantalla cuando iniciamos el éste comportamiento.

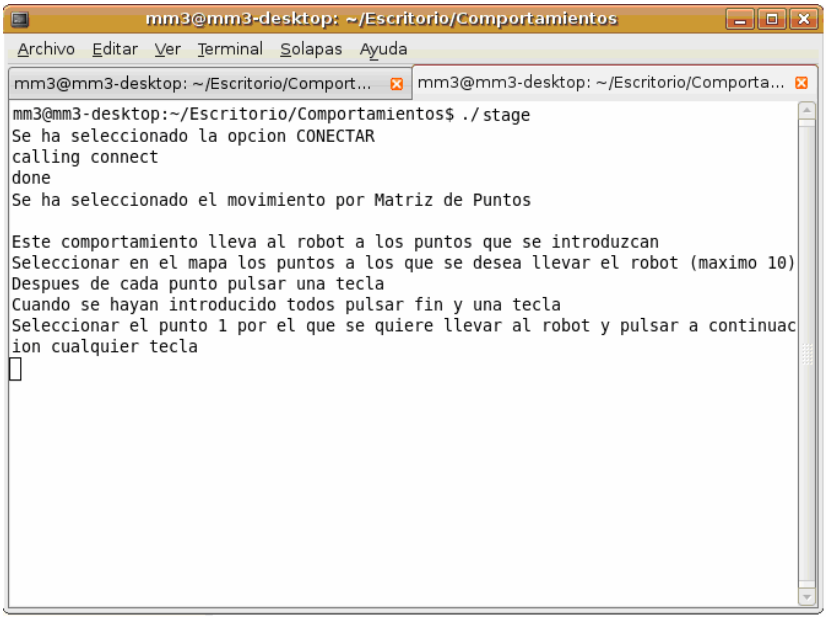


Figura 4. 8 - Mensaje que aparece por pantalla cuando se inicia el comportamiento "Recorrido de un Conjunto de Puntos"

Almacenados los puntos destino deseados para el comportamiento, se entra en un bucle en el que se le hacen llegar, sucesivamente, los destinos seleccionados al driver Wavefront. Las coordenadas de estos destinos se le hacen llegar de la misma manera que se hace en el comportamiento anterior, a través de la función "*playerc_planner_set_cmd_pose*" (ver apartado 4.1).

Después de pasar un destino, el programa espera en un bucle igual que el descrito para el comportamiento "Movimiento Básico". Al igual que entonces, se sale de ese bucle

cuando se ha llegado a una distancia igual o inferior a la fijada por el parámetro "error_destino". El recorrido del robot se dibuja también como en el comportamiento anterior.

Después de alcanzado un destino, se pasa al siguiente, y así sucesivamente hasta llegar al último destino.

Si se pulsa cualquier tecla mientras se desarrolla el comportamiento, o una vez alcanzado el último destino, se devuelve el control al programa principal para que se pueda desconectar el robot o seleccionar otro comportamiento.

4.3. Seguimiento de las Paredes del Entorno

Este comportamiento sirve para hacer que el robot recorra los límites de la estancia o entorno en el que se encuentra, dando una vuelta completa a lo largo de sus paredes.

En la Figura 4.9 aparece el flujograma que resume el funcionamiento de este comportamiento.

Este comportamiento puede resultar útil para la reconstrucción del mapa con el que trabajamos.A diferencia de los demás comportamientos, éste no está basado en la planificación global, sino en la planificación local, puesto que no se le proporciona un mapa, sino que se supone que se desea conocerlo.

Una vez seleccionado este comportamiento en la ventana de opciones (ver Figura 4.1), se entra en la función que lo controla. Dicha función está formada por varios bucles que a continuación se explican.

El primer bucle es de observación, y lo que hace es leer los datos que proporciona el sensor láser. Esto se hace utilizando la función siguiente:

```
playerc_laser_get_geom(playerc_laser_t *dispositivo);
```

Con esa función el dispositivo devuelve una matriz de coordenadas con los puntos donde choca el haz de luz del sensor con la pared, normalmente son las coordenadas de 361 puntos (Nl) distribuidos a lo largo del ángulo de barrido del láser (180º).

Este bucle de observación, recorre la matriz de este sensor y halla la distancia euclidea menor desde cada punto del barrido del láser al robot, mediante la siguiente ecuación:

$$D_i = \sqrt{Xl_i^{\,2} + Yl_i^{\,2}}$$

Siendo: D_i = Distancia euclidea desde el punto de choque *i* hasta el robot.

Xl_i = Coordenada X del punto *i*.

Yl_i = Coordenada Y del punto *i*.

i = índice para recorrer la matriz.

Se guarda la posición (Lm) que ocupan dentro de la matriz las coordenadas correspondientes a la distancia menor, que se corresponden con el punto de choque del láser más cercano al robot.

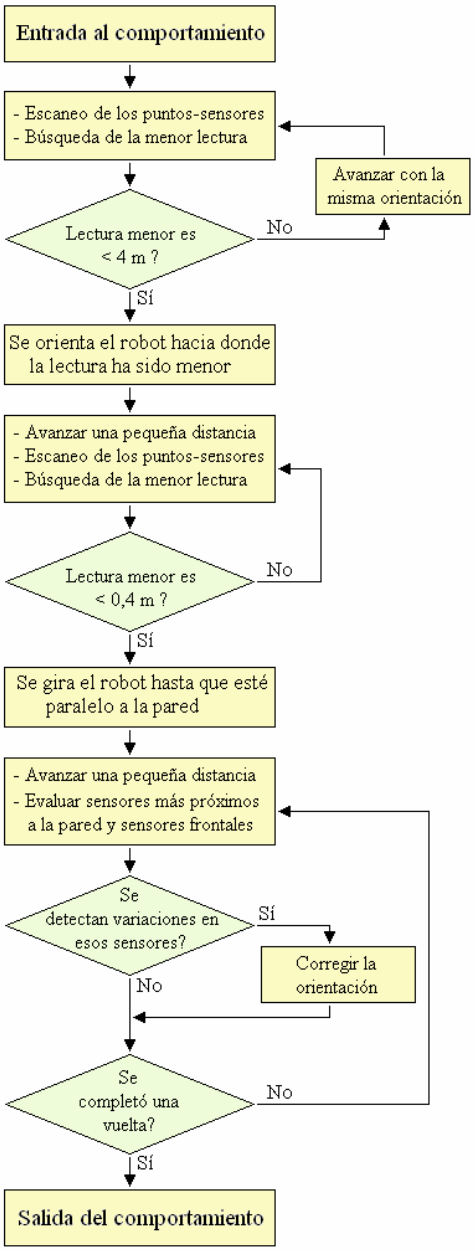


Figura 4. 9 - Flujograma que muestra el funcionamiento del "Seguimiento de las Paredes de una Estancia"

Si la lectura menor que se ha encontrado es inferior al rango máximo del láser que se utiliza, 8m para el láser implementado en Stage, y 4m si se trabaja con el láser Hokuyo, se entra en el bucle de orientación. Este bucle consiste en orientar el robot hacia la dirección que nos ha dado la lectura menor, pues esa será la dirección de la pared más cercana. Para ello, se usa el índice de la matriz que se ha almacenado, y se calcula el ángulo, α, al que se ha hecho esa lectura respecto del origen del robot como:

$$\alpha = \frac{\pi \cdot Lm}{Nl} - \frac{\pi}{2}$$

Siendo: Lm índice de la lectura menor

Nl número de lecturas del láser

La nueva orientación que hay que fijar para el robot es, por tanto, la orientación anterior más el ángulo α.

En la Figura 4.10 se muestra un ejemplo del cálculo del ángulo α.

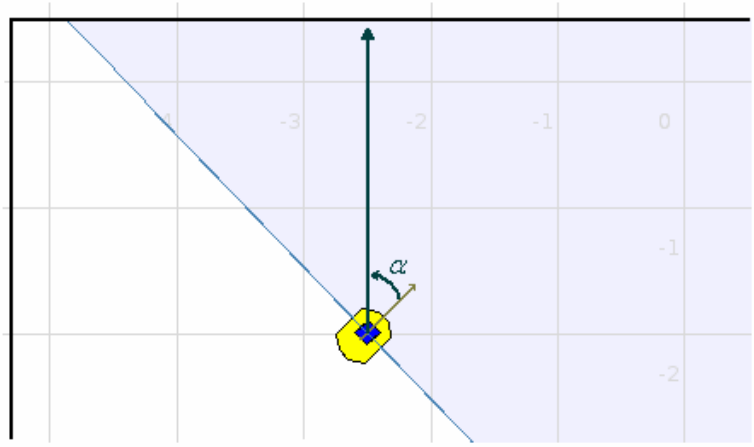


Figura 4. 10 - Ejemplo del cálculo del ángulo α para el comportamiento "Seguimiento de Paredes"

Si la lectura menor del láser es igual al rango máximo de éste (8m para láser de Stage y 4m para el Hokuyo), quiere decir que no hay ningún objeto que devuelva el reflejo del láser. En ese caso, se hace un movimiento de avance que consiste en hacer desplazarse al robot una distancia fijada por el parámetro "dist_avance" manteniendo la misma orientación. Mientras se hace el movimiento de avance se sigue entrando periódicamente al bucle de observación para evaluar las lecturas del láser, hasta que se devuelve una lectura menor que el rango de éste, momento en el que se pasa al bucle de orientación que se ha explicado en el párrafo anterior.

Una vez el robot está orientado en dirección a la pared más próxima, entra en un bucle de aproximación en el que se le hace avanzar hacia ella, sin dejar de contrastar continuamente los datos proporcionados por el dispositivo láser y de calcular la menor lectura. Cuando la lectura más pequeña sea igual o inferior a una distancia de

aproximación fijada por el parámetro "dist_aprox", se inicia el proceso de poner el robot paralelo a la pared. Este proceso consiste en calcular la orientación, β, que debe tener el robot para estar situado en dirección paralela a la pared. Se calcula β en función del punto-sensor que dé la lectura más pequeña. Para calcular β necesitaremos calcular de nuevo α de la misma manera que hemos hecho anteriormente.

Si el ángulo α obtenido es menor de cero, la pared se encuentra a la derecha del robot, y el ángulo β de paralelización se calcula de la siguiente manera:

$$\beta = \alpha + \frac{\pi}{2}$$

En caso contrario, la pared se encuentra a la izquierda del robot, y entonces el parámetro β se calcula de la forma siguiente:

$$\beta = \alpha - \frac{\pi}{2}$$

En la Figura 4.11 se muestra un ejemplo del valor de β en cada uno de los casos analizados.

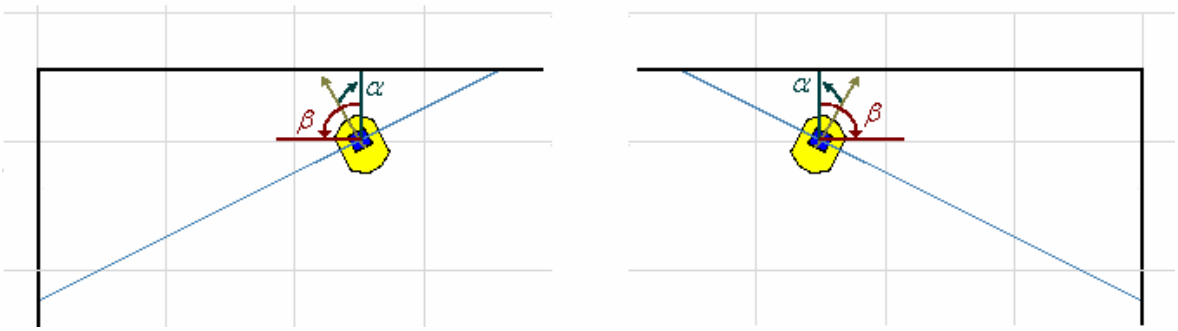


Figura 4. 11 - Ejemplo de cálculo de β dependiendo de en que lado del robot se encuentre la pared

Una vez orientado el robot con ángulo β, se hace que avance a lo largo de ella, manteniendo cierta distancia con la misma, y, para ello, se controlan continuamente las lecturas de los puntos-sensores frontales y los situados más cerca de la pared seguida. Si se detectan variaciones en estas lecturas se corrige la orientación del robot.

Por ejemplo, si se detecta que la distancia lateral del robot a la pared aumenta por encima de la fijada por el parámetro "dist_lateral_máx", o disminuye por debajo de la fijada por el parámetro "dist_lateral_min", nos indicará que el robot se está alejando o acercando, respectivamente, de la pared. En este caso, se inicia una maniobra de reorientación del robot. Esta maniobra consiste en sumar o restar a la orientación actual del robot un pequeño ángulo fijado por el parámetro "angulo_reorientacion". Dependiendo de si éste se está acercando o alejando de la pared, y, también, de si la pared está a la derecha o a la izquierda del robot, hay que sumar o restar ese ángulo a la orientación actual. En la Figura 4.12 aparecen ilustradas las posibles combinaciones que se pueden presentar.

A continuación se detallan los casos posibles de la maniobra de reorientación y la acción a emprender en cada uno de ellos:

- Si la pared está a la derecha y el robot se acerca a ésta → sumamos un pequeño ángulo fijado por el parámetro "angulo_reorientacion" a su orientación anterior.

- Si la pared está a la derecha y el robot se separa de ella → restamos un pequeño ángulo fijado por el parámetro "angulo_reorientacion" a su orientación anterior.

- Si la pared está a la izquierda y el robot se acerca a ésta → restamos un pequeño ángulo fijado por el parámetro "angulo_reorientacion" a su orientación anterior.

- Si la pared está a la izquierda y el robot se separa de ella → sumamos un pequeño fijado por el parámetro "angulo_reorientacion" ángulo a su orientación anterior.

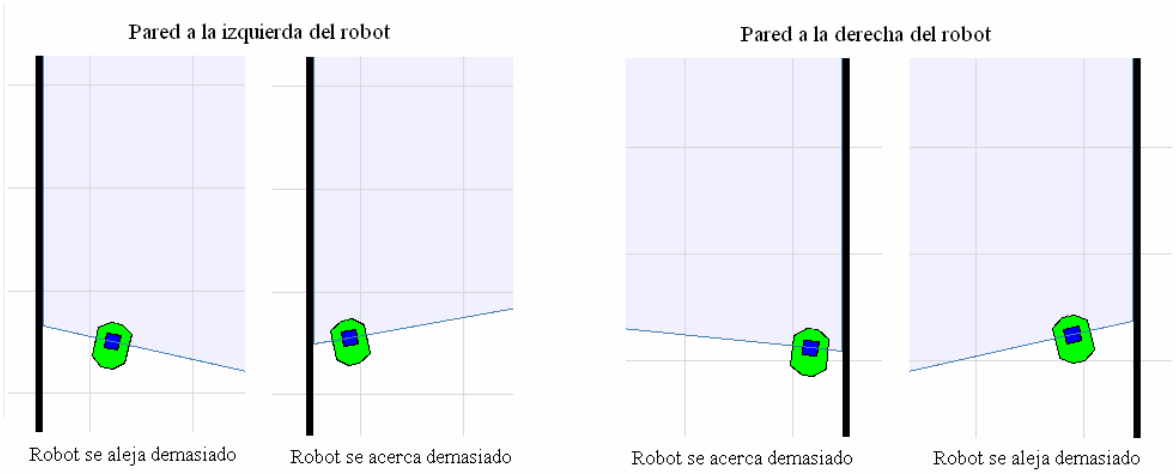


Figura 4. 12 - Ejemplo de posibles alteraciones de la ruta del robot en el comportamiento "Seguimiento de Paredes"

Finalmente, una lectura de una distancia frontal inferior a la fijada en el parámetro "dist_frontal_min", es señal de que el robot se aproxima a una esquina interior. En ese caso se le hará girar 90 grados para esquivarla.

Si en los sensores laterales se detecta una distancia superior a la fijada por el parámetro "dist_esquina_ext", significa que el robot tiene que trazar una esquina exterior, y se le hace girar 90 grados para salvarla.

En los casos de que las esquinas no sean a noventa grados, como sucede en algunas de las esquinas del entorno con el que se trabaja, el giro se realiza igualmente a 90 grados, y posteriormente se corrige la diferencia de orientación del robot siguiendo el procedimiento que se ha explicado en el párrafo anterior. En la Figura 4.13 se muestra un ejemplo de detección de esquinas.

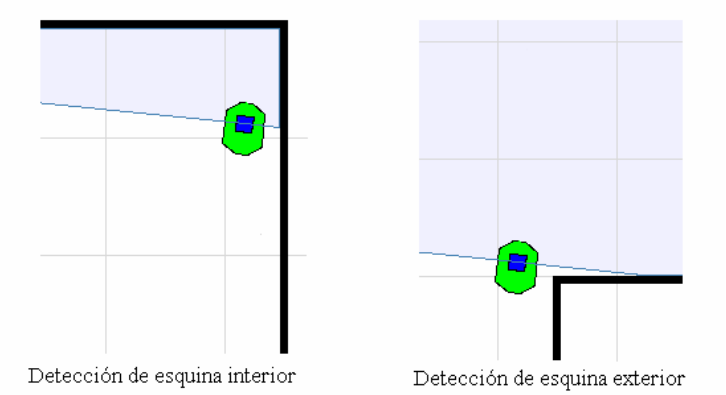


Figura 4. 13 - Ejemplos de detección de esquinas en el comportamiento "Seguimiento de Paredes"

De esta manera, el robot seguirá avanzando y corrigiendo su orientación a lo largo de las paredes.

Este comportamiento sigue hasta que el robot llegue a una distancia del punto desde el que inició el recorrido igual o inferior a la fijada por el parámetro "error_destino", completando así una vuelta a la estancia donde se encuentra. Cuando eso se cumpla, o cuando el usuario pulse cualquier tecla, se sale de está función y se le devuelve el control al programa principal.

Como en este comportamiento se trabaja con planificación local, podría suceder que, por errores de odometría, el robot no llegue nunca a las proximidades del lugar donde comenzó a seguir las paredes. Si esto sucediera, el robot seguiría recorriendo las paredes del entorno contínuamente.

4.4. Seguimiento de una Persona u Objeto Móvil

Este comportamiento consiste en hacer que el robot siga a una persona u objeto móvil. Esto puede resultar interesante a la hora de implementar una aplicación robótica de ayuda personal, como por ejemplo hacer que el robot siga a un individuo cargando con herramientas que vaya a utilizar.

Este comportamiento, al igual que los dos primeros, también se ha diseñado como comportamiento de navegación global, por lo que también se utiliza el algoritmo Wavefront para el trazado de las rutas del robot.

En la Figura 4.14 se muestra el flujograma que representa el funcionamiento de este comportamiento.

Una vez seleccionado este comportamiento en la ventana de opciones (ver Figura 4.1), se entra en la función que lo controla.

Lo primero que hace dicha función es leer la matriz de datos que devuelve el sensor láser, con un bucle de observación igual al que se ha explicado en el comportamiento

anterior. Al igual que en aquel, se busca la distancia menor desde los puntos de choque del haz de luz del láser al robot, y se guarda el índice donde se encontró.

Si la distancia menor igual a al rango del láser (8m en caso del láser en Stage, 4m para el Hokuyo), se hace que el robot se desplace con la dirección que lleva, avanzando para encontrar un objeto que seguir. Se chequean continuamente los datos del láser hasta que se obtenga una lectura que sea menor que el rango máximo del láser.

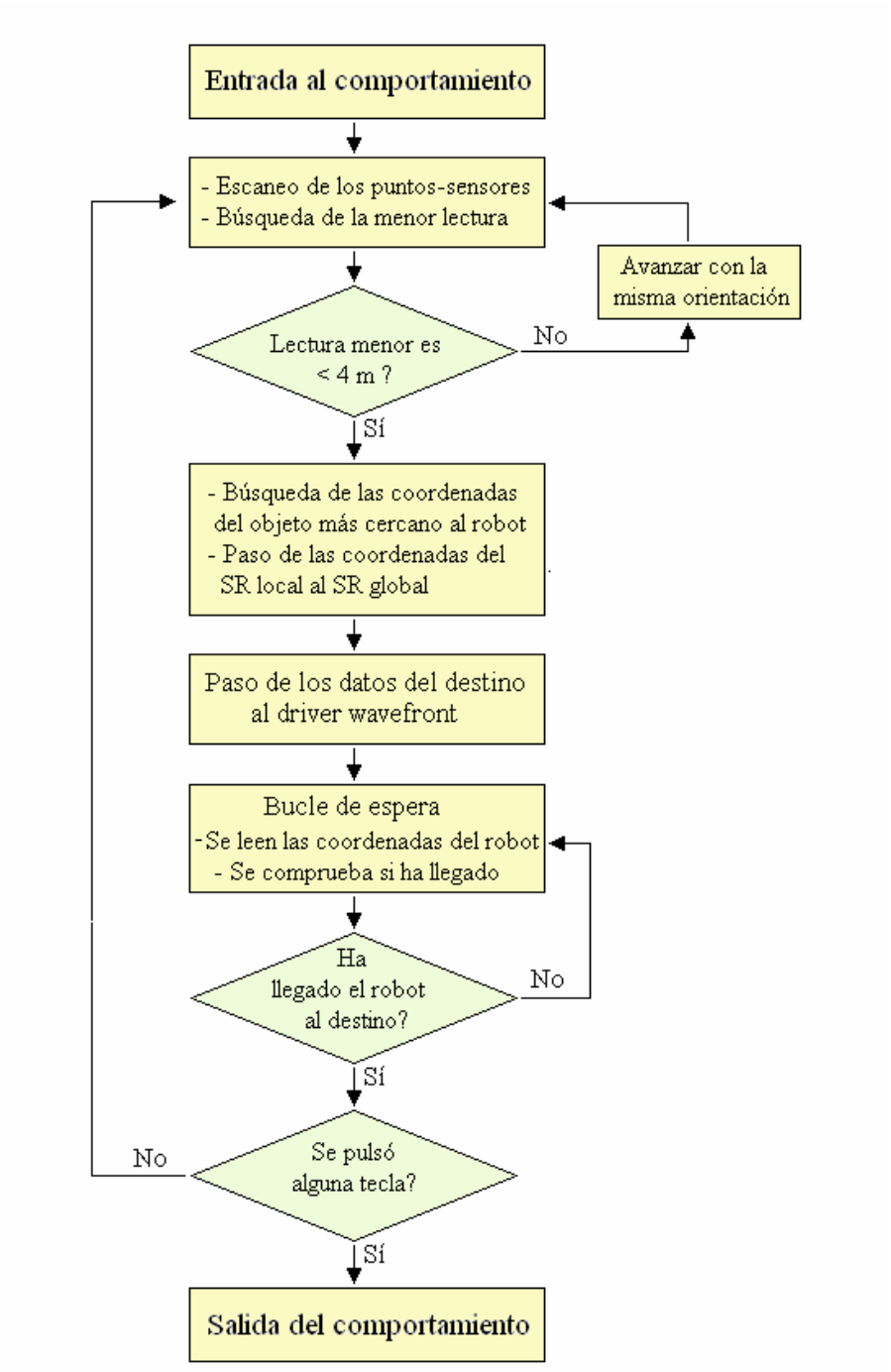


Figura 4. 14 - Flujograma que muestra el funcionamiento del "Seguimiento de Personas"

Cuando la distancia más pequeña sea inferior al rango del láser, se hace que el robot se desplace hasta las proximidades del objeto más cercano al robot. Para ello, se utiliza el índice de la matriz donde hemos encontrado la distancia menor, y se usan las coordenadas guardadas en la posición que indica dicho índice, para llevar al robot hasta el punto correspondiente a esas coordenadas.

Como los datos proporcionados por el láser están referidas al robot, es decir, están en un sistema de referencia local, hay que transformarlas en coordenadas del sistema de referencia global. Para esa transformación se multiplican las coordenadas referidas al robot por una matriz de rotación (R) y se suma la matriz de traslación (T). La ecuación que refleja dicho proceso se muestra a continuación:

$$P_G = R \cdot P_L + P_{G0}$$

$$\begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} \cos\varphi & -sen\,\varphi \\ sen\,\varphi & \cos\varphi \end{pmatrix} \cdot \begin{pmatrix} x' \\ y' \end{pmatrix} + \begin{pmatrix} Xo \\ Yo \end{pmatrix}$$

Siendo: (X, Y) coordenadas en el sistema global donde se tiene que dirigir el robot

(Xo, Yo) coordenadas en el sistema global donde se encuentra el robot

(x', y') coordenadas en el sistema local donde se tiene que dirigir el robot

φ ángulo diferencia entre los dos sistemas de referencia, es decir, es la orientación que tiene el robot respecto del sistema global.

En la Figura 4.15 se muestran los dos sistemas de referencia de los que se habla en el párrafo anterior.

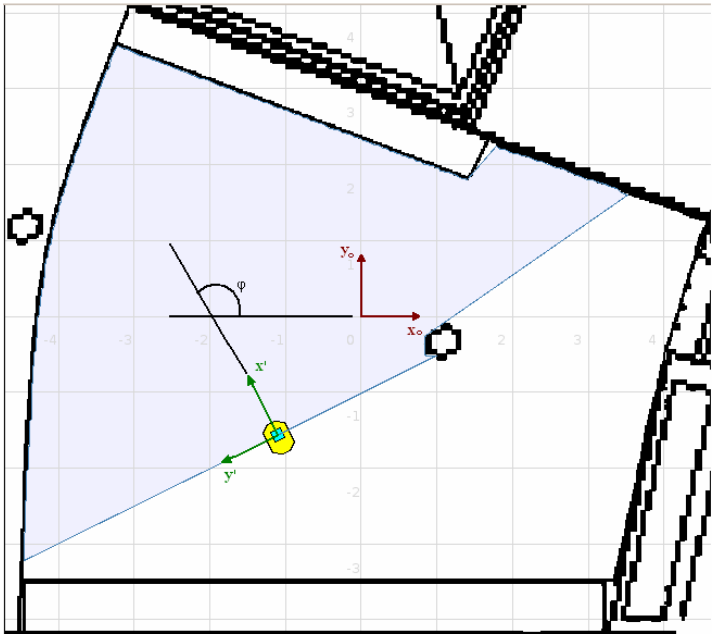


Figura 4. 15 - Sistemas de referencia global (rojo) y local (verde)

Una vez calculadas las coordenadas del objeto más cercano en el sistema de referencia global, se hace que el robot se dirija hacia el punto que éstas representan. Para ello se utiliza, al igual que en los dos primeros comportamientos, el driver de Wavefront, al que se le proporciona la posición a la que se tiene que desplazar mediante la función:

playerc_planner_set_cmd_pose(**playerc_planner_t** * *dispositivo,* **double** *x,* **double** *y,* **double** *theta)*;

Se espera en un bucle que el robot llegue al punto que hemos marcado como destino, comprobando periódicamente, al igual que en los casos anteriores, la distancia euclidea entre el robot y el destino. Una vez que el robot llega a una distancia del robot que se acepta como error marcada por el parámetro "error_destino", evalúa de nuevo la matriz devuelta por el dispositivo láser, buscando el objeto que esté a menor distancia del robot y se desplaza hasta él repitiendo el proceso que se ha explicado.

La salida de este comportamiento se hace pulsando cualquier tecla, y eso hace, al igual que en el resto de los comportamientos, que se devuelva el control al programa principal.

Mediante este comportamiento si el objeto más cercano al robot es un objeto en movimiento, el robot se acerca a él continuamente. En cambio, si el objeto más próximo es estático, el robot hace un primer acercamiento y se queda parado hasta encontrar un objeto más cercano a él.

4.5. Parámetros utilizados en los comportamientos

A continuación se detallan los parámetros utilizados en la programación de los comportamientos y de los que se ha hablado durante la explicación de los mismos.

- "error_destino": este parámetro fija la distancia máxima desde el robot hasta el destino marcado que es aceptada para considerar que el robot ha llegado a su meta. Este parámetro se utiliza en todos los comportamientos diseñados, y su valor es 0,35 m.

- "dist_avance": este parámetro fija la distancia que el robot debe adelantar cuando realiza un movimiento de avance. El valor de este parámetro es 0,20 m, y se usa en el comportamiento "Seguimiento de Paredes".

- "dist_aprox": este parámetro se usa en el comportamiento "Seguimiento de Paredes" y marca la distancia a la que se lleva al robot antes de colocarlo paralelo a la pared. Dicha distancia debe ser mayor que la fijada por el parámetro *safety_dist* del driver Wavefront, debido a que en este comportamiento no se va a permitir al robot aproximarse a los obstáculos más que en los comportamientos diseñados utilizando este algoritmo. En este proyecto se le ha dado el valor de 0,40 m.

- "dist_lateral_max": este parámetro utilizado en el comportamiento "Seguimiento de Paredes" fija la distancia máxima a la que puede llegar a estar el robot de la pared que sigue antes de iniciar una maniobra de reorientación. Su valor es 0,50.

- "dist_lateral_min": este parámetro se utiliza en el comportamiento "Seguimiento de Paredes" y marca la distancia mínima a la que puede llegar a estar el robot de la pared que sigue antes de iniciar una maniobra de reorientación. Dicha distancia debe ser mayor que la fijada por el parámetro *safety_dist* del driver Wavefront, debido a que en este comportamiento no se va a permitir al robot aproximarse a los obstáculos más que en los comportamientos diseñados utilizando este algoritmo. En este proyecto se le ha dado el valor de 0,35 m.

- "angulo_reorientación": este parámetro utilizado en el comportamiento "Seguimiento de Paredes" fija el ángulo que hay que girar en las maniobras de reorientación. Este parámetro toma el valor de 5 grados.

- "dist_frontal_min": este parámetro se utiliza en el comportamiento "Seguimiento de Paredes" y marca la distancia mínima desde el robot a la pared que tiene delante para considerar que el robot está frente a una esquina interior y tiene que comenzar la maniobra para salvarla. Dicha distancia debe ser mayor que la fijada por el parámetro *safety_dist* del driver Wavefront, debido a que en este comportamiento no se va a permitir al robot aproximarse a los obstáculos más que en los comportamientos diseñados utilizando este algoritmo. En este proyecto se le ha dado el valor de 0,45 m.

- "dist_esquina_ext": este parámetro utilizado en el comportamiento "Seguimiento de Paredes" fija la distancia lateral mínima, detectada por los haces del láser más próximos a la pared, que sirve para indicar que el robot se encuentra junto a una esquina exterior y debe comenzar la maniobra para salvarla. Este parámetro toma el valor de 0,80 m.

CAPÍTULO 5

RESULTADOS

En este capítulo se van a detallar las pruebas realizadas al software desarrollado, así como los resultados obtenidos de ellas. Se han realizado pruebas en Stage de cada uno de los comportamientos desarrollados para el robot. Las pruebas realizadas con los robots físicos no se han documentado porque no han sido suficientes para considerarlas concluyentes. Queda pendiente, como trabajo futuro, la aplicación de estas u otras pruebas realizadas con los robots físicos y su documentación.

5.1. Pruebas del comportamiento "Movimiento Básico"

Las pruebas realizadas a este comportamiento consisten en hacer que el robot simulado se desplace de un lugar a otro. Después de realizadas varias pruebas, teniendo en cada una de ellas diferentes puntos de origen y de destino, se observa como el robot simulado comienza su movimiento sin saber con precisión la posición donde se encuentra, pero tras un pequeño desplazamiento es capaz de localizarse correctamente gracias al algoritmo AMCL.

En la Figura 5.1 se ve un ejemplo de lo que se acaba de explicar. En esa prueba se le ha pedido al robot que se desplace desde el punto (x = -3.0 m, y = 0.0 m, θ = 0.0 rad) hasta el destino (x = 1.38 m, y = -2.08 m, θ = 4.7123 rad). Cuando se inicia la ejecución de la aplicación, el algoritmo AMCL nos devuelve la posición más probable donde se encuentra el robot (x = -1.42 m, y = -0.08 m, θ = -0.01 rad), en este caso se sabe que esa posición no

es la correcta. Después de iniciado el movimiento se ve en los puntos de la trayectoria del robot dibujados en la ventana de Stage, que AMCL es capaz de dar la localización correcta del robot.

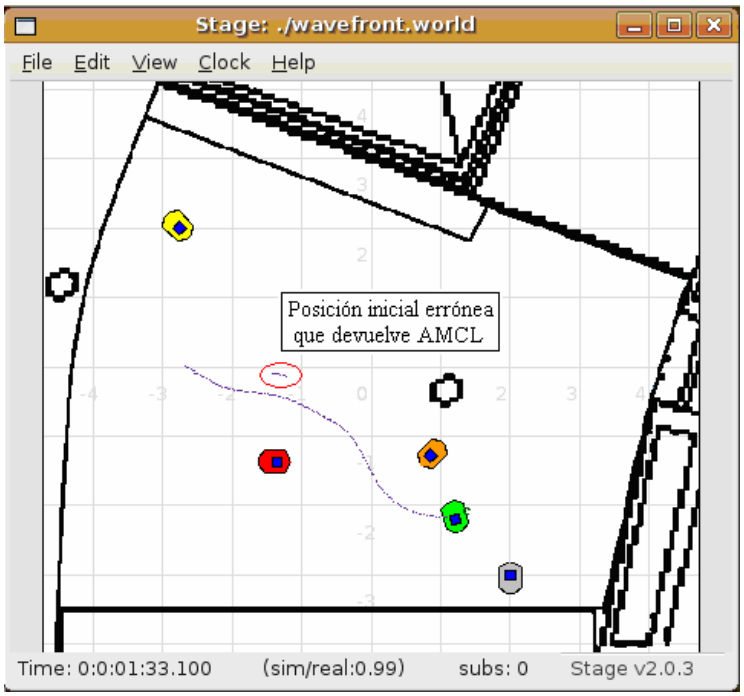


Figura 5. 1 - Primer ejemplo de funcionamiento del comportamiento "Movimiento Básico"

También se puede comprobar que el robot consigue llegar al destino fijado, siempre que no haya obstáculos que bloqueen el punto donde tiene que llegar. Es decir, si al robot se le marca como destino un punto del entorno en el que está situado un obstáculo que no aparece en el mapa, el robot no podrá alcanzar dicho destino, a no ser que las dimensiones del obstáculo sean tales que el robot, manteniendo la distancia de seguridad fijada para la evasión de obstáculos (parámetro *safety_dist*, ver Capítulo 4), pueda acercarse hasta una distancia igual o inferior al error que se le ha fijado al robot como aceptable para considerar que ha llegado (parámetro *distance_epsilon*, ver Capítulo 4).

En la Figura 5.2 se puede ver que se le ha fijado al robot un punto de destino en el cual hay situado otro robot. En esta prueba se le hace al robot desplazarse desde el punto (x = 1.45 m, y = 0.82 m, θ = 0.0 rad) hasta el destino (x = -2.42 m, y = -1.94 m, θ = 3.1415 rad). Se ve que cuando el robot se aproxima al punto fijado como destino, al no poder alcanzarlo, comienza a rodearlo intentando darle alcance, pero hasta que no se retira el obstáculo de la meta, el robot no es capaz de alcanzar ese punto.

Siempre que el robot se encuentra con un obstáculo lo esquiva perfectamente y continúa con la ruta marcada. Si hay muchos obstáculos que impiden que el robot pase por una determinada zona, se observa que éste tiene que dar un rodeo para evitarlos y llegar a su destino.

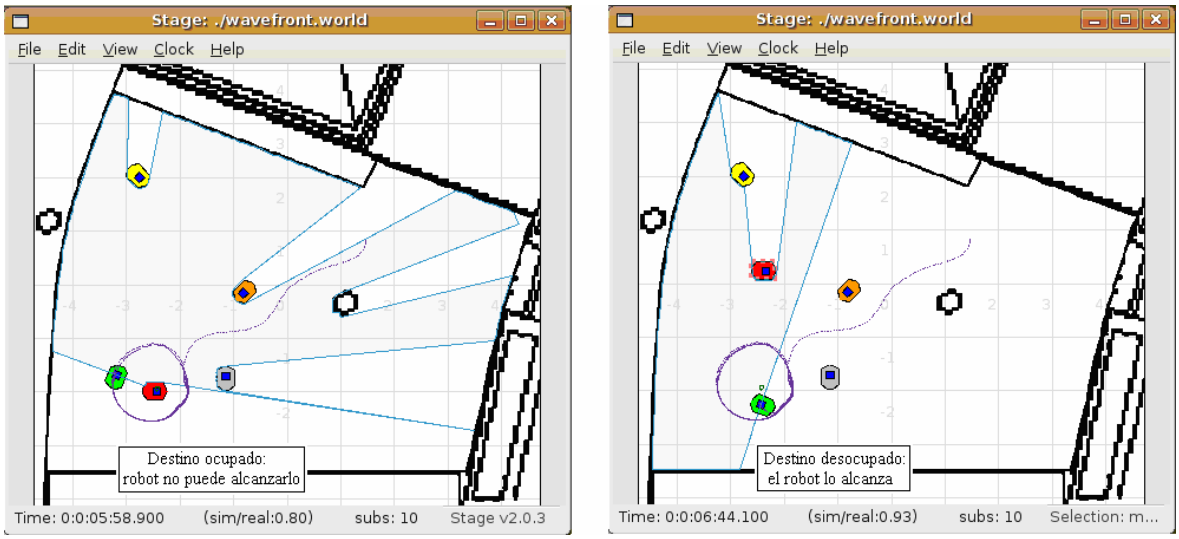


Figura 5. 2 – Segundo ejemplo de funcionamiento del comportamiento "Movimiento Básico"

5.2. Pruebas del comportamiento “Recorrido de un Conjunto de Puntos”

En este comportamiento las pruebas han consistido en hacer que el robot recorriera diversos conjuntos de puntos. Durante dichas pruebas se observa, al igual que en el comportamiento anterior que, al principio del primer desplazamiento que se realiza, el robot no sabe con precisión donde está, pero después de un pequeño movimiento se localiza correctamente.

En las pruebas realizadas a este comportamiento se pueden hacer las mismas observaciones que en el comportamiento anterior. Si el destino está bloqueado por un obstáculo, el robot no consigue llegar a él. Y si el robot encuentra muchos obstáculos dificultando la realización de su ruta, se ve obligado a cambiarla y dar un pequeño rodeo hacia el destino.

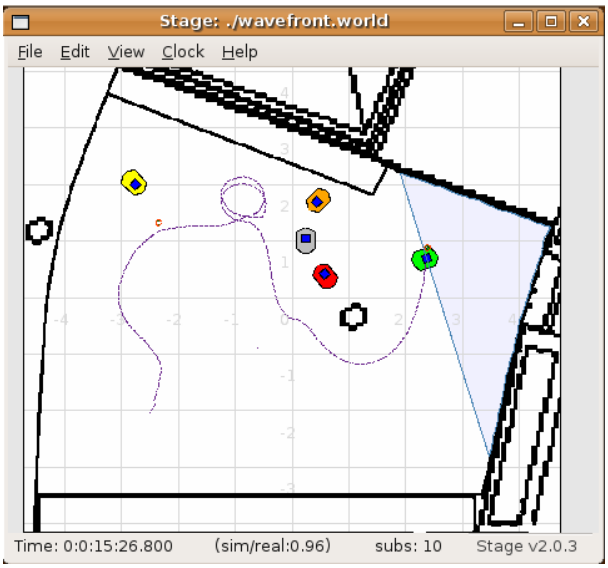


Figura 5. 3 - Ejemplo de funcionamiento del comportamiento "Recorrido de un Conjunto de Puntos"

En la Figura 5.3 se ve un ejemplo de la última peculiaridad del comportamiento que se ha comentado. En esa prueba se le ha pedido al robot que, estando en el punto (x = -2.48 m, y = -2.04 m, θ = 0.0 rad), se desplace primero al punto (x = -2.38 m, y = 1.22 m) y después a (x = 2.37 m , y = 0.96 m). Ambos destinos están representados en la imagen mediante un círculo. Se ve en la mencionada figura que en el segundo desplazamiento el robot hace un rodeo intentando buscar un lugar por el que pasar para llegar a su destino. Esto se debe a la existencia de los parámetros para el replanteamiento de rutas (parámetros *replan_dist_thresh* y *replan_min_time*, ver capítulo 4).

5.3. Pruebas del comportamiento "Seguimiento de las Paredes del Entorno"

Para este comportamiento las pruebas realizadas han consistido en situar el robot en cualquier punto del entorno simulado y ordenarle que encuentre la pared más próxima y la recorra manteniendo cierta distancia con ella.

En la Figura 5.4 se observa como el robot sale de la posición (x = -0.48 m, y = 1.37 m, θ = 3.1234 m), detecta la pared más próxima y comienza a seguirla hasta que se pulsa cualquier tecla para parar el comportamiento.

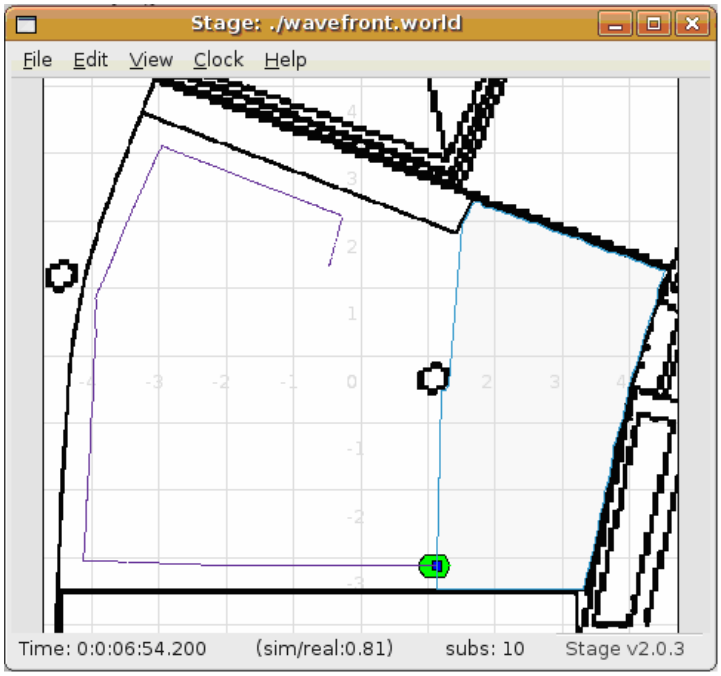


Figura 5. 4 – Primer ejemplo de funcionamiento del comportamiento "Seguimiento de Paredes"

En la Figura 5.4 se ve que la ruta que desarrolla el robot sigue la pared perfectamente porque se ha ejecutado el comportamiento en Stage, y en el simulador no hay errores de odometría.

Al estar basado en la navegación local, este comportamiento utiliza las consignas de posición proporcionadas por el sistema de odometría. En el arranque, Player siempre toma

como posición de origen del robot (x = 0.0 m, y = 0.0 m, θ = 0.0 rad), por tanto, si en el fichero del mundo simulado (.world) la posición inicial del robot es cualquier otra, y no se ha ejecutado otro comportamiento de localización global que proporcione la posición correcta antes que éste, la ruta que se dibuja en la ventana de Stage no es la que sigue el robot en realidad, porque no se usa el localizador global en este comportamiento. A pesar de esto, el robot realiza correctamente el seguimiento de paredes debido a que el comportamiento está basado en navegación reactiva.

En la Figura 5.5 se puede ver un ejemplo del caso que se acaba de mencionar. En esta prueba se ha fijado en el fichero .world la posición de inicio del robot (x = 0.0 m, y = 0.0 m, θ = 180º), pero desde el punto de vista del sistema de odometría se parte del punto (x = 0.0 m, y = 0.0 m, θ = 0.0 rad). Por este motivo, la ruta que se dibuja en la ventana de Stage corresponde con la imagen especular de la que realmente realiza el robot. Aunque en la ventana de Stage el dibujo de la ruta no sea la que sigue el robot, la trayectoria que realiza el éste es la correcta.

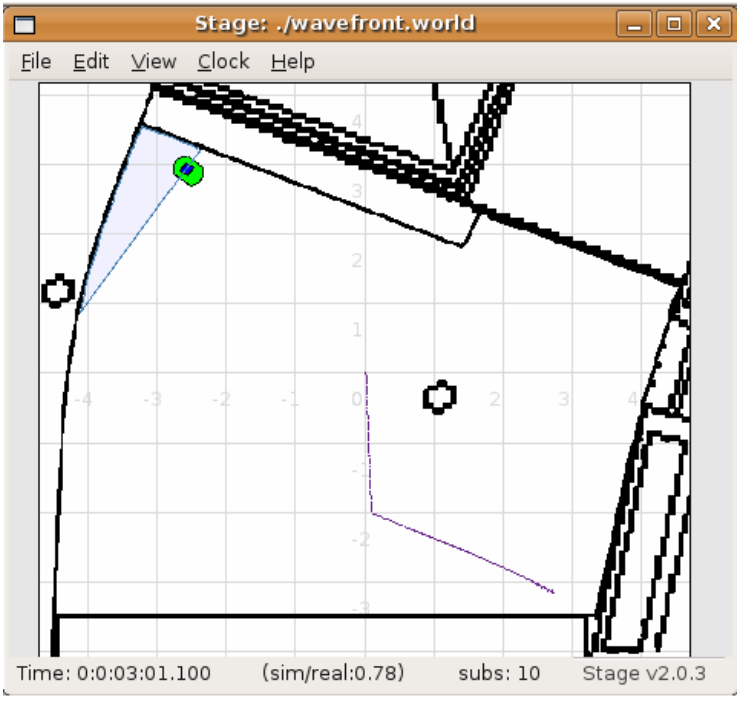


Figura 5. 5 – Segundo ejemplo de funcionamiento del comportamiento "Seguimiento de Paredes"

Durante estos ensayos se observa que si el robot detecta un obstáculo antes de detectar una pared, toma al primero como pared, se dirige hacia él y sigue su contorno. Esto se debe a que en este comportamiento no se utiliza el mapa del entorno, y el robot no es capaz de discriminar lo que es pared y lo que es obstáculo.

En la Figura 5.6 se observa la característica de este comportamiento que se acaba de explicar. En dicha imagen se ve como el robot parte de la posición (x = 0.0 m, y = 0.0 m, θ = 0.0 rad), y comienza a seguir lo que toma como la pared más cercana, que en realidad es un obstáculo fijo.

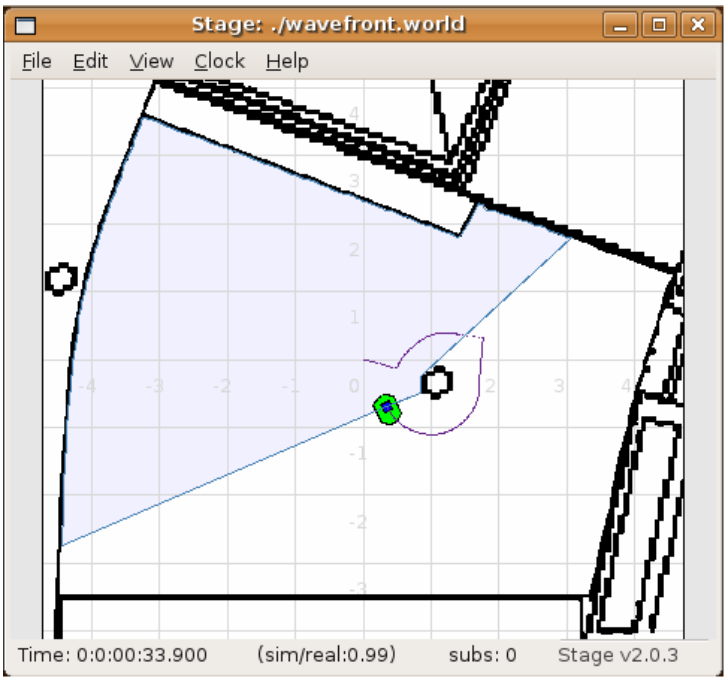


Figura 5. 6 - Tercer ejemplo de funcionamiento del comportamiento "Seguimiento de Paredes"

5.4. Pruebas del comportamiento “Seguimiento de una Persona u Objeto Móvil”

Para este comportamiento los ensayos han consistido en situar el robot en cualquier punto del entorno simulado, y desplazar un obstáculo móvil por las proximidades de éste. Haciendo esto se observa como el robot se desplaza siguiendo el punto del obstáculo que más cerca está de él. Si no hay ningún obstáculo moviéndose cerca de él, el robot se dirige hacia el obstáculo fijo o la pared más cercanos a él.

En la Figura 5.7 se ve un ejemplo de funcionamiento de este comportamiento en el que el avance de los robots está simbolizado mediante el cambio de tonalidad de los mismos. En esta prueba se observa como el robot comandado (verde) se acerca al objeto que más próximo está, que en este caso es otro robot (naranja). Al mover este último, el robot comandado lo persigue, hasta que llega a un punto donde ve otro objeto más cerca, el robot gris, y se acerca a él. Cuando se retira el robot gris, el verde sigue de nuevo al naranja hasta que el usuario detiene el comportamiento. En la imagen mencionada se puede ver el recorrido que hacen los robots verde, naranja, y gris.

En algunas pruebas se ha observado que si el driver AMCL no ha sido capaz de localizar correctamente la posición del robot, éste puede ser enviado a puntos demasiado próximos, o incluidos en las paredes del entorno y el algoritmo Wavefront no encuentra una ruta para llevar al robot a esa posición.

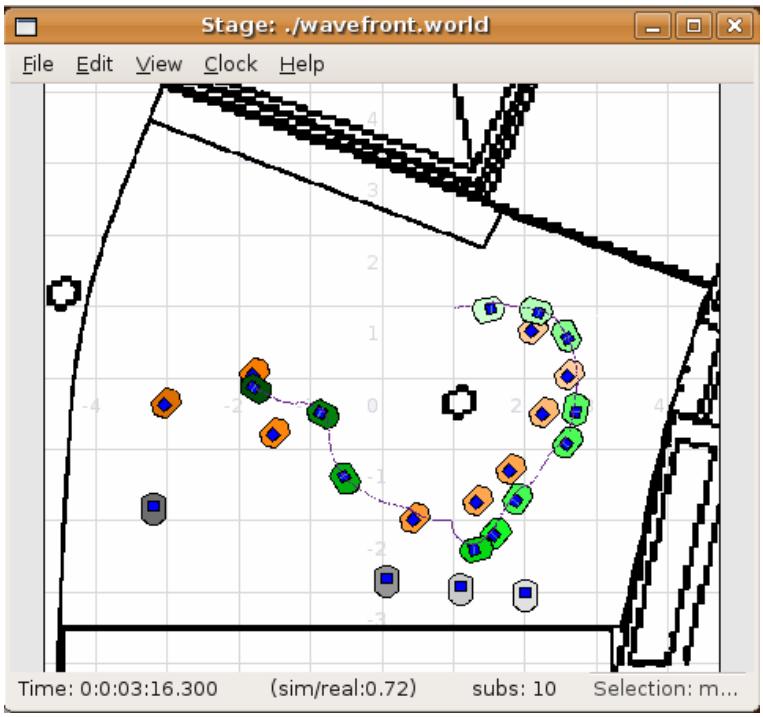


Figura 5. 7 - Ejemplo de funcionamiento del comportamiento "Seguimiento de Personas"

CAPÍTULO 6

CONCLUSIONES Y TRABAJOS FUTUROS

6.1. Conclusiones

La finalidad de este proyecto ha sido desarrollar varios comportamientos para la navegación de robots móviles, tanto físicos como en sus modelos de simulación, utilizando para ello las herramientas disponibles en la plataforma software del Proyecto Player.

El uso de esta plataforma ha resultado útil a la hora de programar las aplicaciones, ya que los algoritmos de navegación local y global utilizados durante este proyecto poseen sus propios drivers en el servidor Player. El hecho de que dichos algoritmos estén ya implementados como drivers de Player ha facilitado su manejo, ya que sólo ha sido necesario conectar al servidor los dispositivos incluidos en cada uno de ellos, y proporcionar las consignas pertinentes al dispositivo correspondiente en cada caso.

La dificultad en el manejo de los drivers de Player ha venido a la hora de manejar sus parámetros para adaptarlos a las necesidades de las aplicaciones programadas. El uso y los valores predeterminados de algunos de esos parámetros no está claramente especificado en la ayuda incluida en el propio Proyecto Player. Por este motivo, en algunos casos se han tenido que hacer pruebas cambiando de forma empírica los valores de dichos parámetros para entender la utilidad de los mismos.

Una vez salvadas las dificultades relacionadas con el uso de los parámetros de los drivers, es relativamente sencillo desarrollar aplicaciones de robótica móvil con este software, siempre que se tengan unos conocimientos básicos de robótica y del lenguaje de programación que se esté utilizando.

Los comportamientos desarrollados se han probado sobre el simulador Stage incluido en la plataforma del Proyecto Player. En las pruebas realizadas en este simulador las aplicaciones implementadas funcionan correctamente, ya que en los modelos de simulación no están incluidos errores tales como los que acumula el sistema de odometría. Sin embargo, en las pruebas de los comportamientos realizadas con los robots físicos se observa que el funcionamiento de los mismos no es el esperado. Para conseguir el funcionamiento deseado habría que hacer un ajuste más concienzudo de los parámetros incluidos en cada uno de los drivers utilizados.

6.2. Trabajos Futuros

a) Ampliación del número de pruebas realizadas con los robots físicos

El objetivo de este proyecto era implementar aplicaciones de control para robots simulados y físicos. Las pruebas realizadas con estos últimos no han sido suficientes para conocer en profundidad el comportamiento sobre los robots de las aplicaciones programadas y probadas en el simulador. Por este motivo estas pruebas no se han documentado en esta memoria.

Por tanto, se puede incrementar el número de pruebas realizadas en los robots físicos para observar más en profundidad, y documentar, el funcionamiento sobre ellos de los comportamientos de navegación desarrollados en este proyecto.

b) Implementación de un navegador global utilizando el localizador global integrado en el Espacio Inteligente

El laboratorio donde se han realizado las pruebas, el espacio inteligente, posee una red de cámaras de video para grabar su interior, que varios desarrolladores han utilizado para implementar allí un localizador global.

A partir de ese localizador se puede desarrollar un navegador global nuevo, o para utilizarlo con un navegador ya implementado. Por ejemplo, se podrían utilizar las aplicaciones desarrolladas en este proyecto sustituyendo el algoritmo AMCL por el mencionado localizador global, que también está basado en un filtro de partículas. (ver [Pizarro 08])

Para hacer uso del mencionado localizador global en las aplicaciones desarrolladas con Player durante este proyecto, habría que crear un sistema servidor-cliente para comunicar

el PC que controla el localizador y el ordenador en el que se ejecutan las aplicaciones de control.

c) Creación de aplicaciones utilizando otros algoritmos implementados en Player

Además de los algoritmos utilizados durante este proyecto, en el Proyecto Player están implementados muchos más. Se podrían programar los mismos comportamientos de control de robots que se han desarrollado en este proyecto haciendo uso de otros algoritmos de navegación y localización diferentes, como por ejemplo el algoritmo ND (ver [Minguez04]).

También se podría hacer una implementación del comportamiento "Seguimiento de Paredes" basado en un algoritmo de navegación global, en lugar de utilizar navegación local como en este proyecto.

Además, se pueden desarrollar las aplicaciones generadas en este proyecto utilizando los sensores de ultrasonidos del robot en lugar de utilizar el láser. En ese caso, sólo se tendría información de un pequeño número de puntos alrededor del robot, a diferencia del láser que proporciona información de toda el área de barrido. Por tanto, habría que tener en cuenta esta característica a la hora de programar las aplicaciones, para ajustar los parámetros adecuadamente.

III. MANUAL DE USUARIO

En esta parte del proyecto se pasará a explicar las acciones a realizar para poner en marcha las aplicaciones software que se han desarrollado para el robot.

III.1. Software Necesario

Para utilizar las aplicaciones programadas en este proyecto es necesario disponer de un ordenador con una distribución de Linux instalada. En este manual se especificarán las instrucciones para una distribución de Ubuntu 7.10.

Además de disponer de este sistema operativo, será necesario tener instalados los programas Player y Stage, para ello se necesitarán, a su vez, las siguientes paquetes accesorios: gcc, g++, autoconf, automake, libtool, make, libgtk2.0-0, libgtk2.0-dev, libgtk-dev, pkg-config, libjpeg62, libjpeg62-dev, libltdl3, libltdl3-dev, gsl-1.9, libxml2-2.6.29, libglade-2.6.1, libart-lgpl-2.3.19 y libgnomecanvas-2.14.0. Este manual está escrito para las versiones player-2.0.4 y stage-2.0.3.

Aparte de estas herramientas, utilizaremos también algunas funciones de las librerías OpenCV para el manejo de imágenes, por lo que será necesario tener instaladas también estas librerías.

III.2. Instrucciones a seguir

Una vez se tienen escritos el archivo de configuración de Player (.cfg), donde configuramos los parámetros de control de los algoritmos, el programa de control (.c), donde tenemos las funciones de conexión del cliente y de control de los dispositivos del robot, y el archivo *makefile* para compilar este último, se pasa a poner en marcha la aplicación desarrollada.

Los pasos a seguir para poner en marcha la aplicación son siempre los mismos, ya se esté trabajando con un robot físico o con un modelo de Stage. Las diferencias entre ambos casos vendrán dadas por el contenido de los archivos de configuración y de control.

La primera acción a realizar será entrar en el directorio donde se encuentran nuestros programas fuente. Para ello se utiliza el comando *cd,* seguido de la ruta del directorio en cuestión (en el ejemplo, Escritorio/Comportamientos), como aparece en la Figura III.1.

Después de haber entrado en el directorio donde se encuentran los fuentes tendremos que abrir una segunda pestaña del terminal (Archivo → Nueva Pestaña o Ctrl+Mayúsculas+T). De esas dos pestañas, una la usaremos para ejecutar el programa servidor y otra para ejecutar el cliente.

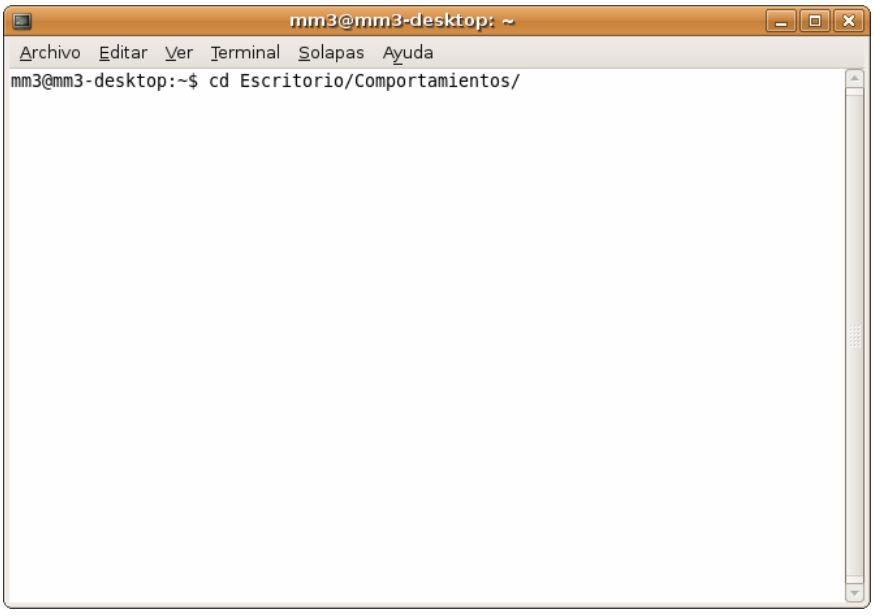


Figura III. 1 - Entrada al directorio donde están nuestros programas

En la primera de las pestañas se pone en marcha el programa servidor. Para ello, se ejecuta el comando *player* seguido del nombre del archivo de configuración con el que se trabaja (en este caso, stage.cfg). Si se trabaja con Stage, al ejecutar esta sentencia se abrirá una ventana con el mundo que hemos creado (fichero.world). Si se trabaja con el robot físico, con esta orden se deja al robot dispuesto para recibir la conexión del cliente.

En la Figura III.2 se muestra la sentencia que pone en marcha el servidor, y en la Figura III.3 se ve la ventana que se abre al ejecutarla cuando se trabaja con Stage. En esta última se muestra el entorno recreado para la simulación.

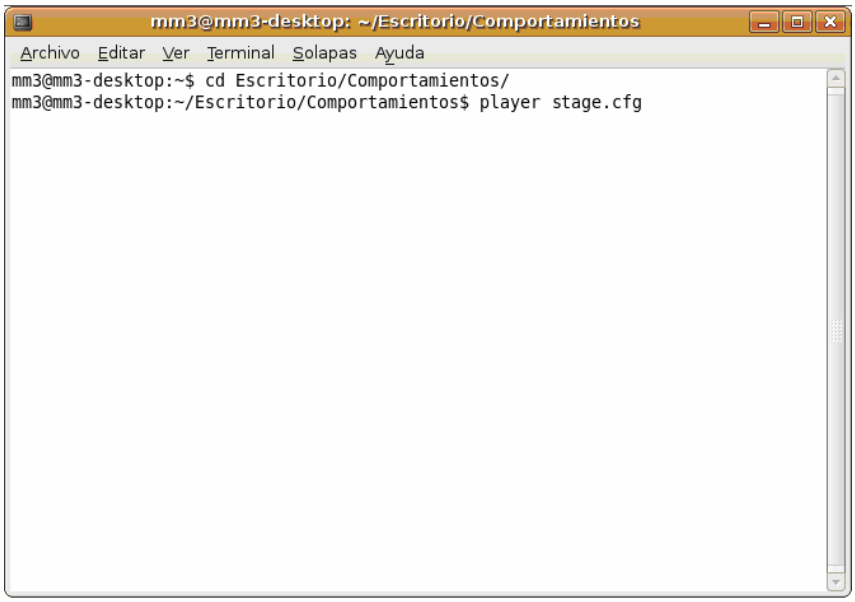


Figura III. 2 - Puesta en marcha del programa servidor

Una vez arrancado el programa servidor, hay que poner en marcha el programa cliente en la otra pestaña. Primero compilaremos el archivo fuente (en este caso, stage.c) para crear un ejecutable. Para ello, en la segunda pestaña ejecutaremos el archivo "*makefile*" que tenemos para generar los ejecutables. En la consola, escribiremos la orden para eliminar posibles ejecutables anteriores (*make clean*), y a continuación la sentencia que nos servirá para crear el nuevo ejecutable (*make all*).

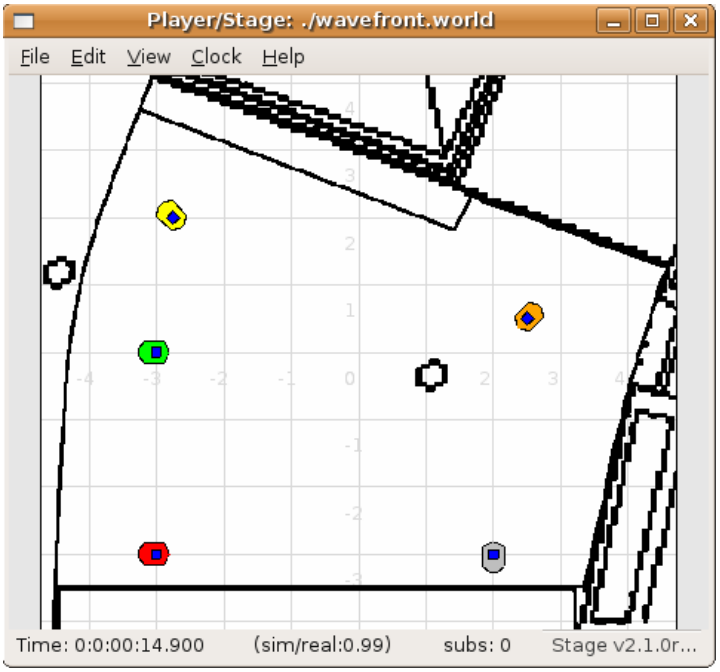


Figura III. 3 - Ventana del entorno simulado

Después de esto, pondremos en marcha el programa cliente escribiendo en el terminal "./ejecutable" (en el ejemplo, ./stage). En la Figura III.4 se muestran las órdenes de compilación y ejecución del programa de control.

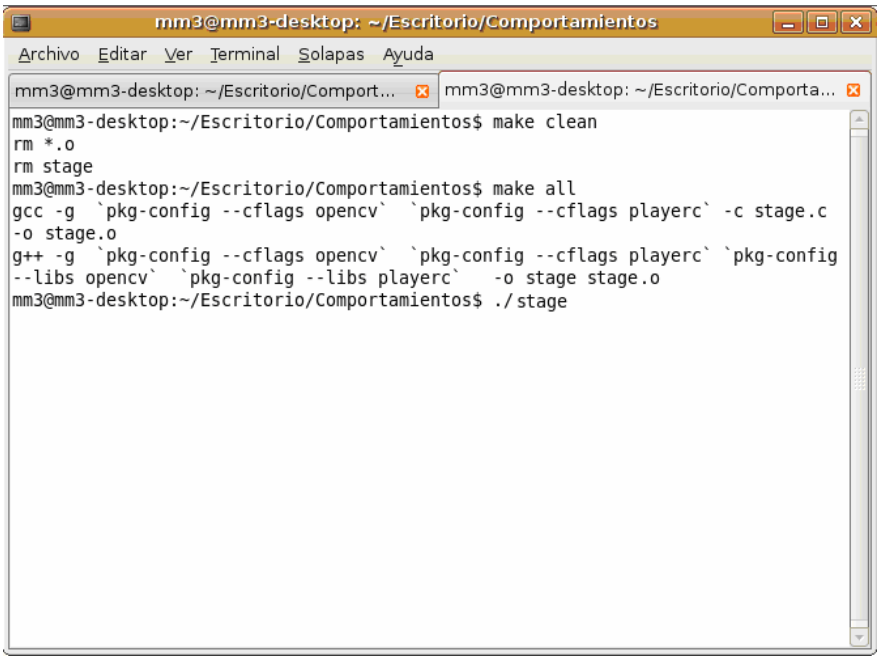


Figura III. 4 - Compilación y puesta en marcha del ejecutable

Al arrancar la aplicación de control que se ha desarrollado se abrirán dos ventanas: una con los botones para elegir la acción a realizar (ver Figura III.6), y otra con el mapa del espacio inteligente (*ispace*), que es el laboratorio que hemos recreado en el simulador para hacer pruebas con el robot (ver Figura III.5).

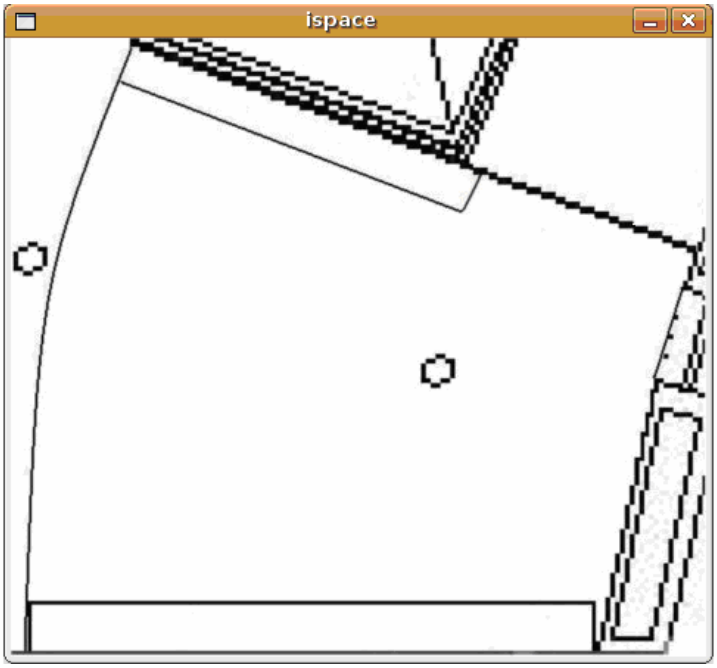


Figura III. 5 - Ventana con el mapa del Espacio Inteligente

En la ventana de opciones (ver Figura III.6) tendremos que pulsar en primer lugar el botón de "Conectar". Con esa acción crearemos un cliente y haremos que se suscriba a todos los dispositivos necesarios. En caso de seleccionar cualquier otra opción antes de pulsar el botón "Conectar", se provoca un error que trae como consecuencia el fin de la aplicación.

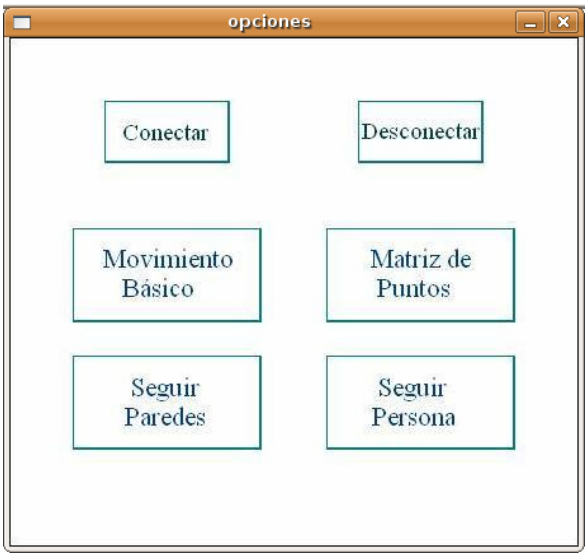


Figura III. 6 - Ventana de Opciones

Después de seleccionar la opción "Conectar", la ventana de Stage que contiene el espacio se verá como muestra la Figura III.7.

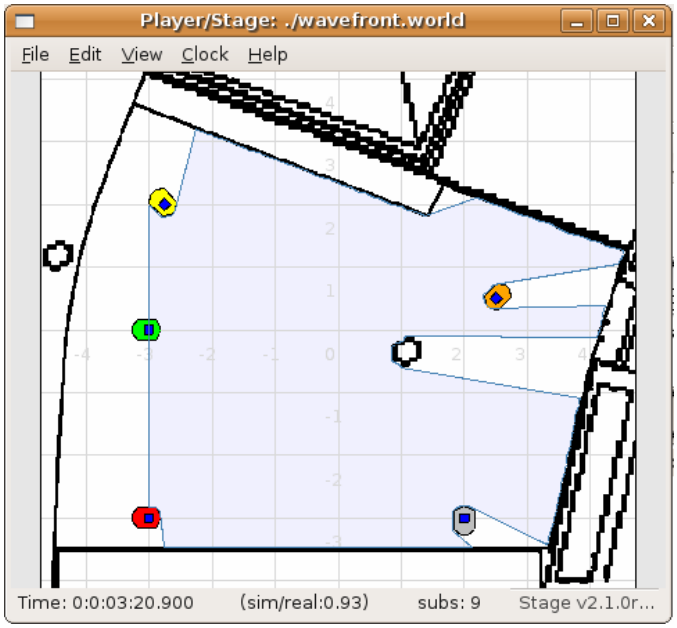


Figura III. 7 - Conexión del cliente y suscripción a los dispositivos

Cuando el robot está conectado, se puede pulsar la opción "Desconectar" o seleccionar alguno de los cuatro comportamientos desarrollados. Si se selecciona "Desconectar" se retira la suscripción a todos los servicios mencionados anteriormente, y se destruye el cliente.

Por otro lado, la selección de cada comportamiento hace que el programa principal ceda el control a la función que rige cada comportamiento. Y, además, al seleccionar algún comportamiento se crea una ventana con el mapa del espacio y, en su caso, botones para seleccionar opciones del propio comportamiento. En esta ventana se dibuja el recorrido que hace el robot al ejecutar el comportamiento.

Los comportamientos de "Seguimiento de Paredes" y "Seguimiento de Personas" arrancan directamente cuando pulsamos su correspondiente botón. En cambio, al ejecutar los comportamientos de "Movimiento Básico" y "Recorrido de un Conjunto de Puntos" se le pide al usuario, con mensajes en la pantalla, que introduzca datos necesarios para realizar el comportamiento por medio del teclado o pinchando sobre la ventana que se crea para cada uno de los comportamientos.

Cuando se selecciona "Movimiento Básico" se pide por la pantalla que se pinche sobre la ventana correspondiente el punto que se quiere como destino, y que se pulse también sobre las flechas preparadas al efecto para seleccionar la orientación final del robot. Una vez hecho esto, el comportamiento entra en funcionamiento. En la Figura III.8 se muestra la ventana que se crea al entrar a este comportamiento.

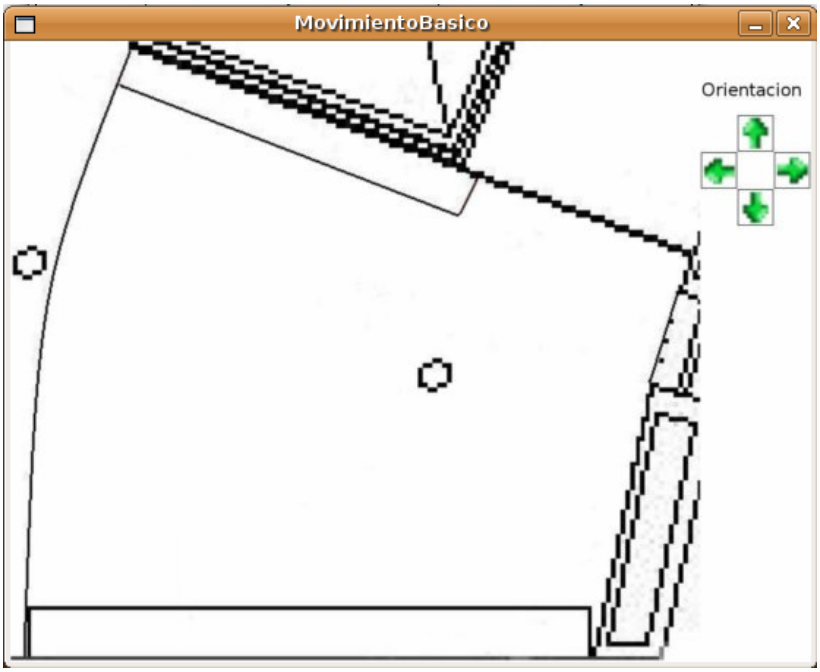


Figura III. 8 - Ventana correspondiente al comportamiento "Movimiento Básico"

Al seleccionar "Recorrido de un Conjunto de Puntos" se pide al usuario que seleccione en la ventana correspondiente los puntos del entorno por los que se quiere hacer pasar al robot. El usuario debe pinchar sobre los puntos deseados y después de pinchar el último punto debe pulsar el botón de "Fin de la selección de puntos". Una ver pulsado este botón el comportamiento entra en funcionamiento. En la Figura III.9 se muestra la ventana que se crea al iniciar este comportamiento.

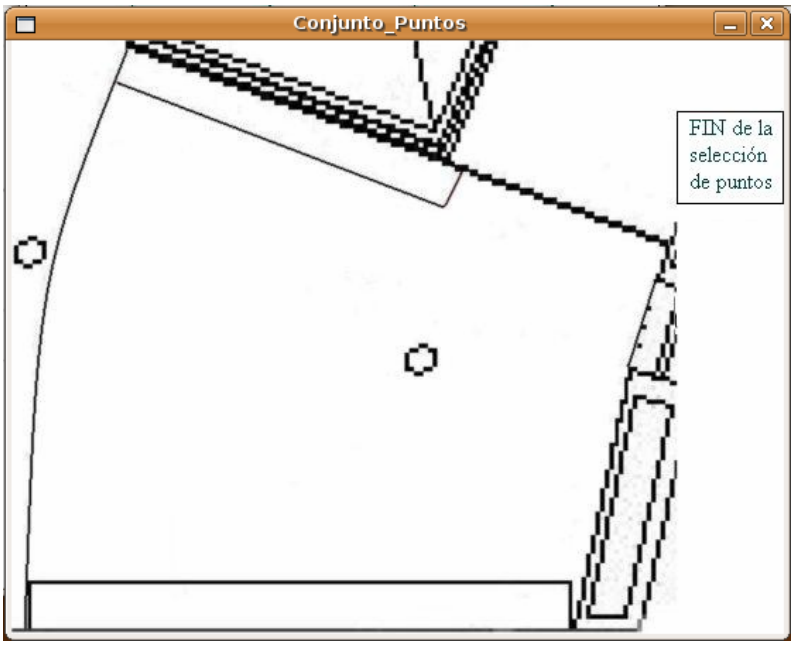


Figura III. 9 - Ventana correspondiente al comportamiento "Recorrido de un Conjunto de Puntos"

Si se desea salir de un comportamiento antes de que éste haya finalizado, se debe pulsar cualquier tecla, teniendo activa la ventana del comportamiento correspondiente. La salida del comportamiento no se lleva a cabo hasta que el robot no finaliza el trayecto en curso.

IV. PLIEGO DE CONDICIONES

A continuación se detallan las características técnicas de los elementos físicos y las herramientas software utilizados durante el desarrollo de este proyecto.

IV.1. Equipos Físicos

- Robot Pioneer3-DX

Dimensiones	44,5 x 40 x 24,5 (largo - ancho - altura)
Peso	9 Kg (con la batería de menor capacidad)
Carga máxima	23 kg
Velocidad lineal máxima	1,2 m/s
Batería	12 V
Autonomía	18 - 24 h
Tiempo de carga	12 h
Tracción	Diferencial, 2 ruedas motrices y 1 *castor*
Anillo frontal de US	Incluido. 8 sónar: 1 a cada lado, 6 delante, 1 cada 15º
Anillo trasero de US	Opcional. 8 sónar: situados como en anillo delantero
Conexión	Inalámbrica o por puerto serie

- Robot Pioneer3-AT

Dimensiones	50 x 49 x 26 (largo - ancho - altura)
Peso	12 Kg (con la batería de menor capacidad)
Carga máxima	30 kg
Velocidad lineal máxima	1,2 m/s

Batería	12 V
Autonomía	4 - 8 h
Tiempo de carga	12 h
Tracción	4 ruedas motrices, *skid-steer*
Anillo frontal de US	Opcional. 8 sónar: 1 a cada lado, 6 delante, 1 cada 15º
Anillo trasero de US	Opcional. 8 sónar: situados como en anillo delantero
Conexión	Inalámbrica o por puerto serie

- Amigobot

Dimensiones	33 x 28 x 15 (largo - ancho - altura)
Peso	3,6 Kg (con la batería de menor capacidad)
Carga máxima	1 kg
Velocidad lineal máxima	1 m/s
Batería	12 V
Autonomía	2 h
Tiempo de carga	6 - 8 h
Tracción	Diferencial, 2 ruedas motrices y 1 *castor*
Anillo frontal de US	Incluido. 8 sónar: 4 delante, 2 a los lados, 2 detrás
Anillo trasero de US	Incluido. 2 sónar detrás
Conexión	Inalámbrica o por puerto serie

- Láser Hokuyo modelo URG-04LX

Dimensiones	50 x 50 x 70 (largo - ancho - altura)
Peso	160 g
Tensión de funcionamiento	5 V
Distancia de reconocimiento	Desde 20 mm hasta 4 m
Resolución	1 mm
Área de barrido	± 120º
Tiempo de escaneo	100 ms/scan

- Ordenador portátil Asus

Microprocesador	Intel - Centrino DUO
Velocidad de reloj	1.66 GHz
Memoria RAM	2 Gbyte
Disco duro	100 GBytes

- Impresora HP Deskjet F380

Tecnología de impresión	Inyección térmica de tinta
Velocidad de páginas (bn/c)	20/14 ppm
Calidad de impresión	4800 x 1200 ppp
Memoria estándar	32 MB

IV.2. Software

- Sistema operativo Ubuntu 7.10.
- Sistema operativo Windows XP.
- Procesador de textos (Microsoft Word, Gedit).

- Programa Player-2.0.4
- Programa Stage-2.0.3
- Librería OpenCV.1.0
- Son necesarios también los siguientes paquetes de Ubuntu: gcc, g++, autoconf, automake, libtool, make, libgtk2.0-0, libgtk2.0-dev, libgtk-dev, pkg-config, libjpeg62, libjpeg62-dev, libltdl3, libltdl3-dev, gsl-1.9, libxml2-2.6.29, libglade-2.6.1, libart-lgpl-2.3.19 y libgnomecanvas-2.14.0.

V. PRESUPUESTO

En esta parte del proyecto se hará una estimación del coste total que supone la ejecución del mismo. En los apartados siguientes aparecen los gastos agrupados según su origen, y en el último apartado se detalla el presupuesto total.

V.1. Ejecución material

A continuación se va a desglosar el presupuesto de ejecución material en los tres elementos siguientes:

- Coste de equipos.
- Coste del material utilizado.
- Coste de mano de obra por el tiempo empleado.

V.1.1. Coste de equipos

EQUIPO	PRECIO
Pioneer3-DX	2.954 €
Pioneer3-AT	4.926 €
Amigobot	1.687 €
Laser Hokuyo	1.673 €
Ordenador Portátil Asus	999 €
Impresora HP	80 €
Total	**12.319 €**

V.1.2. Coste del material utilizado

MATERIAL	PRECIO
Ubuntu 7.10	0 €
Microsoft Windows XP profesional	135 €
Microsoft Office XP	200 €
Player-2.0.4	0 €
Stage-2.0.3	0 €
Librería OpenCV 1.0	0 €
Material de oficina	120 €
Total	**455 €**

V.1.3. Coste de la mano de obra por el tiempo empleado

FUNCIÓN	Nº HORAS	€/HORA	TOTAL
Ingeniería	850	50 €	42.500 €
Mecanografiado	100	12 €	1.200 €
		Total	**43.700 €**

V.1.4. Coste total del presupuesto de ejecución material

PARTIDA	PRECIO
Coste de equipos	12.319 €
Coste Material	455 €
Coste por tiempo empleado	43.700 €
Total	**56.474 €**

V.2. Gastos generales y beneficio industrial

Este apartado abarca los gastos generados por las instalaciones utilizadas durante la realización del proyecto más el beneficio industrial. A efectos de cálculo se estima este apunte como el 20 % del presupuesto de ejecución material del proyecto.

El valor de los gastos generales y beneficio industrial asciende a **11.295 €.**

V.3. Honorarios de dirección y redacción

Se han calculado, tanto los honorarios de dirección como los de redacción, como el 7 % del presupuesto de ejecución material.

Presupuesto de ejecución material	**56.474 €**
Honorarios por dirección	**3.953 €**
Honorarios por redacción	**3.953 €**

V.4. Coste de ejecución por contrata

Este concepto incluye el presupuesto de ejecución material, los gastos generales y el beneficio industrial.

Presupuesto de ejecución material	56.474 €
Gastos generales y beneficio industrial	11.295 €
Honorarios de dirección	3.953 €
Honorarios de redacción	3.953 €
Total	**75.675€**

V.5. Presupuesto total

Presupuesto de ejecución por contrata	75.675 €
IVA (16%)	12.108 €
TOTAL	**87.783 €**

El presupuesto total del proyecto asciende a la cantidad de ochenta y siete mil setecientos ochenta y tres **euros.**

Alcalá de Henares a 24 de Septiembre de 2008.

Firmado: Inés Sanz Alonso

Ingeniero Técnico Industrial

VI. PLANOS

En esta parte del proyecto se adjuntan los códigos que se han generado para las aplicaciones de control del robot. Primero se detallan los programas de configuración de Player (archivos .cfg) trabajando con Stage y con el robot físico. Después se incluyen los códigos que implementan el programa de control del robot (archivos .c y .h).

VI.1. Códigos de configuración de Player

A continuación aparecen los códigos necesarios para configurar los parámetros de los drivers del robot y de los algoritmos utilizados, trabajando con Stage y con los robots físicos.

VI.1.1. Creación del mundo simulado y configuración del robot simulado en Stage

En este subapartado se detallan los códigos de los archivos necesarios para trabajar con el robot simulado en Stage.

a) stage.world

El siguiente código se corresponde con el archivo utilizado para definir el mundo simulado (stage.world).

```
#*********************************************************************************//
# NOMBRE: stage.world                                                            //
# AUTOR: Inés Sanz Alonso                                                        //
# FUNCIÓN: Archivo para crear el mundo simulado                                  //
# FECHA: Junio 08                                                                //
#*********************************************************************************//
# defines Pioneer-like robots
include "pioneer.inc"
# defines 'map' object used for floorplans
include "map.inc"
# defines sick laser scanner
include "sick.inc"
# size of the world in meters
size [9.43 8.38]
# set the resolution of the underlying raytrace model in meters
resolution 0.02
interval_sim 100
interval_real 100
# configure the GUI window
window
(
  size [472.000 420.000]
  center [0 0]
  scale 0.022
)
# load an environment bitmap
map
(
  bitmap "espacio.PNG"
  size [9.43 8.38]
  name "espacio"
)
# create a robot
pioneer2dx
(
  name "robot1"
  color "green"
  pose [0 0 0] # Angulo en grados
  sick_laser()
  watchdog_timeout -1.0
)
# creamos varios robot-obstaculo
pioneer2dx
(
  name "robot2"
  color "red"
  pose [-3 -3 0]
  sick_laser()
  watchdog_timeout -1.0
)
pioneer2dx
(
  name "robot3"
  color "grey"
  pose [2 -3 90]
  sick_laser()
  watchdog_timeout -1.0
)
pioneer2dx
(
  name "robot4"
  color "orange"
  pose [2.5 0.5 -135]
  sick_laser()
  watchdog_timeout -1.0
)
pioneer2dx
(
  name "robot5"
  color "yellow"
  pose [-2.75 2 -45]
  sick_laser()
  watchdog_timeout -1.0
)
```

b) pioneer.inc

El siguiente código corresponde con el archivo para definir el modelo del robot Pioneer (pioneer.inc).

```
#*********************************************************************************//
# NOMBRE: pionner.inc                                                            //
# AUTOR: Andrew Howard, Richard Vaughan (Player Proyect)                         //
# FUNCIÓN: Definición del modelo de los robots pioneer                           //
# FECHA: Jun 02                                                                  //
#*********************************************************************************//
# The Pioneer2DX sonar array
define p2dx_sonar ranger
(
  scount 16
  # define the pose of each transducer [xpos ypos heading]
  spose[0] [ 0.075 0.130 90 ]
  spose[1] [ 0.115 0.115 50 ]
  spose[2] [ 0.150 0.080 30 ]
  spose[3] [ 0.170 0.025 10 ]
  spose[4] [ 0.170 -0.025 -10 ]
  spose[5] [ 0.150 -0.080 -30 ]
  spose[6] [ 0.115 -0.115 -50 ]
  spose[7] [ 0.075 -0.130 -90 ]
  spose[8] [ -0.155 -0.130 -90 ]
  spose[9] [ -0.195 -0.115 -130 ]
  spose[10] [ -0.230 -0.080 -150 ]
  spose[11] [ -0.250 -0.025 -170 ]
  spose[12] [ -0.250 0.025 170 ]
  spose[13] [ -0.230 0.080 150 ]
  spose[14] [ -0.195 0.115 130 ]
  spose[15] [ -0.155 0.130 90 ]
  # define the field of view of each transducer [range_min range_max view_angle]
  sview [0 5.0 15]
  # define the size of each transducer [xsize ysize] in meters
  ssize [0.01 0.05]
)
# a Pioneer 2 or 3 in standard configuration
define pioneer2dx position
(
  # actual size
  size [0.44 0.33]
  # the pioneer's center of rotation is offset from its center of area
  origin [-0.04 0.0 0]
  # draw a nose on the robot so we can see which way it points
  gui_nose 1
  # estimated mass in KG
  mass 15.0
  # this polygon approximates the shape of a pioneer
  polygons 1
  polygon[0].points 8
  polygon[0].point[0] [  0.23  0.05 ]
  polygon[0].point[1] [  0.15  0.15 ]
  polygon[0].point[2] [ -0.15  0.15 ]
  polygon[0].point[3] [ -0.23  0.05 ]
  polygon[0].point[4] [ -0.23 -0.05 ]
  polygon[0].point[5] [ -0.15 -0.15 ]
  polygon[0].point[6] [  0.15 -0.15 ]
  polygon[0].point[7] [  0.23 -0.05 ]
  # differential steering model
  drive "diff"
  # uncomment this line if you want to model real pioneers with SICK
  # lasers, where the laser is taller than the robot
  # laser_return 0
  # use the sonar array defined above
  p2dx_sonar()
)
#define pioneer2dx_battery pioneer2dx
#(
# power( pose [-0.14 0 0] capacity 1000 probe_range 0.50  give 0 give_rate 100 take_rate
   100 )
#)
# The AmigoBot sonar array
define amigo_sonar ranger
```

```
(
  scount 8
  spose[0] [ 0.073 0.105 90 ]
  spose[1] [ 0.130 0.078 41 ]
  spose[2] [ 0.154 0.030 15 ]
  spose[3] [ 0.154 -0.030 -15 ]
  spose[4] [ 0.130 -0.078 -41 ]
  spose[5] [ 0.073 -0.105 -90 ]
  spose[6] [ -0.146 -0.060 -145 ]
  spose[7] [ -0.146 0.060 145 ]
)
define amigobot position
(
  size [.330 .280]
  #origin [0.0 0.0] # what should this value be? send email to vaughan@sfu.ca.
  amigo_sonar()
)
# define 10 straight bumpers around the edge of the robot
# (these angles are correct for p2dx but the offsets are approximate - RTV)
# format: bumper[x] [x y th length radius] (zero radius gives a straight line)
# WARNING: bumpers are not currently supported by Stage>=1.5
# define pioneer2dxbumper bumper
# (
#   bumpers10
#   bumper[0] [  0.17 -0.22  -52  0.105 0.0 ]
#   bumper[1] [  0.24 -0.12  -19  0.105 0.0 ]
#   bumper[2] [  0.26  0.00    0  0.105 0.0 ]
#   bumper[3] [  0.24  0.12   19  0.105 0.0 ]
#   bumper[4] [  0.17  0.22   52  0.105 0.0 ]
#   bumper[5] [ -0.25  0.22  128  0.105 0.0 ]
#   bumper[6] [ -0.32  0.12  161  0.105 0.0 ]
#   bumper[7] [ -0.34  0.00  180  0.105 0.0 ]
#   bumper[8] [ -0.32 -0.12  199  0.105 0.0 ]
#   bumper[9] [ -0.25 -0.22  232  0.105 0.0 ]
# )
```

c) map.inc

A continuación aparece el código correspondiente con el archivo para definir el modelo del mapa (map.inc).

```
#*********************************************************************************//
# NOMBRE: map.inc                                                                 //
# AUTOR: Player Proyect                                                           //
# FUNCIÓN: Definición del modelo del mapa                                         //
# FECHA: ??                                                                       //
#*********************************************************************************//
define map model
(
  # sombre, sensible, artistic
  color "black"
  # most maps will need a bounding box
  boundary 1
  gui_nose 0
  gui_grid 1
  gui_movemask 0
  gui_outline 0
  gripper_return 0
  fiducial_return 0
)
```

d)sick.inc

A continuación aparece el código correspondiente con el archivo para definir el modelo del laser implementado en el robot simulado de Stage (sick.inc).

```
#*********************************************************************************//
# NOMBRE: sick.inc                                                                //
```

```
# AUTOR: Player Proyect                                                          //
# FUNCIÓN: Definición del modelo del láser del robot simulado                    //
# FECHA: ??                                                                      //
#*******************************************************************************//
define sick_laser laser
(
  range_min 0.0
  range_max 8.0
  fov 180.0
  samples 361
  color "blue"
  size [ 0.14 0.14 ]
)
```

e) stage.cfg

El siguiente código es el del archivo de configuración de Player cuando trabajamos con el robot del simulador Stage (stage.cfg).

```
#*******************************************************************************//
# NOMBRE: stage.cfg                                                              //
# AUTOR: Inés Sanz Alonso                                                        //
# FUNCIÓN: Configuración de los drivers utilizados                               //
# FECHA: Junio 08                                                                //
#*******************************************************************************//
# load the Stage plugin simulation driver
driver
(
  name "stage"
  provides ["simulation:0"]
  plugin "libstageplugin"
  # load the named file into the simulator
  worldfile "wavefront.world"
)
# Export the map
driver
(
  name "stage"
  provides ["map:0" "graphics2d:0"]
  model "espacio"
)
# Create a Stage driver and attach position2d and laser interfaces
# to the model "robot1"
driver
(
  name "stage"
  provides ["6665:position2d:0" "6665:laser:0"]
  model "robot1"
)
#drivers wavefront junto con vfh y amlc
driver
(
  name "amcl"
  provides ["6665:position2d:2" "6665:localize:0"]
  requires ["odometry::6665:position2d:1" "6665:laser:0" "laser::6665:map:0"]
  init_pose [0 0 0]
  init_pose_var [4 4 6.28318]    # el 6,28318 = 2pi
  laser_range_max 8
)
driver
(
  name "vfh"
  provides ["6665:position2d:1"]
  requires ["6665:position2d:0" "6665:laser:0"]
  safety_dist 0.15
  distance_epsilon 0.2
  angle_epsilon 10
)
driver
(
  name "wavefront"
  provides ["6665:planner:0"]
  requires ["output::6665:position2d:1" "input::6665:position2d:2" "6665:localize:0"
```

```
       "map:0"]
  safety_dist 0.15
  distance_epsilon 0.3
  angle_epsilon 20
)
```

VI.1.2. Configuración de los robots reales Pioneer

a) robot.cfg

El archivo que aparece a continuación es el que se utiliza para configurar los parámetros de los drivers de Player cuando trabajamos con los robots reales.

```
#*******************************************************************************//
# NOMBRE: robot.cfg                                                             //
# AUTOR: Inés Sanz Alonso                                                       //
# FUNCIÓN: Configuración de los drivers utilizados                              //
# FECHA: Agosto 08                                                              //
#*******************************************************************************//
#driver del robot
driver
(
  name "p2os"
  provides ["odometry:::position2d:0" "sonar:0" "power:0"]
  port "/dev/ttyUSB0"
)
#driver del laser Hokuyo
driver
(
  name "urglaser"
  provides ["laser:0"]
  port "/dev/ttyACM0"
  pose [0.0 0.0 0.0]
  min_angle -90.0
  max_angle 90.0
  baud 115200
  alwayson 1
)
#driver del mapa
driver
(
  name "mapfile"
  provides ["map:0"]
  resolution 0.02       # 2 cm por pixel
  filename "espacioMod.PNG"
)
#drivers wavefront junto con vfh y amlc
driver
(
  name "amcl"
  provides ["6665:position2d:2" "6665:localize:0"]
  requires ["odometry::6665:position2d:1" "6665:laser:0" "laser::6665:map:0"]
  init_pose [0 0 0]
  init_pose_var [4 4 6.28318]    #2pi
  laser_range_max 4
)
driver
(
  name "vfh"
  provides ["6665:position2d:1"]
  requires ["6665:position2d:0" "6665:laser:0"]
  safety_dist 0.25
  distance_epsilon 0.3
  angle_epsilon 15
)
driver
(
  name "wavefront"
  provides ["6665:planner:0"]
  requires ["output::6665:position2d:1" "input::6665:position2d:2" "6665:localize:0"
       "map:0"]
```

```
  safety_dist 0.25
  distance_epsilon 0.4
  angle_epsilon 25
)
```

VI.2. Códigos de los programas de control de la aplicación

En este apartado se incluyen los códigos de todas las funciones creadas durante la programación de las aplicaciones desarrolladas en este proyecto. Aparecen primero los archivos de la aplicación de control desarrollada en el simulador, y en el siguiente subapartado aparecen los ficheros en los que se encuentran diferencias significativas a los anteriores, las diferencias de menor importancia sólo se comentarán.

VI.2.1. Programa de control trabajando con Stage

a) stage.c

A continuación aparece el código del programa principal de control (stage.c).

```
//*************************************************************************************//
// NOMBRE: stage.c                                                                     //
// AUTOR: Inés Sanz Alonso                                                             //
// FUNCIÓN: Programa principal de control                                              //
// FECHA: Junio 08                                                                     //
//*************************************************************************************//
#include "librerias.h"
// Funcion de reconocimiento del pick del ratón en la ventana1
void opciones(int event, int x, int y, int flags, void* param)
{
 switch(event)
 {
  case CV_EVENT_LBUTTONDOWN:
  if((x>75)&&(x<174)&&(y>50)&&(y<99))       //Coordenadas del boton de conectar
  {
   printf("Se ha seleccionado la opcion CONECTAR\n");
   conectar();
  }
  else if((x>275)&&(x<374)&&(y>50)&&(y<99)) //Coordenadas del boton de desconectar
  {
   printf("Se ha seleccionado la opcion DESCONECTAR\n");
   desconectar();
  }
  else if((x>50)&&(x<199)&&(y>150)&&(y<224)) //Coordenadas del boton de comportamiento1
  {
   printf("Se ha seleccionado el Movimiento Básico\n");
   mover_mediante_wavefront();
  }
  else if((x>250)&&(x<399)&&(y>150)&&(y<224)) //Coordenadas del boton de comportamiento2
  {
   printf("Se ha seleccionado el movimiento por Matriz de Puntos\n");
   mover_con_matriz_introducida();
  }
  else if((x>50)&&(x<199)&&(y>250)&&(y<324))    //Coordenadas del boton de comportamiento3
  {
   printf("Se ha seleccionado Seguir Paredes\n");
   mover_siguiendo_paredes();
  }
  else if((x>250)&&(x<399)&&(y>250)&&(y<324))    //Coordenadas del boton de comportamiento4
  {
   printf("Se ha seleccionado Seguir a una Persona\n");
   seguir_personas();
  }
  else
   printf("Se ha pinchado fuera de los botones, comprobar que no se ha pinchado en el borde
        de un boton\n");
  break;
```

```
 default:
  break;
 }
}
///////////////// MAIN ////////////////
int main(int argc, char** argv)
{
 int salida;
 //Reservar memoria para los puntos y asignarles el color
 puntos_robot=(player_point_2d_t *)malloc(sizeof(player_point_2d_t)*(1)); // (1) punto
 puntos_ruta=(player_point_2d_t *)malloc(sizeof(player_point_2d_t)*(20)); // (20) puntos.
 puntos_ruta2=(player_point_2d_t *)malloc(sizeof(player_point_2d_t)*(20));// (20) puntos.
 color_robot.red=104; color_robot.green=34; color_robot.blue=139;         // Morado
 color_ruta.red=255; color_ruta.green=69; color_ruta.blue=0;              // Naranja
 color_ruta2.red=0; color_ruta2.green=102; color_ruta2.blue=0;            // VerdeOscuro
 //Crear imagen
 img1 = cvCreateImage(cvSize(200,400),IPL_DEPTH_8U,1);
 //Cargar la imagen_ppal
 img1 = cvLoadImage(cadena1,1);
 //Crear ventana1
 cvNamedWindow("Opciones", CV_WINDOW_AUTOSIZE);
 //Mostrar imagen principal
 cvShowImage("Opciones",img1);
 //Cargar la imagen del espacio
 img2 = cvLoadImage(cadena2,1);
 //Crear ventana2
 cvNamedWindow("Ispace", CV_WINDOW_AUTOSIZE);
 //Mostrar imagen del espacio
 cvShowImage("Ispace",img2);
 do
 {
  //Espera al tick del raton en la pantalla principal
  cvSetMouseCallback("Opciones",opciones,NULL);
  salida = cvWaitKey(10);
  if(salida == 115)  // s ->115
   break;
 }while(1);
 //Liberar imágen1 y destruir ventana1
 cvReleaseImage(&img1);
 cvDestroyWindow("Opciones");
 //Liberar imágen2 y destruir ventana2
 cvReleaseImage(&img2);
 cvDestroyWindow("Ispace");
}
```

b) librerias.h

El archivo siguiente corresponde con el ficheros de cabecera que incluye las librerías utilizadas (librerias.h).

```
//*****************************************************************************//
// NOMBRE: librerías.h                                                         //
// AUTOR: Inés Sanz Alonso                                                     //
// FUNCIÓN: Incluye las librerías utilizadas durante la programación de las aplicaciones//
// FECHA: Julio 08                                                             //
//*****************************************************************************//
//Libreria para el manejo de las funciones de player
#include <libplayerc/playerc.h>
// Librerias para meter/sacar datos de la entrada/salida estandar
#include <stdio.h>
#include <stdlib.h>
// Libreria necesaria para poder hacer operaciones matematicas
#include <math.h>
// Librerias OpenCV
#include "cv.h"
#include "highgui.h"
//Librerias propias del proyecto
#include "variables.h"
#include "circulo.h"
#include "conectar_desconectar.h"
#include "mov_basico.h"
#include "conjunto_puntos.h"
#include "seguir_paredes.h"
```

```
#include "seguir_personas.h"
```

c) variables.h

El archivo siguiente corresponde con el ficheros de cabecera que incluye las variables globales utilizadas (variables.h).

```
//*****************************************************************************//
// NOMBRE: variables.h                                                         //
// AUTOR: Inés Sanz Alonso                                                     //
// FUNCIÓN: Variables globales utilizadas en la programación                   //
// FECHA: Junio 08                                                             //
//*****************************************************************************//
#define PI 3.141592
//Definiciones de los parámetros creados para control de comportamientos
#define error_destino 0.35    //metros
#define dist_avance 0.2       //metros
#define dist_aprox 0.4//metros
#define dist_lateral_max 0.55 //metros
#define dist_lateral_min 0.35 //metros
#define angulo_reorientacion 5.0     //grados
#define dist_frontal_min 0.45 //metros
#define dist_esquina_ext 0.8  //metros
#define rango_laser 8.0       //metros
//Definicion de las variables necesarias para el manejo de imagenes
char cadena1[9]="img1.JPG";
char cadena2[12]="espacio.JPG";
char cadena3[11]="basico.JPG";
char cadena4[11]="puntos.JPG";
IplImage *img1, *img2, *img3, *img4, *img5, *img6;
//Variables para pintar en las ventanas creadas
CvPoint centro_circulo, centro_circulo2, pto_ruta, pto_ruta1;
int x_centro, y_centro, x_centro2, y_centro2, x_pto, y_pto, x_pto1, y_pto1;
//Variables generales
double new_x, new_y, new_theta;
// Variables para el comportamiento de "conjunto_puntos"
double aux_x, aux_y;
int n_puntos=0;
int fin_ptos = 0;
//Variables para el comportamiento de seguir paredes
int menor_sensor1, menor_sensor2;
double menor_distancia1, menor_distancia2;
double r; // r es la distancia que detecta cada sensor
//Variables para el comportamiento "Seguir Personas"
double distancia_cercana;
int sensor_cercano;
//Crear las estructuras necesarias para suscribirse a los servicios de player
playerc_client_t *client;
playerc_position2d_t *position2d_robot;
playerc_position2d_t *position2d_vfhprovides;
playerc_position2d_t *position2d_inputwave;
playerc_laser_t *laser;
playerc_map_t *mapa;
playerc_planner_t *planner;
playerc_localize_t *localize;
//Estructuras necesarias para dibujar en la ventana de Stage
playerc_graphics2d_t *graficos_robot;
playerc_graphics2d_t *graficos_ruta;
playerc_graphics2d_t *graficos_ruta2;
player_point_2d_t *puntos_robot;
player_point_2d_t *puntos_ruta;
player_point_2d_t *puntos_ruta2;
player_color_t color_robot;
player_color_t color_ruta;
player_color_t color_ruta2;
```

d) conectar_desconectar.h

A continuación se presenta el código de las funciones que sirven para conectar y desconectar el cliente y el resto de dispositivos necesarios al servidor.

```
//*************************************************************************************//
// NOMBRE: conectar_desconectar.h                                                      //
// AUTOR: Inés Sanz Alonso                                                             //
// FUNCIÓN: Funciones de conexión y desconexión del cliente y dispositivos al servidor //
// FECHA: Julio 08                                                                     //
//*************************************************************************************//
void conectar(void);
void conectar_local(void);
void desconectar(void);
void desconectar_local(void);
//Funcion conectar
void conectar(void)
{
 // Crear un cliente y conectarlo al servidor
 client = playerc_client_create(NULL, "localhost", 6665);
 if (playerc_client_connect(client) != 0)
 {
  fprintf(stderr, "error: %s\n", playerc_error_str());
 }
 // Cambiar a modo PULL
 if (playerc_client_datamode (client, PLAYERC_DATAMODE_PULL) != 0)
 {
  fprintf(stderr, "error: %s\n", playerc_error_str());
 }
 // Reglas para la gestion de la cola de clientes
 if (playerc_client_set_replace_rule (client, -1, -1, PLAYER_MSGTYPE_DATA, -1, 1) != 0)
 {
  fprintf(stderr, "error: %s\n", playerc_error_str());
 }
 // Suscribirse a position2d:0
 position2d_robot = playerc_position2d_create(client, 0);
 if (playerc_position2d_subscribe(position2d_robot, PLAYER_OPEN_MODE))
 {
  fprintf(stderr, "error: %s\n", playerc_error_str());
 }
 // Suscribirse a position2d:1
 position2d_vfhprovides = playerc_position2d_create(client, 1);
 if (playerc_position2d_subscribe(position2d_vfhprovides, PLAYER_OPEN_MODE))
 {
  fprintf(stderr, "error: %s\n", playerc_error_str());
 }
 // Suscribirse a position2d:2
 position2d_inputwave = playerc_position2d_create(client, 2);
 if (playerc_position2d_subscribe(position2d_inputwave, PLAYER_OPEN_MODE))
 {
  fprintf(stderr, "error: %s\n", playerc_error_str());
 }
 // Suscribirse a laser:0
 laser = playerc_laser_create(client, 0);
 if (playerc_laser_subscribe(laser, PLAYER_OPEN_MODE))
 {
  fprintf(stderr, "error: %s\n", playerc_error_str());
 }
 // Suscribirse a map:0
 mapa = playerc_map_create(client, 0);
 if (playerc_map_subscribe(mapa, PLAYER_OPEN_MODE))
 {
  fprintf(stderr, "error: %s\n", playerc_error_str());
 }
 // Suscribirse a planner:0
 planner = playerc_planner_create(client, 0);
 if (playerc_planner_subscribe(planner, PLAYER_OPEN_MODE))
 {
  fprintf(stderr, "error: %s\n", playerc_error_str());
 }
 // Suscribirse a localize:0
 localize = playerc_localize_create(client, 0);
 if (playerc_localize_subscribe(localize, PLAYER_OPEN_MODE))
 {
```

```
 fprintf(stderr, "error: %s\n", playerc_error_str());
}
// Suscribirse a los sistemas para representacion grafica
graficos_robot = playerc_graphics2d_create(client, 0);
if (playerc_graphics2d_subscribe(graficos_robot, PLAYER_OPEN_MODE) != 0)
{
 fprintf(stderr, "error: %s\n", playerc_error_str());
}
graficos_ruta = playerc_graphics2d_create(client, 0);
if (playerc_graphics2d_subscribe(graficos_ruta, PLAYER_OPEN_MODE) != 0)
{
 fprintf(stderr, "error: %s\n", playerc_error_str());
}
graficos_ruta2 = playerc_graphics2d_create(client, 0);
if (playerc_graphics2d_subscribe(graficos_ruta2, PLAYER_OPEN_MODE) != 0)
{
 fprintf(stderr, "error: %s\n", playerc_error_str());
}
// Establecer los colores
playerc_graphics2d_setcolor (graficos_robot, color_robot);
playerc_graphics2d_setcolor (graficos_ruta, color_ruta);
playerc_graphics2d_setcolor (graficos_ruta, color_ruta2);
// Borramos la pantalla
playerc_graphics2d_clear(graficos_robot);
playerc_graphics2d_clear(graficos_ruta);
playerc_graphics2d_clear(graficos_ruta2);
// Habilitar motores
playerc_position2d_enable(position2d_robot,1);
//Habilitar movimiento del robot
playerc_planner_enable(planner,1);
}
//Funcion conectar para posicionamiento local
void conectar_local(void)
{
// Crear un cliente y conectarlo al servidor
client = playerc_client_create(NULL, "localhost", 6665);
if (playerc_client_connect(client) != 0)
{
 fprintf(stderr, "error: %s\n", playerc_error_str());
}
// Cambiar a modo PULL
if (playerc_client_datamode (client, PLAYERC_DATAMODE_PULL) != 0)
{
 fprintf(stderr, "error: %s\n", playerc_error_str());
}
// Reglas para la gestion de la cola de clientes
if (playerc_client_set_replace_rule (client, -1, -1, PLAYER_MSGTYPE_DATA, -1, 1) != 0)
{
 fprintf(stderr, "error: %s\n", playerc_error_str());
}
// Suscribirse a position2d:0
position2d_robot = playerc_position2d_create(client, 0);
if (playerc_position2d_subscribe(position2d_robot, PLAYER_OPEN_MODE))
{
 fprintf(stderr, "error: %s\n", playerc_error_str());
}
// Suscribirse a laser:0
laser = playerc_laser_create(client, 0);
if (playerc_laser_subscribe(laser, PLAYER_OPEN_MODE))
{
 fprintf(stderr, "error: %s\n", playerc_error_str());
}
// Suscribirse a los sistemas para representacion grafica
graficos_robot = playerc_graphics2d_create(client, 0);
if (playerc_graphics2d_subscribe(graficos_robot, PLAYER_OPEN_MODE) != 0)
{
 fprintf(stderr, "error: %s\n", playerc_error_str());
}
graficos_ruta = playerc_graphics2d_create(client, 0);
if (playerc_graphics2d_subscribe(graficos_ruta, PLAYER_OPEN_MODE) != 0)
{
 fprintf(stderr, "error: %s\n", playerc_error_str());
}
// Establecer los colores
playerc_graphics2d_setcolor (graficos_robot, color_robot);
playerc_graphics2d_setcolor (graficos_ruta, color_ruta);
playerc_graphics2d_setcolor (graficos_ruta, color_ruta2);
```

```
 // Borramos la pantalla
 playerc_graphics2d_clear(graficos_robot);
 playerc_graphics2d_clear(graficos_ruta);
 playerc_graphics2d_clear(graficos_ruta2);
 // Habilitar motores
 playerc_position2d_enable(position2d_robot,1);
}
//Funcion desconectar
void desconectar(void)
{
 //Se retira la suscripcion a todos los servicios
 playerc_position2d_unsubscribe(position2d_robot);
 playerc_position2d_destroy(position2d_robot);
 playerc_position2d_unsubscribe(position2d_vfhprovides);
 playerc_position2d_destroy(position2d_vfhprovides);
 playerc_position2d_unsubscribe(position2d_inputwave);
 playerc_position2d_destroy(position2d_inputwave);
 playerc_laser_unsubscribe(laser);
 playerc_laser_destroy(laser);
 playerc_map_unsubscribe(mapa);
 playerc_map_destroy(mapa);
 playerc_planner_unsubscribe(planner);
 playerc_planner_destroy(planner);
 playerc_localize_unsubscribe(localize);
 playerc_localize_destroy(localize);
 playerc_graphics2d_unsubscribe(graficos_robot);
 playerc_graphics2d_destroy(graficos_robot);
 playerc_graphics2d_unsubscribe(graficos_ruta);
 playerc_graphics2d_destroy(graficos_ruta);
 playerc_graphics2d_unsubscribe(graficos_ruta2);
 playerc_graphics2d_destroy(graficos_ruta2);
 //Desconectar el cliente
 playerc_client_disconnect(client);  playerc_client_destroy(client);
 printf("Todos los servicios y el cliente han sido desconectados, para cerrar la ventana
       pulsar la tecla 's'\n");
}
void desconectar_local(void)
{
 playerc_position2d_unsubscribe(position2d_robot);
 playerc_position2d_destroy(position2d_robot);
 playerc_laser_unsubscribe(laser);
 playerc_laser_destroy(laser);
 playerc_graphics2d_unsubscribe(graficos_robot);
 playerc_graphics2d_destroy(graficos_robot);
 playerc_graphics2d_unsubscribe(graficos_ruta);
 playerc_graphics2d_destroy(graficos_ruta);
 playerc_graphics2d_unsubscribe(graficos_ruta2);
 playerc_graphics2d_destroy(graficos_ruta2);
 playerc_client_disconnect(client);  playerc_client_destroy(client);
}
```

e) mov_basico.h

El siguiente código corresponde con la función de manejo del comportamiento "Movimiento Básico".

```
//*****************************************************************************//
// NOMBRE: mov_basico.h                                                        //
// AUTOR: Inés Sanz Alonso                                                     //
// FUNCIÓN: Funciones de control del comportamiento "Movimiento Básico"        //
// FECHA: Julio 08                                                             //
//*****************************************************************************//
void posicion_meta(int event, int x, int y, int flags, void* param);
void orientacion(int event, int x, int y, int flags, void* param);
void mover_mediante_wavefront(void);
//Funcion para coger el punto meta (x,y)
void posicion_meta(int event, int x, int y, int flags, void* param)
{
 switch(event)
 {
  case CV_EVENT_LBUTTONDOWN:
   if((x>0)&&(x<471)&&(y>0)&&(y<419))      //Coordenadas que delimitan el mapa
   {
```

```
    new_x = (double)(x-236)/50;
    new_y = (double)(210-y)/50;
   }
   else
    printf("Pinchar en una zona valida del mapa\n");
   break;
  default:
   break;
 }
}
//Funcion para coger la orientacion que se quiere dar al robot
void orientacion(int event, int x, int y, int flags, void* param)
{
 switch(event)
 {
  case CV_EVENT_LBUTTONDOWN:
   if((x>523)&&(x<546)&&(y>75)&&(y<100))      //Coordenadas del boton de theta = 0
   {
    new_theta = 0;
   }
   else if((x>498)&&(x<521)&&(y>50)&&(y<75))      //Coordenadas del boton de theta = 90
   {
    new_theta = PI/2;
   }
   else if((x>473)&&(x<496)&&(y>75)&&(y<100))    //Coordenadas del boton de theta = 180
   {
    new_theta = PI;
   }
   else if((x>498)&&(x<521)&&(y>100)&&(y<125))    //Coordenadas del boton de theta = 270
   {
    new_theta = (3*PI/2);
   }
   else if((x>498)&&(x<521)&&(y>75)&&(y<100))  // Boton de mantener el mismo theta
   {
    playerc_client_read(client);
    new_theta = position2d_robot->pa;
   }
   else
    printf("Se ha pinchado fuera de los botones, comprobar que no se ha pinchado en el
        borde de un boton\n");
   break;
  default:
   break;
 }
}
//Funcion para mover el robot mediante el algoritmo Wavefront
void mover_mediante_wavefront(void)
{
 double x_aux, y_aux, theta_aux;
 int i, j=0, ptos;
 int tecla_q;
 //Cargar imagen
 img3 = cvCreateImage(cvSize(547,420),IPL_DEPTH_8U,1);
 img3 = cvLoadImage(cadena3,1);
 //Crear ventana3
 cvNamedWindow("MovimientoBasico", CV_WINDOW_AUTOSIZE);
 //Mostrar imagen del espacio
 cvShowImage("MovimientoBasico",img3);
 //Lectura de la posicion actual del robot
 playerc_client_read(client);
 printf("Posicion actual (odometria): x = %f, y = %f, theta = %f\n", position2d_robot->px,
       position2d_robot->py, position2d_robot->pa);
 printf("Posicion media (amcl): x = %f, y = %f, theta = %f\n", position2d_inputwave->px,
       position2d_inputwave->py, position2d_inputwave->pa);
 printf("Seleccionar primero un punto del mapa y despues una orientacion, y pulsar una
       tecla cualquiera\n");
 //Espera al tick del raton en la pantalla principal
 cvSetMouseCallback("MovimientoBasico",posicion_meta,NULL);
 cvWaitKey(0);
 cvSetMouseCallback("MovimientoBasico",orientacion,NULL);
 cvWaitKey(0);
 //Confirmacion del punto pinchado
 printf("Punto pinchado: x = %f, y = %f, theta = %f\n", new_x, new_y, new_theta);
 //Se dibuja el destino en la pantalla de stage
 DibujaCirculo(new_x, new_y, 0.04, color_ruta2);
 //Se dibuja el destino en la ventana
 x_centro2 = (int)((new_x*50)+236);
```

```
 y_centro2 = (int)(210-(new_y*50));
 centro_circulo2 = cvPoint(x_centro2, y_centro2);
 cvCircle(img3, centro_circulo2, 3, CV_RGB(0,102,0), -1, 8, 0);
 cvShowImage("MovimientoBasico",img3);
 //Paso de parámetros al servicio planner
 playerc_planner_set_cmd_pose(planner, new_x, new_y, new_theta);
 //Leer informacion del Planner
 playerc_planner_get_waypoints(planner);
 //Escribir informacion de los numeros intermadios
 printf("Nº de puntos intermedios: %d.\n",planner->waypoint_count);
 ptos = planner->waypoint_count;
 //Bucle de lectura de puntos intermedios
 for(i=0;i< planner->waypoint_count;i++)
 {
  printf("Punto intermedio %d de la ruta: %f, %f, %f.\n",(i+1),
        planner->waypoints[i][0],planner->waypoints[i][1],planner->waypoints[i][2]);
  //Se dibuja el punto en la pantalla de stage
  DibujaCirculo(planner->waypoints[i][0], planner->waypoints[i][1], 0.05, color_ruta);
  //Se dibuja el punto en la ventana
  x_centro = (int)((planner->waypoints[i][0]*50)+236);
  y_centro = (int)(210-(planner->waypoints[i][1]*50));
  centro_circulo = cvPoint(x_centro, y_centro);
  cvCircle(img3, centro_circulo, 4, CV_RGB(255,0,0), 2, 8, 0);
  cvShowImage("MovimientoBasico",img3);
 }
 //Bucle de espera hasta que llegue al punto-meta
 while(sqrt(pow(position2d_inputwave->px - new_x,2.0) + pow(position2d_inputwave->py
        new_y,2.0)) > error_destino)
 {
  playerc_client_read(client);
  // Dibujar la posición actual del robot en la ventana de stage
  puntos_robot[0].px=position2d_inputwave->px;
  puntos_robot[0].py=position2d_inputwave->py;
  playerc_graphics2d_draw_points (graficos_robot, puntos_robot, 1);   //Se dibuja un punto
  // Dibujar la posicion del robot en la ventana
  x_pto = (int)((position2d_inputwave->px*50)+236);
  y_pto = (int)(210-(position2d_inputwave->py*50));
  pto_ruta = cvPoint(x_pto, y_pto);
  cvCircle(img3, pto_ruta, 1, CV_RGB(0,255,0), -1, 8, 0);
  cvShowImage("MovimientoBasico",img3);
 }
 //Llegada del robot a la meta
 printf("\nEl robot ha llegado al punto-meta\n");
 //Pintamos en la ventana principal el punto donde se encuentra
 playerc_client_read(client);
 x_pto1 = (int)((position2d_inputwave->px*50)+236);
 y_pto1 = (int)(210-(position2d_inputwave->py*50));
 pto_ruta1 = cvPoint(x_pto1, y_pto1);
 cvCircle(img2, pto_ruta1, 3, CV_RGB(0,0,255), -1, 8, 0);
 cvShowImage("Ispace",img2);
 do
 {
  tecla_q = cvWaitKey(10);
  if (tecla_q == 113)
   break;
 }
 while (1);
 //Liberar imágen3 y destruir ventana3
 cvReleaseImage(&img3);
 cvDestroyWindow("MovimientoBasico");
}
```

f) conjunto_puntos.h

A continuación aparece el código correspondiente con el manejo del comportamiento “Recorrido de un Conjunto de Puntos”.

```
//*****************************************************************************//
// NOMBRE: conjunto_puntos.h                                                   //
// AUTOR: Inés Sanz Alonso                                                     //
// FUNCIÓN: Funciones de control del comportamiento "Recorrido de un Conjunto de Puntos"//
// FECHA: Julio 08                                                             //
//*****************************************************************************//
```

```
void recoger_punto(int event, int x, int y, int flags, void* param);
void mover_con_matriz_introducida(void);
//Funcion para coger los puntos seleccionados (x, y)
void recoger_punto(int event, int x, int y, int flags, void* param)
{
 switch(event)
 {
  case CV_EVENT_LBUTTONDOWN:
   if((x>0)&&(x<471)&&(y>0)&&(y<419))      //Coordenadas que delimitan el mapa
   {
    aux_x = (double)(x-236)/50;
    aux_y = (double)(210-y)/50;
   }
   else if((x>472)&&(x<545)&&(y>50)&&(y<115))
   {
    fin_ptos = 1;
   }
   else
   printf("Pinchar en una zona valida del mapa\n");
   break;
  default:
   break;
 }
}
//Funcion para mover al robot siguiendo una matriz de puntos previamente introducidos
void mover_con_matriz_introducida(void)
{
 double array_x[10];
 double array_y[10];
 double theta_llegada, x_anterior, y_anterior;
 int k, l, m;
 int tecla, tecla_q;
 //Cargar imagen
 img4 = cvLoadImage(cadena4,1);
 //Crear ventana3
 cvNamedWindow("Conjunto_Puntos", CV_WINDOW_AUTOSIZE);
 //Mostrar imagen del espacio
 cvShowImage("Conjunto_Puntos",img4);
 //Inicializamos la variable que indica que se introdujeron los puntos y su numero
 fin_ptos = 0;
 n_puntos = 0;
 //Leemos la posicion del robot y almacenamos la orientacion en variable auxiliar
 playerc_client_read(client);
 x_anterior = position2d_inputwave->px;
 y_anterior = position2d_inputwave->py;
 //Introducir el numero de puntos por los que se quiere que se pase
 printf("\nEste comportamiento lleva al robot a los puntos que se introduzcan\n");
 printf("Seleccionar en el mapa los puntos a los que se desea llevar el robot (maximo
       10)\n");
 printf("Despues de cada punto pulsar una tecla\nCuando se hayan introducido todos pulsar
       fin y una tecla\n");
 for(k=0;k<=10;k++)
 {
  printf("Seleccionar el punto %d por el que se quiere llevar al robot y pulsar a
continuacion cualquier tecla\n",(k+1));
  //Espera al tick del raton en la pantalla principal
  cvSetMouseCallback("Conjunto_Puntos",recoger_punto,NULL);
  cvWaitKey(0);
  if(fin_ptos == 0)
  {
   array_x[k] = aux_x;
   array_y[k] = aux_y;
   printf("Punto %d: %f, %f\n",(k+1), array_x[k], array_y[k]);
   n_puntos++;
  }
  else
   break;
 }
 printf("Se recogieron los puntos deseados\n");
 for(m=0;m<n_puntos;m++)
 {
  //Dibujamos los puntos-intermedios por donde pasa el robot en la ventana de stage
  DibujaCirculo(array_x[m], array_y[m], 0.04, color_ruta2);
  //Dibujamos tb los puntos en la ventana
  x_centro2 = (int)((array_x[m]*50)+236);  //(int)((array_x[l] + 2.36)*100);
  y_centro2 = (int)(210-(array_y[m]*50));  //(int)((2.10 - array_y[l])*100);
  centro_circulo2 = cvPoint(x_centro2, y_centro2);
```

```
  cvCircle(img4, centro_circulo2, 4, CV_RGB(0,102,0), -1, 8, 0);
  cvShowImage("Conjunto_Puntos",img4);
 }
 for(l=0;l<n_puntos;l++)
 {
  //Hayamos la orientacion con la que tiene que llegar el robot
  new_theta = atan2((array_y[l]-y_anterior), (array_x[l]-x_anterior));
  printf("angulo de llegada: %f\t",theta_llegada);
  printf("desplazamiento de (%f,%f) a (%f,%f)\n",array_x[l],array_y[l],x_anterior,
       y_anterior);
  //Almacenamos las coordenadas actuales para hayar la orientacion siguiente
  x_anterior = array_x[l];
  y_anterior = array_y[l];
  // Movemos el robot a la posicion que corresponda
  playerc_planner_set_cmd_pose(planner, array_x[l], array_y[l], theta_llegada);
  //Dibujamos los puntos-intermedios por donde pasa el robot en la ventana de stage
  DibujaCirculo(array_x[l], array_y[l], 0.05, color_ruta);
  //Dibujamos tb los puntos en la ventana
  x_centro = (int)((array_x[l]*50)+236);  //(int)((array_x[l] + 2.36)*100);
  y_centro = (int)(210-(array_y[l]*50));  //(int)((2.10 - array_y[l])*100);
  centro_circulo = cvPoint(x_centro, y_centro);
  cvCircle(img4, centro_circulo, 5, CV_RGB(255,0,0), 2, 8, 0);
  cvShowImage("Conjunto_Puntos",img4);
  // Bucle de espera hasta que se llegue al punto deseado
  while (sqrt(pow(position2d_inputwave->px - array_x[l],2.0) + pow(position2d_inputwave->py
       - array_y[l],2.0)) > error_destino )
  {
   playerc_client_read(client);
   // Se dibuja la posicion del robot en la ventana de stage
   puntos_robot[0].px=position2d_inputwave->px;
   puntos_robot[0].py=position2d_inputwave->py;
   playerc_graphics2d_draw_points (graficos_robot, puntos_robot, 1);
   // Se dibuja la posicion del robot en la ventana
   x_pto = (int)((position2d_inputwave->px*50)+236);
   y_pto = (int)(210-(position2d_inputwave->py*50));
   pto_ruta = cvPoint(x_pto, y_pto);
   cvCircle(img4, pto_ruta, 1, CV_RGB(0,255,0), -1, 8, 0);
   cvShowImage("Conjunto_Puntos",img4);
  }
  tecla = cvWaitKey(50);
  if(tecla >= 0)
  {
   printf("Se presiono una tecla para salir del comportamiento\n");
   break;
  }
 }
 printf("Se completo el recorrido especificado\n");
 //Pintamos en la ventana principal el punto donde esta el robot
 playerc_client_read(client);
 x_pto1 = (int)((position2d_inputwave->px*50)+236);
 y_pto1 = (int)(210-(position2d_inputwave->py*50));
 pto_ruta1 = cvPoint(x_pto1, y_pto1);
 cvCircle(img2, pto_ruta1, 3, CV_RGB(153,0,102), -1, 8, 0);
 cvShowImage("Ispace",img2);
 do
 {
  tecla_q = cvWaitKey(10);
  if (tecla_q == 113)
   break;
 }
 while (1);
 //Liberar imágen4 y destruir ventana4
 cvReleaseImage(&img4);
 cvDestroyWindow("Conjunto_Puntos");
}
```

g) seguir_paredes.h

El siguiente código corresponde con el control del comportamiento “Seguir las Paredes del Entorno”.

```
//*****************************************************************************//
// NOMBRE: seguir_paredes.h                                                    //
```

```
// AUTOR: Inés Sanz Alonso                                                    //
// FUNCIÓN: Funciones de control del comportamiento "Seguimiento de Paredes"  //
// FECHA: Agosto 08                                                           //
//****************************************************************************//
void buscar_menor(void);
void buscar_menor2(void);
void mover_siguiendo_paredes(void);
//Funcion para buscar el menor cuando ningun punto-sensor ha detectado una pared
void buscar_menor(void)
{
 int i;
 playerc_client_read(client);
 playerc_laser_get_geom (laser);
 for(i=1; i<laser->scan_count; i++)
 {
  // r es la distancia que detecta cada punto sensor
  r = sqrt((pow(laser->point[i].px,2.0 ) + pow(laser->point[i].py,2.0 ) ));
  if(r < menor_distancia1)
  {
   menor_distancia1 = r;
   menor_sensor1 = i;
  }
 }
}
//Funcion para buscar el menor una vez que algun punto-sensor ha detectado una pared
void buscar_menor2(void)
{
 int j;
 playerc_client_read(client);
 playerc_laser_get_geom (laser);
 for(j=1; j<laser->scan_count; j++)
 {
  // r es la distancia que detecta cada punto sensor
  r = sqrt((pow(laser->point[j].px,2.0 ) + pow(laser->point[j].py,2.0 ) ));
  if(r < menor_distancia2)
  {
   menor_distancia2 = r;
   menor_sensor2 = j;
  }
 }
}
//Funcion que controla el comportamiento en el que el robot sigue las paredes hasta que
       completa una vuelta al lugar donde se encuentra
void mover_siguiendo_paredes(void)
{
 int j,k=0, esquina_fuera=0;
 int vuelta=0; // identificador de si ha dado una vuelta completa el robot
 int movimiento=0; //identificador de si se ha movido para cmpletar una vuelta el robot
 int pared_d;  //indicador de si el robot tiene la pared a su derecha(1), a izquierda (0)
 double alfa; //alfa es el angulo que corregimos para orientar el robot hacia la pared
 double beta;  //angulo que hay que corregir pared para situar el robot paralelo
 double vuelta_ini_x, vuelta_ini_y;
 double dist_lateral, dist_lateral2; //distancia sensor-pared cuando circula paralelo
 double dist_frontal;
 int a=0;
 int tecla, tecla_q;
 //Cargar imagen
 img5 = cvLoadImage(cadena2,1);
 //Crear ventana
 cvNamedWindow("SeguirParedes", CV_WINDOW_AUTOSIZE);
 //Mostrar imagen del espacio
 cvShowImage("SeguirParedes",img5);
 //conectamos para posicionamiento local
 desconectar();
 conectar_local();
 playerc_client_read(client);
 playerc_laser_get_geom(laser);
 //Definir el rango del laser
 menor_distancia1=rango_laser;
 menor_distancia2=rango_laser;
 do
 {
  playerc_client_read(client);
  //avanzar con la misma arientacion
  new_theta = position2d_robot->pa;
  new_x = position2d_robot->px + dist_avance * cos(new_theta);
  new_y = position2d_robot->py + dist_avance * sin(new_theta);
```

```
 playerc_position2d_set_cmd_pose(position2d_robot, new_x, new_y, new_theta,1);
 while(((pow(position2d_robot->px - new_x,2.0) + pow(position2d_robot->py - new_y,2.0)) >
      0.05))
 {
  playerc_client_read(client);
  playerc_laser_get_geom (laser);
  // Dibujar la posición actual del robot en la ventana de stage
  puntos_robot[0].px=position2d_robot->px;
  puntos_robot[0].py=position2d_robot->py;
  playerc_graphics2d_draw_points (graficos_robot, puntos_robot, 1);
  // Dibujar la posicion del robot en la ventana
  x_pto = (int)((position2d_robot->px*50)+236);
  y_pto = (int)(210-(position2d_robot->py*50));
  pto_ruta = cvPoint(x_pto, y_pto);
  cvCircle(img5, pto_ruta, 1, CV_RGB(0,255,0), -1, 8, 0);
  cvShowImage("SeguirParedes",img5);
 }
 //llamar a funcion que busque el menor
 buscar_menor();
 printf("\nEl sensor que ha dado una lectura menor ha sido el %d\n",menor_sensor1);
}while(menor_distancia1 >= rango_laser);
printf("\nEl sensor que ha dado una lectura menor ha sido el %d\nse sale del
      bucle\n",menor_sensor1);
//posicionar el robot hacia esa direccion
playerc_client_read(client);
playerc_laser_get_geom (laser);
//Alfa es la orientacion en radianes del punto-sensor mas cercano
alfa = PI/laser->scan_count*menor_sensor1 - (PI/2);
printf("orientacion del sensor mas cercano: %f\n",alfa);
new_theta = position2d_robot->pa + alfa;
printf("nueva orientacion: %f\n",new_theta);
new_x = position2d_robot->px;
new_y = position2d_robot->py;
playerc_position2d_set_cmd_pose(position2d_robot, new_x, new_y, new_theta,1);
while(sqrt(pow(position2d_robot->pa - new_theta,2.0)) > (PI*2/180))
{
 playerc_client_read(client);
 playerc_laser_get_geom (laser);
}
do
{
 //avanzar hacia la direccion del menor
 playerc_client_read(client);
 playerc_laser_get_geom (laser);
 new_theta = position2d_robot->pa;
 new_x = position2d_robot->px + dist_avance * cos(new_theta);
 new_y = position2d_robot->py + dist_avance * sin(new_theta);
 playerc_position2d_set_cmd_pose(position2d_robot, new_x, new_y, new_theta,1);
 while(sqrt((pow(position2d_robot->px - new_x,2.0) + pow(position2d_robot->py –
      new_y,2.0))> 0.02))
 {
  playerc_client_read(client);
  playerc_laser_get_geom (laser);
  // Dibujar la posición actual del robot en la ventana de stage
  puntos_robot[0].px=position2d_robot->px;
  puntos_robot[0].py=position2d_robot->py;
  playerc_graphics2d_draw_points (graficos_robot, puntos_robot, 1);   //Se dibuja un punto
  // Dibujar la posicion del robot en la ventana
  x_pto = (int)((position2d_robot->px*50)+236);
  y_pto = (int)(210-(position2d_robot->py*50));
  pto_ruta = cvPoint(x_pto, y_pto);
  cvCircle(img5, pto_ruta, 1, CV_RGB(0,255,0), -1, 8, 0);
  cvShowImage("SeguirParedes",img5);
 }
 buscar_menor();
 if(menor_distancia1 <= dist_aprox)
  break;
}while(1);
printf("Ha llegado a 40 cm de la pared");
buscar_menor2();
//Evaluamos el angulo que hay que girar
playerc_laser_get_geom (laser);
//Alfa es la orientacion en radianes del punto-sensor mas cercano
alfa = PI/laser->scan_count*menor_sensor2 - (PI/2);
if(alfa<0)
{                          //sumamos angulo positivo (giro antihorario)
 beta = alfa + (PI/2);    // beta es el angulo que sumamos a la orientacion actual
```

```
 pared_d=1;                //La pared esta a la derecha del robot
}
else
{                          //sumamos angulo negativo (giro horario)
 beta = alfa - (PI/2);    // beta es el angulo que sumamos a la orientacion actual
 pared_d=0;                //La pared esta a la izquierda del robot
}
//orientamos el robot paralelo a la pared
playerc_client_read(client);
new_theta = position2d_robot->pa + beta;
new_x = position2d_robot->px;
new_y = position2d_robot->py;
playerc_position2d_set_cmd_pose(position2d_robot, new_x, new_y, new_theta,1);
while(sqrt(pow(position2d_robot->pa - new_theta,2.0)) > (PI*2/180))
{
 playerc_client_read(client);
 playerc_laser_get_geom (laser);
}
printf("\nComienza a dar una vuelta siguiendo las paredes\n");
playerc_client_read(client);
vuelta_ini_x = position2d_robot->px;
vuelta_ini_y = position2d_robot->py;
do
{
 if(esquina_fuera==1)
 {
  k++;
  if(k==100)
  {
   esquina_fuera=0;
   k=0;
  }
 }
 //Mover paralelo a pared
 playerc_client_read(client);
 //hacemos que el robot se mueva paralelo a la pared
 new_theta = position2d_robot->pa;
 new_x = position2d_robot->px + dist_avance * cos(new_theta);
 new_y = position2d_robot->py + dist_avance * sin(new_theta);
 playerc_position2d_set_cmd_pose(position2d_robot, new_x, new_y, new_theta,1);
 while(((pow(position2d_robot->px - new_x,2.0) + pow(position2d_robot->py - new_y,2.0))>
      0.5))
 {
  playerc_client_read(client);
  playerc_laser_get_geom (laser);
  // Dibujar la posición actual del robot en la ventana de stage
  puntos_robot[0].px=position2d_robot->px;
  puntos_robot[0].py=position2d_robot->py;
  playerc_graphics2d_draw_points (graficos_robot, puntos_robot, 1);
  // Dibujar la posicion del robot en la ventana
  x_pto = (int)((position2d_robot->px*50)+236);
  y_pto = (int)(210-(position2d_robot->py*50));
  pto_ruta = cvPoint(x_pto, y_pto);
  cvCircle(img5, pto_ruta, 1, CV_RGB(0,255,0), -1, 8, 0);
  cvShowImage("SeguirParedes",img5);
 }
 //Comprobar que el rumbo es correcto y aplicar modificaciones necesarias
 playerc_client_read(client);
 playerc_laser_get_geom (laser);
 dist_frontal = sqrt((pow(laser->point[laser->scan_count / 2].px,2.0 ) +
      pow(laser->point[laser->scan_count / 2].py,2.0 ) ));
 if(pared_d == 0)  //Pared a la izquierda --> ultimo punto-sensor evaluado = 360
 {
  if(dist_frontal <= dist_frontal_min)  //Esquina interior
  {
   new_theta = position2d_robot->pa - (PI/2);   // theta anterior - 90º
  }
  dist_lateral = sqrt((pow(laser->point[laser->scan_count - 1].px,2.0 ) +
      pow(laser->point[laser->scan_count - 1].py,2.0 ) ));
  dist_lateral2 = sqrt((pow(laser->point[laser->scan_count - 20].px,2.0 ) +
      pow(laser->point[laser->scan_count - 20].py,2.0 ) ));
  if((dist_lateral >= dist_esquina_ext)&&(esquina_fuera==0)) //Esquina exterior
  {
   playerc_client_read(client);
   new_theta = position2d_robot->pa;
   new_x = position2d_robot->px + 0.5*cos(new_theta);
   new_y = position2d_robot->py + 0.5*sin(new_theta);
```

```
   playerc_position2d_set_cmd_pose(position2d_robot, new_x, new_y, new_theta,1);
   while(((pow(position2d_robot->px - new_x,2.0) + pow(position2d_robot->py - new_y,2.0))>
      0.05))
   {
    playerc_client_read(client);
    playerc_laser_get_geom (laser);
    // Dibujar la posición actual del robot en la ventana de stage
    puntos_robot[0].px=position2d_robot->px;
    puntos_robot[0].py=position2d_robot->py;
    playerc_graphics2d_draw_points (graficos_robot, puntos_robot, 1);
    // Dibujar la posicion del robot en la ventana
    x_pto = (int)((position2d_robot->px*50)+236);
    y_pto = (int)(210-(position2d_robot->py*50));
    pto_ruta = cvPoint(x_pto, y_pto);
    cvCircle(img5, pto_ruta, 1, CV_RGB(0,255,0), -1, 8, 0);
    cvShowImage("SeguirParedes",img5);
   }
   new_theta = position2d_robot->pa + (PI/2);   // theta anterior + 90º
   esquina_fuera=1;
  }
  if((((dist_lateral-dist_lateral2) >= 0.001)||(dist_lateral2<dist_lateral_min))&&
      (dist_lateral<dist_esquina_ext)&&(dist_lateral2<dist_esquina_ext)&&
      (dist_frontal > dist_frontal_min))
  {                                          //Si se acerca demasiado a la pared
   new_theta = position2d_robot->pa - (PI*angulo_reorientacion/180);// theta anterior - 5º
  }
  if((((dist_lateral2-dist_lateral) >= 0.005)||(dist_lateral2>dist_lateral_max))&&
      (dist_lateral<dist_esquina_ext)&&(dist_lateral2<dist_esquina_ext)&&
      (dist_frontal > dist_frontal_min))
  {                                           //Si se aleja un poco de la pared
   new_theta = position2d_robot->pa + (PI*angulo_reorientacion/180);// theta anterior + 5º
  }
 }
 if(pared_d == 1)     //Pared a la derecha --> punto-sensor evaluado es el primero = 1
 {
  if(dist_frontal <= dist_frontal_min) // Esquina interior
  {
   new_theta = position2d_robot->pa + (PI/2);   // theta anterior + 90º
  }
  dist_lateral = sqrt((pow(laser->point[1].px,2.0 ) + pow(laser->point[1].py,2.0 ) ));
  dist_lateral2 = sqrt((pow(laser->point[20].px,2.0 ) + pow(laser->point[20].py,2.0 ) ));
  if((dist_lateral >= dist_esquina_ext)&&(esquina_fuera==0)) //Esquina exterior
  {
   playerc_client_read(client);
   new_theta = position2d_robot->pa;
   new_x = position2d_robot->px + 0.5*cos(new_theta);
   new_y = position2d_robot->py + 0.5*sin(new_theta);
   playerc_position2d_set_cmd_pose(position2d_robot, new_x, new_y, new_theta,1);
   while(((pow(position2d_robot->px - new_x,2.0) + pow(position2d_robot->py - new_y,2.0))>
      0.05))
   {
    playerc_client_read(client);
    playerc_laser_get_geom (laser);
    // Dibujar la posición actual del robot en la ventana de stage
    puntos_robot[0].px=position2d_robot->px;
    puntos_robot[0].py=position2d_robot->py;
    playerc_graphics2d_draw_points (graficos_robot, puntos_robot, 1);
    // Dibujar la posicion del robot en la ventana
    x_pto = (int)((position2d_robot->px*50)+236);
    y_pto = (int)(210-(position2d_robot->py*50));
    pto_ruta = cvPoint(x_pto, y_pto);
    cvCircle(img5, pto_ruta, 1, CV_RGB(0,255,0), -1, 8, 0);
    cvShowImage("SeguirParedes",img5);
   }
   new_theta = position2d_robot->pa - (PI/2);   // theta anterior - 90º
   esquina_fuera=1;
  }
  if((((dist_lateral-dist_lateral2) >= 0.001)||(dist_lateral2<dist_lateral_min))&&
      (dist_lateral<dist_esquina_ext)&&(dist_lateral2<dist_esquina_ext)&&
      (dist_frontal > dist_frontal_min))
  {                                           //Si se acerca demasiado a la pared
   new_theta = position2d_robot->pa + (PI*angulo_reorientacion/180);// theta anterior + 5º
  }
  if((((dist_lateral2-dist_lateral) >= 0.005)||(dist_lateral2>dist_lateral_max))&&
      (dist_lateral<dist_esquina_ext)&&(dist_lateral2<dist_esquina_ext)&&
      (dist_frontal > dist_frontal_min))
  {                                           //Si se aleja un poco de la pared
```

```
    new_theta = position2d_robot->pa - (PI*angulo_reorientacion/180);// theta anterior - 5º
   }
  }
  playerc_client_read(client);
  playerc_laser_get_geom (laser);
  new_x = position2d_robot->px;
  new_y = position2d_robot->py;
  playerc_position2d_set_cmd_pose(position2d_robot, new_x, new_y, new_theta,1);
  // Dibujar la posición actual del robot en la ventana de stage
  puntos_robot[0].px=position2d_robot->px;
  puntos_robot[0].py=position2d_robot->py;
  playerc_graphics2d_draw_points (graficos_robot, puntos_robot, 1);   //Se dibuja un punto
  // Dibujar la posicion del robot en la ventana
  x_pto = (int)((position2d_robot->px*50)+236);
  y_pto = (int)(210-(position2d_robot->py*50));
  pto_ruta = cvPoint(x_pto, y_pto);
  cvCircle(img5, pto_ruta, 1, CV_RGB(0,255,0), -1, 8, 0);
  cvShowImage("SeguirParedes",img5);
  a=0;
  while(sqrt(pow(position2d_robot->pa - new_theta,2.0)) > (PI*4/180))
  {
   playerc_client_read(client);
   playerc_laser_get_geom (laser);
   a++;
   if(a>=100)
   {
    a=0;
    break;
   }
  }
  //Comprobar si ha dado una vuelta completa
  playerc_client_read(client);
  if(sqrt((pow(position2d_robot->px - vuelta_ini_x,2.0) + pow(position2d_robot->py –
       vuelta_ini_y,2.0))> 0.4))
  {
   movimiento=1;
  }
  if( (sqrt((pow(position2d_robot->px - vuelta_ini_x,2.0) + pow(position2d_robot->py –
       vuelta_ini_y,2.0))< error_destino)) && (movimiento==1) )
  {
   vuelta=1;
   printf("se completo una vuelta\n");
  }
  tecla = cvWaitKey(50);
  if(tecla >= 0)
  {
   printf("Se presiono una tecla para salir del comportamiento");
   break;
  }
 }while(vuelta!=1);
 //desconectamos para posicionamiento local
 desconectar_local();
 conectar();
 do
 {
  tecla_q = cvWaitKey(10);
  if (tecla_q == 113)
   break;
 }
 while (1);
 //Liberar imágen5 y destruir ventana5
 cvReleaseImage(&img5);
 cvDestroyWindow("SeguirParedes");
}
```

h) seguir_personas.h

El siguiente código corresponde con el control del comportamiento “Seguimiento de Personas”.

```
//*****************************************************************************//
// NOMBRE: seguir_personas.h                                                   //
// AUTOR: Inés Sanz Alonso                                                     //
```

```
// FUNCIÓN: Funciones de control del comportamiento "Seguimiento de Personas"           //
// FECHA: Agosto-Septiembre 08                                                          //
//**************************************************************************************//
void buscar_mas_cercano(void);
void seguir_personas (void);
//Funcion que busca el punto-sensor que detecta el objeto mas cercano
void buscar_mas_cercano(void)
{
 int i;
 double r;
 playerc_client_read(client);
 playerc_laser_get_geom (laser);
 for(i=0; i<laser->scan_count; i++)
 {
  // r es la distancia que detecta cada punto sensor
  r = sqrt((pow(laser->point[i].px,2.0 ) + pow(laser->point[i].py,2.0 ) ));
  if(r < distancia_cercana)
  {
   distancia_cercana = r;
   sensor_cercano = i;
  }
 }
}
//Funcion que regula el comportamiento Seguir Personas
void seguir_personas (void)
{
 double x_rot, y_rot, x_tras, y_tras;       //Coordenadas del obstaculo
 int tecla, tecla_q;
 //Cargar imagen
 img6 = cvLoadImage(cadena2,1);
 //Crear ventana3
 cvNamedWindow("SeguirPersonas", CV_WINDOW_AUTOSIZE);
 //Mostrar imagen del espacio
 cvShowImage("SeguirPersonas",img6);
 do
 {
  distancia_cercana = rango_laser;
  //Leer informacion del robot
  playerc_client_read(client);
  playerc_laser_get_geom(laser);
  //Buscar sensor más cercano
  buscar_mas_cercano();
  if(distancia_cercana < rango_laser)
  {
   //Rotación del SR del láser
   x_rot = (cos(position2d_inputwave->pa) * laser->point[sensor_cercano].px) -
        (sin(position2d_inputwave->pa) * laser->point[sensor_cercano].py);
   y_rot = (sin(position2d_inputwave->pa) * laser->point[sensor_cercano].px) +
        (cos(position2d_inputwave->pa) * laser->point[sensor_cercano].py);
   //Traslación del SR del láser
   x_tras = x_rot + position2d_inputwave->px;
   y_tras = y_rot + position2d_inputwave->py;
   new_theta = atan2(y_rot, x_rot);
   new_x = x_tras - (0.1 * cos(new_theta));
   new_y = y_tras - (0.1 * sin(new_theta));
  }
  else
  {
   new_theta = position2d_inputwave->pa;
   new_x = position2d_inputwave->px + 1*cos(new_theta);
   new_y = position2d_inputwave->py + 1*sin(new_theta);
  }
  playerc_planner_set_cmd_pose(planner, new_x, new_y, new_theta);
   playerc_client_read(client);
   // Dibujar la posición actual del robot en la ventana de stage
   puntos_robot[0].px=position2d_inputwave->px;
   puntos_robot[0].py=position2d_inputwave->py;
   playerc_graphics2d_draw_points (graficos_robot, puntos_robot, 1);
   // Dibujar la posicion del robot en la ventana
   x_pto = (int)((position2d_inputwave->px*50)+236);
   y_pto = (int)(210-(position2d_inputwave->py*50));
   pto_ruta = cvPoint(x_pto, y_pto);
   cvCircle(img6, pto_ruta, 1, CV_RGB(0,255,0), -1, 8, 0);
   cvShowImage("SeguirPersonas",img6);
  tecla = cvWaitKey(50);
 }while(tecla < 0);
 printf("Salida del comportamiento\n");
```

```
 //Pintamos en la ventana principal
 playerc_client_read(client);
 x_pto1 = (int)((position2d_inputwave->px*50)+236);
 y_pto1 = (int)(210-(position2d_inputwave->py*50));
 pto_ruta1 = cvPoint(x_pto1, y_pto1);
 cvCircle(img2, pto_ruta1, 3, CV_RGB(51,255,255), -1, 8, 0);
 cvShowImage("Ispace",img2);
 do
 {
  tecla_q = cvWaitKey(10);
  if (tecla_q == 113)
   break;
 }
 while (1);
 //Liberar imágen6 y destruir ventana6
 cvReleaseImage(&img6);
 cvDestroyWindow("SeguirPersonas");
}
```

i) circulo.h

A continuación se muestra la función utilizada para dibujar cículos en la ventana de Stage por aproximación a polígono.

```
//*************************************************************************//
// NOMBRE: circulo.h                                                       //
// AUTOR: Elena López Guillén                                              //
// FUNCIÓN: Dibujar círculos por aproximación a polígono en la ventana de Stage    //
// FECHA: Mayo 08                                                          //
//*************************************************************************//
void DibujaCirculo(double xc, double yc, double radio, player_color_t color);
//Función para dibujar circulos
void DibujaCirculo(double xc, double yc, double radio, player_color_t color)
{
 double x[20], senos[20], cosenos[20], inc;
 int i;
 player_point_2d_t puntos[20];
 inc=2*PI/20;
 x[0]=0.0;
 for(i=1;i<20;i++)
 {
  x[i]=x[i-1]+inc;
 }
 for(i=0;i<20;i++)
 {
  senos[i]=sin(x[i]);
  cosenos[i]=cos(x[i]);
 }
 for(i=0;i<20;i++)
 {
  puntos[i].px=xc+radio*cosenos[i];
  puntos[i].py=yc+radio*senos[i];
 }
 playerc_graphics2d_setcolor (graficos_ruta, color);
 playerc_graphics2d_draw_polyline (graficos_ruta, puntos, 20);
}
```

VI.2.2. Programa de control trabajando con los robots reales

Trabajando con los robots reales hay que hacer ligeras modificaciones en los códigos utilizados con el simulador. Por ejemplo, en el caso de este proyecto se han hecho más restrictivos cuando se trabaja con los robots para paliar de esa manera posibles errores que se puedan producir trabajando en tiempo real.

En este subapartado sólo se incluyen los ficheros cabecera que se han modificado significativamente. El resto de esos ficheros sólo varían en que no se utilizan las funciones de dibujo en la ventana de Stage, ya que con esta configuración no se usa el simulador.

a) librerias.h

El siguiente código corresponde con el archivo de cabecera donde aparecen las librerías utilizadas cuando trabajamos con los robots físicos.

```
//*****************************************************************************//
// NOMBRE: librerías.h                                                         //
// AUTOR: Inés Sanz Alonso                                                     //
// FUNCIÓN: Incluye las librerías utilizadas durante la programación de las aplicaciones//
// FECHA: Julio 08                                                             //
//*****************************************************************************//
//Libreria para el manejo de las funciones de player
#include <libplayerc/playerc.h>
// Librerias para meter/sacar datos de la entrada/salida estandar
#include <stdio.h>
#include <stdlib.h>
// Libreria necesaria para poder hacer operaciones matematicas
#include <math.h>
// Librerias OpenCV
#include "cv.h"
#include "highgui.h"
//Librerias propias del proyecto
#include "variables.h"
#include "conectar_desconectar.h"
#include "mov_basico.h"
#include "conjunto_puntos.h"
#include "seguir_paredes.h"
#include "seguir_personas.h"
```

b) variables.h

A continuación aparece el código del archivo de cabecera donde aparecen las variables globales que se utilizan cuando trabajamos con los robots reales.

```
//*****************************************************************************//
// NOMBRE: variables.h                                                         //
// AUTOR: Inés Sanz Alonso                                                     //
// FUNCIÓN: Variables globales utilizadas en la programación                   //
// FECHA: Junio 08                                                             //
//*****************************************************************************//
#define PI 3.141592
//Definiciones de los parámetros creados para control de comportamientos
#define error_destino 0.40
#define dist_avance 0.2
#define dist_aprox 0.45
#define dist_lateral_max 0.60
#define dist_lateral_min 0.40
#define angulo_reorientacion 5.0
#define dist_frontal_min 0.50
#define dist_esquina_ext 0.8
#define rango_laser 4.0
//Definicion de las variables necesarias para el manejo de imagenes
char cadena1[9]="img1.JPG";
char cadena2[12]="espacio.JPG";
char cadena3[11]="basico.JPG";
char cadena4[11]="puntos.JPG";
IplImage *img1, *img2, *img3, *img4, *img5, *img6;
//Variables para pintar en las ventanas creadas
CvPoint centro_circulo, centro_circulo2, pto_ruta, pto_ruta1;
int x_centro, y_centro, x_centro2, y_centro2, x_pto, y_pto, x_pto1, y_pto1;
//Variables generales
double new_x, new_y, new_theta;
// Variables para el comportamiento de "conjunto_puntos"
```

```
double aux_x, aux_y;
int n_puntos=0;
int fin_ptos = 0;
//Variables para el comportamiento de seguir paredes
int menor_sensor1, menor_sensor2;
double menor_distancia1, menor_distancia2;
double r; // r es la distancia que detecta cada sensor
//Variables para el comportamiento "Seguir Personas"
double distancia_cercana;
int sensor_cercano;
//Crear las estructuras necesarias para suscribirse a los servicios de player
playerc_client_t *client;
playerc_position2d_t *position2d_robot;
playerc_position2d_t *position2d_vfhprovides;
playerc_position2d_t *position2d_inputwave;
playerc_laser_t *laser;
playerc_map_t *mapa;
playerc_planner_t *planner;
playerc_localize_t *localize;
```

VI.3. Archivos para la compilación de los programas (Makefiles)

En este apartado se incluyen los ficheros para la compilación de los programas desarrollados. Tanto para trabajar con el simulador como con los robots reales, los dos archivos son similares.

VI.3.1. Makefile trabajando con Stage

El archivo para compilar los programas cuando trabajamos con Stage es el siguiente (Makefile).

```
#*****************************************************************************//
# NOMBRE: Makefile                                                            //
# AUTOR: Inés Sanz Alonso                                                     //
# FUNCIÓN: Archivo para la compilación de los ficheros .c y .h utilizando     //
#           las librerías de  Player y las librerías Opencv                   //
# FECHA: Julio 08                                                             //
#*****************************************************************************//
CFLAGS=`pkg-config --cflags opencv`
CFLAGS2=`pkg-config --cflags playerc`
LDFLAGS= `pkg-config --libs opencv`
LDFLAGS2=`pkg-config --libs playerc`
CC=gcc -g
CC2=g++ -g
all:stage.o
	$(CC2) $(CFLAGS) $(CFLAGS2) $(LDFLAGS) $(LDFLAGS2) -o stage stage.o
stage.o: stage.c circulo.h conectar_desconectar.h mov_basico.h conjunto_puntos.h
		seguir_paredes.h seguir_personas.h
	$(CC) $(CFLAGS) $(CFLAGS2) -c stage.c  -o $@
clean:
	-rm *.o
	rm stage
```

VI.3.2. Makefile trabajando con los robots reales

El archivo para compilar los programas cuando trabajamos con Stage es el siguiente (Makefile).

```
#*****************************************************************************//
# NOMBRE: Makefile                                                            //
# AUTOR: Inés Sanz Alonso                                                     //
# FUNCIÓN: Archivo para la compilación de los ficheros .c utilizando las librerías de  //
#          Player y las librerías Opencv                                      //
```

```
# FECHA: Julio 08                                                                    //
#*************************************************************************************//
CFLAGS=`pkg-config --cflags opencv`
CFLAGS2=`pkg-config --cflags playerc`
LDFLAGS= `pkg-config --libs opencv`
LDFLAGS2=`pkg-config --libs playerc`
CC=gcc -g
CC2=g++ -g
all:robot.o
	$(CC2) $(CFLAGS) $(CFLAGS2) $(LDFLAGS) $(LDFLAGS2) -o robot robot.o
robot.o: robot.c conectar_desconectar.h mov_basico.h conjunto_puntos.h seguir_paredes.h
	seguir_personas.h
	$(CC) $(CFLAGS) $(CFLAGS2) -c robot.c  -o $@
clean:
	-rm *.o
	rm robot
```

VII. BIBLIOGRAFÍA

[ActivMedia] Página web de Movile Robots© (antigua ActivMedia©) http://www.activrobots.com/ROBOTS.

[Barraquand90] J. Barraquand and J.C. Latombe. "Controllabilitu of Mobile Robots with Kinematic Constraints", Technical Report No. STAN-CS-90-1317, Departament of Computer Science, Stanford University, 1990

[Borenstein91] J. Borenstein and Y. Koren. "The Vector Field Histogram – Fast obstacle avoidance for mobile robots", IEEE Journal of Robotics and Automation Vol. 7, No 3, June 1991, pp. 278-288.

[Borenstein96] J. Borenstein, H. R. Everett and L. Feng. "Where am I? Sensors and Metods for Mobile Robot Positioning", The University of Michigan 1996.

[Fox03a] Dieter Fox. "Adapting the sample size in particle filters through KLD-sampling", The International Journal of Robotics Research, Vol. 22, No. 12, pp 985-1003, December 2003.

[Fox03b] Dieter Fox, Jeffrey Hightower, Lin Liao, Dirk Schulz and Gaetano Borriello. "Bayesian Filtering for Location Estimation", published by the IEEE CS and IEEE ComSoc. 2003.

[Latombe91] Jean-Claude Latombe. “Robot Motion Planning”, 1991 Kluwer Academic Publishers.

[López08] Elena López Guillén. “Métodos probabilísticos de navegación de robots móviles”, Tema 4 de los apuntes de la asignatura Robótica Móvil del Máster Oficial en Sistemas Electrónicos Avanzados. Sistemas Inteligentes. 2008.

[Minguez04] Javier Minguez and Luis Montano. “Nearness Diagram (ND) Navigation: Collision Avoidance in Troublesome Scenarios”, IEEE Transactions on Robotics and Automation, Vol. 20, No. 1, Feb 2004.

[Ocaña08] Manuel Ocaña Miguel. “Métodos clásicos de navegación de robots móviles”, Tema 3 de los apuntes de la asignatura Robótica Móvil del Máster Oficial en Sistemas Electrónicos Avanzados. Sistemas Inteligentes. 2008.

[Pizarro08] D. Pizarro, M. Marrón, D. Peón, M. Mazo, J.C. García, M.A. Sotelo, E. Santiso. “Robot and obstacles localization and tracking with an external camera ring”. Proceedings of the 2008 IEEE International Conference on Robotics and Automation (ICRA08), ISBN: 978-1-4244-1647-9, pp. 516-521, Pasadena, May 2008.

[PlayerProyect] Geoff Biggs, Toby Collet, Brian Gerkey, Andrew Howard, Nate Koenig, Jordi Polo, Radu Rusu and Richard Vaughan. “The Player Proyect”, http://playerstage.sourceforge.net/player

[Ulrich98] Iwan Ulrich and Johann Borenstein. “VFH+: Reliable obstacle avoidance for fast mobile robots”, Proc. of the IEEE Intl. Conf. on Robotics Automation (ICRA), pages 1572-1577, Leuven Belgium, May 1998.

[URG] Hokuyo Automatic Co. Hojas de Características del láser URG-04LX Hokuyo. http://www.hokuyo-aut.jp.