UNIVERSIDAD REY JUAN CARLOS

Ingeniería Técnica en Informática de Sistemas

Escuela Superior de Ciencias Experimentales y Tecnología

Curso académico 2004-2005

Proyecto Fin de Carrera

Navegación global de un robot usando el método del gradiente.

Tutor: José M. Cañas Plaza

Autor: José Raúl Isado García

Febrero 2.005

A mi abuelo,

que estaría muy orgulloso de poder ver esto

Agradecimentos.

Quiero dar las gracias de manera especial al grupo de robótica de la URJC. En particular a José Mª Cañas por la ayuda y conocimientos facilitados, no exentos de paciencia, sin los que no se podria haber llevado a cabo este proyecto.

A mi familia, por estar apoyándome siempre, y darme todas las facilidades para llevar esto a cabo.

A mis amigos de siempre, Pako, David, Almudena, Gema, Alberto, Rubén.... por aguantarme todo este tiempo.

A mis compañeros de clase:

- Luis y Txuso, mucho más que compañeros, amigos.
- Ana, Arturo, Carlos, Chechu, Chema, Chiqui, Ernesto, Fran, Goyo, Laura, Manu, y *last, but not least*, las porteras ; por los magníficos ratos juntos.

Índice general

Resumen **1**

1. Introducción. **1**
1.1. Navegación de robots 4
1.1.1. Algoritmos de planificación de ruta 5
1.1.2. Algoritmos de navegación local 6
1.2. Grupo de Robótica de la URJC 7

2. Objetivos. **9**
2.1. Objetivos 9
2.2. Requisitos 9
2.3. Metodología 10

3. Plataforma de desarrollo **13**
3.1. Robot Pioneer 13
3.2. Arquitectura y modelo de programación con JDE 14
3.2.1. Servidor Otos 16
3.2.2. Servidor Oculo 16
3.3. Bibliotecas Gridslib y Fuzzylib 17
3.4. Simulador SRIsim 17
3.5. Player/Stage 20
3.5.1. Soporte JDE para Player/Stage 23

4. Descripción informática **24**
4.1. Aproximación al método del gradiente 24
4.2. Diseño General 25
4.3. Esquema *Gradient Planning* 26
4.4. Esquema *Gradient Builder* 28
4.5. Esquema *Follow Grid* 33
4.6. Esquema VFF 37
4.7. Esquema *Guixforms* 40

5. Resultados Experimentales **42**
5.1. Ejecución típica 42
5.2. Campo antiobstáculos 43
5.3. Evitación de obstáculos imprevistos 45
5.4. Mejoras de la navegación 46
5.4.1. Vecindad-1 VS Vecindad-2 47
5.4.2. Control de tracción 48
5.4.3. Limitador de velocidad 50

6. Conclusiones y trabajos futuros **52**
6.1. Conclusiones 52
6.2. Trabajos futuros 54

Índice de figuras

1.1. Los androides R2-D2 (a) y C-3PO (b). 1
1.2. Un brazo P.U.M.A. (a) y un robot cargador (b) 2
1.3. La sonda de exploración marciana Opportunity (a), Minerva, el robot guia (b) y una cosechadora autónoma del proyecto Demeter (c). 3
1.4. El perrito mascota Aibo (a), Asimo (b) y Roomba (c) 4
1.5. Los robots de los que dispone el grupo de Róbotica de la URJC 7

2.1. Modelo en espiral 11

3.1. Robot Pioneer sobre el que se basa el proyecto 13
3.2. Arquitectura software con servidores y clientes de JDE 15
3.3. Arquitectura software del sistema programado con esquemas 15
3.4. Interfaz gráfica de SRIsim 18
3.5. Mapa del lado izquierdo de la 1ª planta del Edificio departamental II para SRIsim 20
3.6. Stage simulando una parte de Madrid con 2 robots 21
3.7. Fichero PNM del departamental II (a), y su representación en *Stage* . . 22
3.8. Implementación del servidor Otos con soporte para Player/Stage 23

4.1. Jerarquía de esquemas del comportamiento 25
4.2. División de cada segmento en pasos. 27
4.3. Pseudocódigo del algoritmo de expansión del frente. 29
4.4. Representación de la expansión del frente en la rejilla. 30
4.5. Ruta mínima generada con frente de obstáculos(a) y sin él (b) 31
4.6. Mapa sin campo (a), Mapa con el campo de los obstáculos propagándose (b-c), Mapa con el campo propagándose (d-j) y Mapa con el campo propagado y la ruta de referencia (c). 32
4.7. Esquema de ángulos para las 24 vecinas de la celda ocupada por el robot 33
4.8. Controlador borroso para la velocidad angular 35
4.9. Zonas de seguridad para VFF:cuadrada(a) y redonda(b) 38
4.10. Visualización de fuerzas que comandan el esquema VFF 39
4.11. Interfaz gráfica de la aplicación 41

5.1. Ejecución típica de navegación. 42
5.2. Mapa con campo nulo (a) y ruta seguida (b). 43
5.3. Mapa con campo alto (a) y ruta seguida (b) 44
5.4. Mapa con campo medio (a) y ruta seguida (b) 44
5.5. Ruta afectada por un obstáculo imprevisto 45
5.6. Ruta afectada por un obstáculo imprevisto 46
5.7. Ruta seguida usando Vecindad 2 (a), y Ruta usando Vecindad 1 (b) . . 47

5.8. Ruta seguida usando Vecindad 2 (a), y Ruta usando Vecindad 1 (b) . . 48
5.9. Ruta seguida con control de tracción (a), y Ruta sin control de tracción (b) 49
5.10. Ruta seguida con control de tracción (a), y Ruta seguida sin control de tracción (b) 49
5.11. Ruta con limitador de velocidad (a), y Ruta sin limitador de velocidad (b) 51
5.12. Ruta seguida con limitador de velocidad (a), y Ruta sin limitador de velocidad (b) 51

Índice de cuadros

5.1. Tabla resumen de las Rutas de prueba 48
5.2. Tabla resumen de las Rutas de prueba 50
5.3. Tabla resumen de las Rutas de prueba 51

Resumen

Uno de los objetivos que persigue la robótica móvil es el movimiento autónomo de los robots: la posibilidad de moverse a través de su entorno, sin chocar con los posibles obstáculos que se le presenten mientras navega hacia algún destino. A éstas acciones se las denomina navegación. Podemos además diferenciar entre navegación local, basada únicamente en la información instantánea proporcionada por los sensores sobre el entorno cercano del robot; y navegación global, basada en un conocimiento a priori del escenario completo en que se puede mover el robot.

El objetivo de este proyecto busca una combinación entre ambas capacidades de navegación. Proporcionando al robot un mapa de la planta 1 del edificio departamental II, se desea que éste sea capaz de moverse autónomamente de un lado a otro, usando para ello la información del mapa, así como la información sensorial de su entorno cercano. Además, el robot deberá evitar obstáculos inesperados no presentes en el mapa. Puesto que necesitamos saber en todo momento la posición del robot, se ha usado un simulador, dejando el problema de la localización para otro proyecto futuro.

La navegación global se ha resuelto con la técnica del campo de gradiente oGradient Path Planning, que garantiza caminos óptimos. Esta técnica genera un campo de potencial, que se expande como una onda por los huecos libres del mapa, desde el objetivo hasta la posición del robot, asignando a cada punto del espacio valores crecientes. Para navegar hacia un destino, simplemente se siguen los valores de este campo en la dirección del gradiente.

La navegación local complementa al algoritmo anterior, haciéndolo más robusto. Le dota de un comportamiento reactivo, a fin de que pueda esquivar obstáculos inesperados que no aparecen en el mapa. Para ello se usa la técnica de los campos de fuerza virtuales o Virtual Force Field (VFF). En ella, el objetivo marcado por la navegación global ejerce una fuerza atractiva sobre el robot y los obstáculos sensados directamente generan fuerzas repulsivas. El robot se dirigirá por la suma vectorial de ambas fuerzas. Ésto consigue dotar al robot de una navegación completa

La forma de combinar estas capacidades ha sido a través de esquemas JDE, eligiéndose una u otra dependiendo del contexto proporcionado por la información del mapa y de los datos sensado directamente. JDE es una plataforma desarrollada por el grupo de robótica de la URJC, que proporciona acceso a los sensores y actuadores de un robot. JDE se compone de esquemas, cada uno de los cuales realiza una tarea específica. Combinando varios esquemas obtenemos comportamientos complejos, como el que buscamos. Ésto nos permite separar claramente en esquemas cada una de las funcionalidades, y combinarlas para conseguir la navegación total que buscamos.

Capítulo 1

Introducción.

Ya desde la revolución industrial el hombre ha buscado liberarse de esfuerzo a través de la ciencia, y más concretamente, del uso de máquinas. En un principio se usaron para construir transportes más rápidos y potentes. Según avanzaba la técnica, se crearon las primeras máquinas autónomas, los primeros robots. Robot proviene de la palabra checa *robota*, que significa *trabajo forzado*. Esta palabra se debe al autor checo Karel Capek, que en su obra "Rossum's Universal Robots", de 1921, denomina así a unas máquinas construidas por el hombre y dotadas de inteligencia, que se ocupaban de todos los trabajos pesados. Partiendo de esta primera obra, el uso de los robots ha sido continuo en obras de ficción, dotándoseles de posibilidades todavía difícilmente alcanzables para nuestra tecnología actual. Como ejemplos de estos elementos de ficción podríamos nombrar a los famosos replicantes descritos por *Philip K. Dick*, en su relato *"Sueñan los androides con ovejas electrónicas"*, base de la película *Blade Runner* o los simpáticos y archireconocidos androides de *Star Wars, R2-D2 y C-3PO (figura 1.1)*

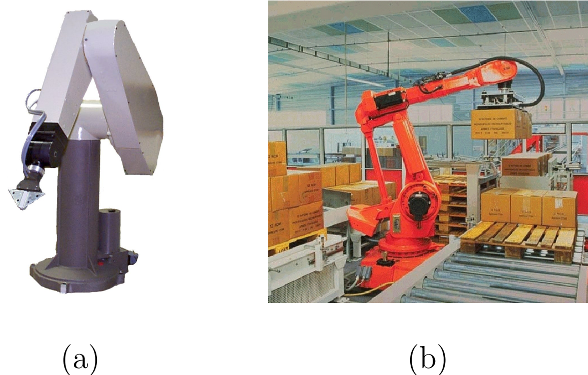
(a)

(b)

Figura 1.1: Los androides R2-D2 (a) y C-3PO (b).

En contraste con estos elementos de ficción, hoy día disponemos de robots reales, fundamentalmente en fábricas y centros de investigación. Robots que podríamos diferenciar en una división grosera entre robots manipuladores y robots móviles.
Los robots manipuladores son patrimonio casi exclusivo del sector industrial. En un principio eran controlados por humanos, de tal forma que éstos comandaban órdenes al manipulador, y éste los ejecutaba. Los manipuladores industriales más comunes se dedican a mover piezas, cortar, soldar, etc. Según la técnica evolucionaba, estos ma-

nipuladores fueron capaces de tomar decisiones respecto a sus propios movimientos, independizándose del operador humano. El primer tipo de brazo manipulador fue el PUMA (*Programmable Universal Manipulator for Assembly*) como el de la figura 1.2(a) Estos brazos robotizados aportan una mayor precisión, rapidez y potencia respecto al brazo humano. Uno de los principales usuarios de robots es la industria del automóvil. Por ejemplo, la empresa General Motors utiliza aproximadamente 16.000 robots para trabajos como soldadura por puntos, pintura, carga de máquinas, transferencia de piezas y montaje. El montaje es una de las aplicaciones industriales de la robótica que más está creciendo. Esta aplicación exige una mayor precisión que la soldadura o la pintura y emplea sistemas de sensores de bajo coste y computadoras potentes y baratas. Los robots se usan por ejemplo en el montaje de aparatos electrónicos, para montar microchips en placas de circuito.



(a) (b)

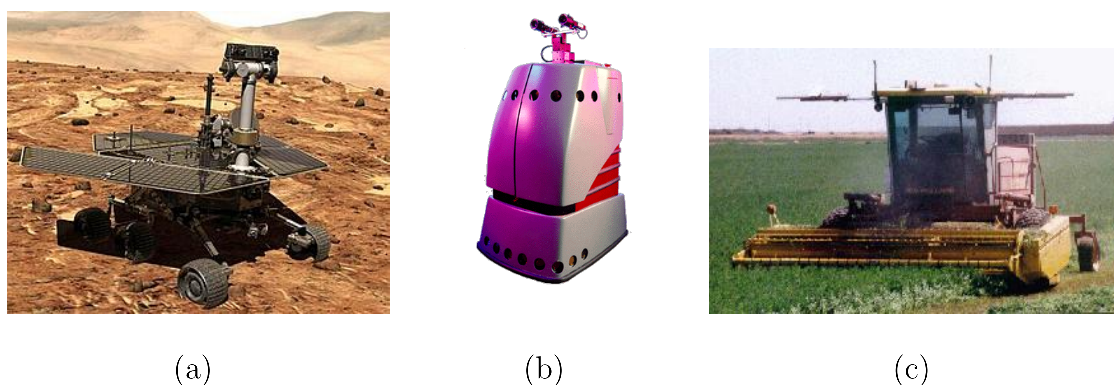
Figura 1.2: Un brazo P.U.M.A. (a) y un robot cargador (b)

La robótica móvil es uno de los campos de investigación que más auge está teniendo últimamente. Dentro de este campo tenemos dos bloques bien diferenciados. Los robots móviles teleoperados, todos aquellos robots cuyos movimientos son comandados a distancia por un operador humano, y los robots autónomos, capaces de tomar decisiones de movimiento por sí mismos. Los primeros han sido usados desde su nacimiento para ejecutar tareas peligrosas o acceder a lugares de difícil acceso para los seres humanos. Algunos de sus cometidos son la desactivación de bombas, las exploraciones submarinas, exploraciones en ambientes extremos, como el interior de centrales nucleares, búsqueda de supervivientes en edificios derruidos, etc. Estos robots son los usados también para la exploración interplanetaria. Ejemplos recientes son las sondas americanas *Sojourner*, o las más recientes *Spirit* y *Opportunity* (figura 1.3(a)). Estas últimas podríamos considerarlas casi autónomas, ya que si bien se les indica lo que tienen que hacer, son capaces de realizar autónomamente los pasos necesarios para ello, pues el retardo de las ondas entre Marte y la Tierra desaconsejan una teleoperación directa.

Los robots móviles autónomos se caracterizan porque deciden cual es su movimiento sobre la información que recaban del mundo que les rodea, sin ninguna intervención humana. La mayoría de estos robots son prototipos de investigación, cuyos principales usos se dan entornos cerrados, en los que actúan como guías de museos o bibliotecas, como enfermeros robóticos que recorren la planta de un hospital llevando a los enfermos su medicación, o en ambientes de oficina. Ejemplos de estos robots son *Minerva*[1]

[1]http://www-2.cs.cmu.edu/ minerva/

(figura1.3(b)), que actuó como guia en el *Smithsonian's National Museum of American History* o *Rhino*, que lo hace en el *Deutsches Museum Bonn*. Sin embargo los robots autónomos van abandonando los entornos cerrados para salir a espacios al aire libre. Prueba de esto son los robots recolectores de tomates, melones, pepinos o champiñones, que utilizan visión artificial para identificar el fruto. [Pérez, 2004] Otra aplicación en este campo son los cosechadores automáticos de cereales (figura1.3(c)), que recorren de modo semiautónomo los inmensos campos de cereales segando y recogiendo el grano[2].



(a) (b) (c)

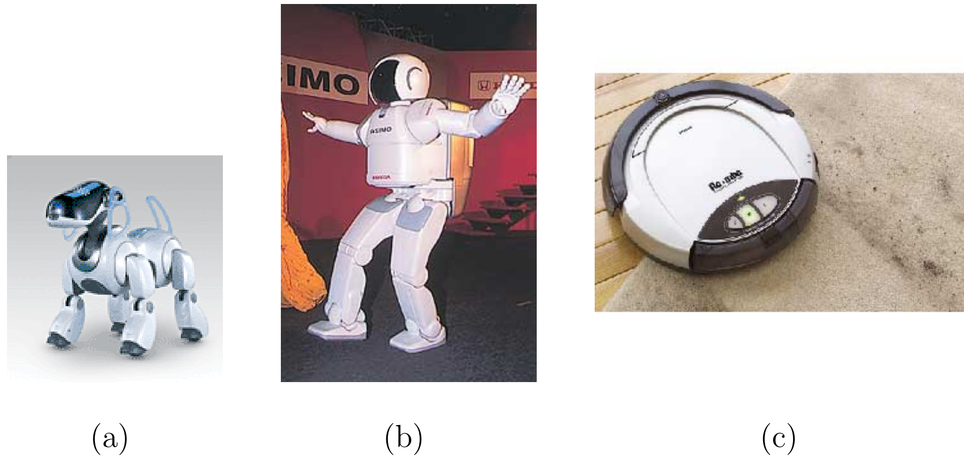
Figura 1.3: La sonda de exploración marciana Opportunity (a), Minerva, el robot guia (b) y una cosechadora autónoma del proyecto Demeter (c).

A cada día que pasa la robótica se introduce en el hogar cotidiana como robots de servicio, llegando a tener ya mascotas robóticas como el perrito *AIBO (figura 1.4(a))* de Sony. Además se ha avanzado mucho en los modelos humanoides, tales como *ASIMO* (figura 1.4(b))de HONDA o *QRIO*, también de SONY, consiguiendo comportamientos bípedos avanzados tales como andar con dos piernas, y subir y bajar escaleras. Esta integración en el mundo ordinario viene dado por el interés de dotar a estos robots con la habilidad de desempeñar tareas domésticas tales como planchar, lavar, barrer, acompañar a personas mayores y demás. En esta línea irobot ha lanzado el robot *Roomba*[3] (figura 1.4(c)),capaz de aspirar una habitación, pasando incluso por debajo de los muebles. También podemos mencionar los robots limpiadores de piscinas[4], o los cortacesped robóticos[5].

Como vemos, la robótica ha pasado del ámbito meramente científico, al comercial, donde las grandes marcas como Honda, Sony, Nec, Toyota, Fujitsu... etc, tienen sus robots estrella. Esto es un síntoma de la potencialidad del mercado de los robots

A lo largo de esta introducción hemos presentado robots que pueden adquirir distintas formas, y seguir distintas filosofías de movimiento, como usar ruedas, patas, o incluso hélices para volar o nadar. Pero todas estas filosofías necesitan de la navegación para funcionar. Todos los robots que hemos presentado resuelven de alguna manera el problema de la navegación, para prestar su servicio. Por ejemplo, el robot roomba sigue una espiral para barrer la habitación, las cosechadoras siguen una ruta indicada por un dispositivo GPS, o el perrito Aibo sigue a su dueño por la casa. Así pues el problema de la navegación es crítico para todos los robots autónomos. Se ha mejorado mucho en

[2]http://www.rec.ri.cmu.edu/projects/demeter/demeter.shtml
[3]http://www.irobot.com/home.cfm
[4]http://www.aquavacsystems.com/
[5]http://www.es.husqvarna.com/node1533.asp



(a) (b) (c)

Figura 1.4: El perrito mascota Aibo (a), Asimo (b) y Roomba (c)

la autonomía de los robots, pero éste es un camino por el que apenas se han dado los primeros pasos.

1.1. Navegación de robots

Como hemos comentado antes, la navegación es un problema crítico en los robots autónomos. Sin una buena navegación, el movimiento de los robots, y por tanto los servicios que podrían ofrecer, se verían muy limitados. Tradicionalmente se ha dividido la navegación en: navegación global y navegación local.

La navegación global trata de llevar al robot de un lado a otro del escenario, siguiendo un plan o una ruta. Para ello debe tener un conocimiento preciso del escenario. Este conocimiento suele proporcionársele a priori, mediante un mapa del escenario, o a posteriori, construyendo el mapa con los sensores disponibles. Una vez tiene el mapa, calcula la ruta entre el punto destino y la posición del robot. Para ello se usan técnicas de planificación y búsqueda, que suelen llevar tiempo. La ruta generada suele satisfacer algún criterio de optimidad.

La navegación local consiste en avanzar en cierta dirección evitando los obstáculos. Se diferencia de la anterior en que sólo trata la información del entorno directamente percibido a través de sus sensores, decidiendo a dónde moverse según ésta. Es una navegación que responde ante obstáculos no previstos en el mapa, o que aparezcan de repente, pudiendo, por lo tanto, evitarlos. Esta navegación dota de reacción rápida ante los imprevistos al robot.

La finalidad de este proyecto es conseguir que un robot se mueva en un ambiente de oficinas, entre dos puntos especificados gráficamente sin chocar contra ningún obstáculo, previsto o imprevisto. Así, debe ser capaz de planificar la mejor ruta entre dos lugares y recorrerla, evitando además todos los posibles obstáculos que se crucen con él. La solución propuesta combina, navegación global con navegación local.

1.1.1. Algoritmos de planificación de ruta

Los algoritmos de planificación, o de navegación global se encargan de generar un plan de acción para mover el robot desde un punto de origen a un punto objetivo. Para ello han de tener un conocimiento de su entorno, a fin de calcular la mejor trayectoria posible. Este conocimiento puede venir a priori, porque se le haya proporcionado un mapa de su entorno, o a posteriori, construyendo por medio de sus sensores el mapa de su entorno.

Algunos algoritmos de planificación son:

- Grafos de visibilidad (Nilsson 1969): Los grafos de visibilidad proporcionan un enfoque geométrico para solventar el problema de la planificación. Este método necesita modelos de entornos definidos con polígonos, y puede trabajar tanto en el plano como en el espacio. Un grafo de visibilidad GV es un grafo no dirigido, en el que cada nodo es cada uno de los vértices de los obstáculos que pertenecen al mapa. Se dice que dos nodos están conectados si y solo si son "visibles", es decir, se puede alcanzar el segundo nodo desde el primero (o viceversa) al seguir la línea recta que los une, sin interceptar algún obstáculo del entorno. También se consideran visibles si el segmento que une los dos nodos yace sobre una arista del polígono que modela a un obstáculo. El algoritmo de planificación basado en grafos de visibilidad consta de dos fases fundamentales: una primera, de construcción del grafo; y una segunda, de búsqueda. Esta última encontrará una ruta desde el origen al destino siguiendo los arcos del mismo. La ruta consiste en la sucesión de nodos por los cuales deberá pasar el robot al seguir los arcos, para llegar a la configuración final qf partiendo desde la de inicio qa. Así, esta se define por un conjunto ordenado de nodos del grafo. Este algoritmo ha sido desarrollado este año en el grupo de robótica [López, 2005].

- *Gradient Path Planning* [Konolige, 2000]: Este algoritmo se basa en asignar a cada punto del espacio un valor numérico. Al destino al que viajará el robot se le asigna valor 0, y a partir de aquí generamos una onda, similar a la generada por una piedra al caer al agua. El frente de esta onda se expandirá desde el destino, por el espacio libre y frenándose en los obstáculos, asignando un valor creciente a cada punto del espacio por el que pase. Así los puntos inmediatamente más cercanos al objetivo, tendrán valor 1, los siguientes valor 2, y así sucesivamente. El algoritmo se detiene al llegar al punto en que se encuentra el robot. Si el frente de onda llega a un punto que ya tenía un valor asignado, existen dos caminos para llegar a él, pero nos quedamos con el valor ya existente, pues se ha llegado antes a él, por lo tanto, es el camino más corto.El camino óptimo para llegar al destino es aquel que viaja por los puntos del espacio cuyo valor del gradiente sea menor que en el que nos encontramos. La analogía es crear un pozo, con el objetivo a viajar en su sima. Para llegar al objetivo, simplemente nos dejaríamos caer por la zona de mayor pendiente siempre. Este algoritmo además, incorpora los obstáculos del mapa al gradiente, por lo que genera un camino evitando éstos. Por construcción, este algoritmo asegura que el camino generado es el más corto. Además este algoritmo puede calcular una trayectoria a priori, o simplemente consultar el valor del campo cuando se necesite, lo que dota de mayor robustez al algoritmo, ya que podemos desviarnos de la ruta óptima, y aun así, encontrar el objetivo. Este es la técnica que usaremos para la navegación global.

1.1.2. Algoritmos de navegación local

La navegación local consiste en avanzar en cierta dirección evitando el choque con obstáculos próximos del entorno directamente percibidos a través de los sensores. Se diferencia de la navegación global en tanto en cuanto que no necesita información de la posición, navega usando la información de los obstáculos directamente proporcionada por los sensores de a bordo. Es además una navegación reactiva, pudiendo esquivar obstáculos inesperados por el plan general. Esta navegación es importante en entornos de oficina, pudiendo aparecer obstáculos imprevistos en cualquier momento, puesto que la gente se mueve constantemente, hay sillas, mesas, papeleras y demás, que no aparecen en el mapa..
Algunos métodos de navegación local son:

- Método de velocidad y curvatura *(CVM)* [GSyC, 2004]: Este método trabaja añadiendo restricciones al espacio de velocidad y escogiendo el punto en el espacio que satisface todas las restricciones y maximiza una función objetivo. Las restricciones derivan de las limitaciones físicas de las velocidad y aceleración propias del robot, y de los datos sensoriales que indican la presencia de obstáculos. Éstos últimos se usan para representar, para cada posible curvatura, cuan lejos puede viajar el robot antes de colisionar con un obstáculo. La distancia curva a los obstáculos se haya dividiendo el espacio de velocidad en un número discreto de regiones, cada una de los cuales tiene una distancia constante al punto de impacto. Este método encuentra el punto de cada región que maximiza la función objetivo. El punto máximo de entre todos, se usa para comandar el robot.

- Método de carriles y velocidad*(VLM)* [GSyC, 2004]: Este método combina el *CVM* con una nuevo método direccional llamado el *método de carriles*. El *método de carriles* divide en entorno en carriles, y luego elige el mejor a seguir para optimizar el viaje hacia la dirección deseada. Una dirección local es calculada para entrar y seguir el mejor carril, y CVM usa esta dirección para determinar las velocidades lineal y angular óptimas. Para ello considera la dirección a seguir, las limitaciones físicas y las restricciones del entorno. Combinando los métodos de dirección y velocidad, VLM consigue un movimiento libre de colisiones así como un movimiento suave teniendo en cuenta la dinámica del robot.

- Campos de fuerza virtuales *VFF o Virtual Force Fields* [J. Borenstein, 1989]. :Con esta técnica el robot tiene que viajar hacia un objetivo, que ejerce una fuerza atractiva sobre él, simulando un tirón gravitatorio. Esta fuerza es mayor, cuanto más lejos esté el robot del objetivo. Sin embargo no es esta la única fuerza que actúa sobre el robot. Puesto que el fin último de esta técnica es evitar los obstáculos más cercanos al robot, éstos generan una fuerza repulsiva sobre el robot, mayor cuanto más cerca está de éste. Así el movimiento del robot se ve gobernado por la suma vectorial de todas las fuerzas que afectan al robot. Esta técnica garantiza el alejarse de los obstáculos y recuperar el rumbo hacia el objetivo una vez se han superado éstos. Es la técnica usada para dotar de navegación local al robot.

1.2. Grupo de Robótica de la URJC

El grupo de Robótica de la Universidad Rey Juan Carlos está compuesto por profesores y alumnos. Los intereses de este grupo giran alrededor de la generación de comportamiento artificial en robots, tocando multitud de temas como lógica borrosa, visión artificial, estimación, teoría de control, elaboración de mapas, inteligencia artificial, arquitecturas de control... Básicamente el grupo se dedica a la programación de robots y el tratamiento de la información recabada por los distintos sensores. El grupo dispone de 30 robots Lego, 6 robots EyeBot, 2 robot Pioneer, 2 cuellos mecánicos y 3 perritos Aibo de Sony (figura 1.5).

Figura 1.5: Los robots de los que dispone el grupo de Róbotica de la URJC

Uno de los objetivos del grupo es crear un equipo de robots capaz de participar en la RoboCup[6]. Este campeonato mundial es uno de los más conocidos en robótica móvil. En un principio se usaron los robots EyeBot, pero actualmente se usan los perritos Aibo de Sony.

Otra de las líneas de investigación que se siguen, está basada en la generación de comportamientos autónomos en entornos de oficina, dentro de la que se ubicaría este proyecto. En esta línea se han producido múltiples trabajos que abordan y resuelven temas como seguimiento de una persona [Calvo, 2004] [Herencia, 2004], en los que un robot tenía que seguir a una persona a través de un edificio, navegación local [Lobato, 2003], localización [Crespo, 2003] [Benítez, 2004] y navegación global [López, 2005], donde también entraría este proyecto
Estos comportamientos se programan sobre una plataforma desarrollada íntegramente en el grupo llamada JDE (Jerarquía Dinámica de Esquemas). JDE proporciona un acceso sencillo a los sensores y actuadores del robot. Se basa en crear pequeños esquemas, cada uno de los cuales realiza una función muy determinada; consiguiendo generar comportamientos complejos uniendo varios de estos esquemas. Los esquemas pueden ser de dos tipos: perceptivos, si sólo se encargan de recoger información de los sensores, y de actuación, si se encargan de controlar el robot u otros actuadores. Veremos esta plataforma con detalle en el capítulo 3

Esta memoria trata de explicar cómo se ha llegado a la solución final para este comportamiento de navegación a lo largo de 6 capítulos. El capítulo 2 fija los objetivos y requisitos planteados a la hora de realizar el proyecto. El capítulo 3 presenta la

[6] http://www.robocup.org

plataforma de desarrollo, tanto hardware como software, sobre la que ha sido programado el código del proyecto. El capítulo 4 descubre la solución final para el problema, explicando la implementación del comportamiento. El capítulo 5 presenta los resultados experimentales validando el comportamiento del robot que hemos programado en distintos escenarios y, por último, el capítulo 6 refleja las conclusiones y las posibles líneas futuras por las que se puede hacer crecer este proyecto.

Capítulo 2

Objetivos.

En este capítulo se describen los objetivos del presente proyecto, así como sus requisitos de partida y la metodología.

El objetivo principal de este proyecto es dotar al robot de un algoritmo de navegación, capaz de conducirle de un despacho a otro de la segunda planta del edificio departamental II, sin chocar contra ningún obstáculo. Además de ésto debe ser capaz de navegar sorteando obstáculos, bien previstos, como paredes o puertas, bien imprevistos, como personas, sillas o papeleras.

2.1. Objetivos

El objetivo es conseguir dotar a un robot de la habilidad de moverse a través de un entorno de oficinas, como puede ser la segunda planta del edificio departamental II de la ESCET, desde un punto origen a un destino conocido. El robot ha de ser capaz además de llegar al objetivo sin chocar contra ningún obstáculo, bien sea de tipo estático como una pared, bien sea de tipo dinámico como una persona. Los subojetivos en los que se articula este objetivo principal, son:

1. Desarrollar un algoritmo de navegación global.
2. Desarrollar un algoritmo de navegación local.
3. Combinar ambos algoritmos para tener una navegación total.
4. Ampliar JDE de tal manera que acepte la plataforma de simulación de robots Player/Stage.

2.2. Requisitos

El desarrollo del proyecto estará guiado por los objetivos comentados anteriormente y deberá ajustarse a los requisitos de partida, que condicionan la solución de navegación desarrollada en este proyecto. Estos requisitos son los siguientes:

1. El robot concreto al que se orientan los algoritmos de navegación de este proyecto es el robot Pioneer. Sin embargo no se va a trabajar sobre un robot real, se trabajará sobre un simulador, ya que se necesita saber en todo momento la localización del robot. En el robot real los sensores internos de posición acumulan

errores, llevando a incertidumbres sobre la posición actual del robot, pensando que está en un punto mientras realmente está en otro. Éste es un problema que aún no está resuelto de modo fiable por el grupo. Para solventarlos usaremos simuladores donde, explicitamente, se puede anular el ruido en la posición del robot. Los simuladores usados serán el SRIsim de Saphira, y el simulador proporcionado por la plataforma Player/Stage. Ambos simuladores son capaces de simular un robot Pioneer perfectamente, así como el entorno de oficinas necesario. El hecho de usar uno u otro, dependerá de si queremos introducir obstáculos dinámicos, o sea, imprevistos, para lo que usaríamos Player/Stage, o no. La solución final debe funcionar sobre ambas plataformas.

2. En cuanto al software, se ha de desarrollar el comportamiento del robot en la plataforma denominada JDE que se explicará detalladamente en el siguiente capítulo.Por un lado, la utilización de esta plataforma simplifica el desarrollo del comportamiento. Por el otro lado, restringe a que la navegación se programe en lenguaje C, y utilizando esquemas. La solución final debe funcioar sobre JDE (versión 2.4) y otos (versión 5.2).

3. La navegación debe ser vivaz. Los algoritmos desarrollados no pueden perder mucho tiempo pensándose el próximo movimiento ha realizar, porque ha de reaccionar rápido para evitar la colisión con cualquier posible obstáculo.

2.3. Metodología

En esta sección se describe la metodología utilizada para la realización de este proyecto. Básicamente ha consistido en realizar iteraciones que se componen de: diseño, implementación y experimentos, así como reuniones periódicas con el tutor.

Para la realización del proyecto se establecieron unas tareas a realizar entre la idea del proyecto hasta la realización de la misma produciéndose el producto final.

El desarrollo de este proyecto se ha basado en el modelo de desarrollo en espiral basado en prototipos. La elección de este modelo de desarrollo se basa en la necesidad de separar el comportamiento final en varias subtareas más sencillas para luego fusionarlas. Cada tarea finalizada aporta los requisitos e información necesaria para abordar la siguiente iteración del modelo de desarrollo.

La gran ventaja de este modelo de desarrollo es la existencia de puntos de control al finalizar cada iteración. Además es altamente flexible en cuanto al cambio de requisitos, hecho muy común en este tipo de proyectos.

En este tipo de modelo de desarrollo, los productos son creados gracias al número de iteraciones que se da en el proceso de vida de software. En la figura 2.1 se puede observar los cuatro ciclos que forman este modelo de desarrollo: **Análisis de requisitos, diseño e implementación, pruebas y planificación del próximo ciclo de desarrollo.**

Al final de cada iteración se produce un prototipo. Un prototipo es una versión preliminar de un sistema con fines de demostración o evaluación de ciertos requisitos.

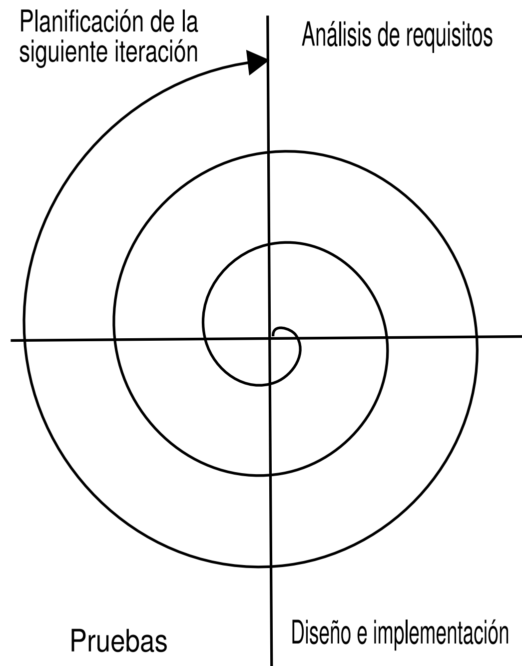


Figura 2.1: Modelo en espiral

Los prototipos creados a lo largo del proyecto, han sido:

- **Prototipo 1:** Generación del gradiente.
- **Prototipo 2:** Generación del gradiente incorporando obstáculos a éste.
- **Prototipo 3:** Navegación global usando el gradiente, bajo SRIsim.
- **Prototipo 4:** VFF bajo SRIsim.
- **Prototipo 5:** Navegación global+VFF bajo SRIsim;
- **Prototipo 6:** Soporte para Player/Stage.
- **Prototipo 7:** Navegación global usando el gradiente, bajo Player/Stage.
- **Prototipo 8:** VFF bajo Player/Stage.
- **Prototipo 9:** Navegación global+VFF bajo Player/Stage;

Estos prototipos han sido las salidas de cada una de las iteraciones hechas en el desarrollo del software. Estas iteraciones han sido planeadas en reuniones semanales con el tutor del proyecto, presentándose versiones numeradas del código generado. Para un mejor seguimiento semanal, se ha llevado paralelamente al desarrollo del software, un blog[1], permanentemente accesible desde Internet, con cada uno de los avances y problemas surgidos a lo largo de la semana.

Cada uno de los prototipos anteriores se dividen en su correspondiente especificación de requisitos, su diseño e implementación. Una muestra de este ciclo lo podemos ver en el siguiente ejemplo, basado en el prototipo 1:

[1] http://ladooscuro.no-ip.com/phpwiki

- **Análisis de requisitos:**
  - Hay que capturar el mapa del entorno.
  - Usar la librería `Gridslib`, para poder almacenar la información del mundo.
  - Desde un punto fijo, generar el gradiente y que se expanda hasta ocupar todo el mundo.
- **Diseño**:
  - Necesitaremos un analizador sintáctico que capture la información del fichero del mundo.
  - Usaremos la librería `Gridslib` para crear una rejilla donde cada celda almacenará información sobre algún punto del espacio.
  - Para expandir el campo, seguiremos una función recursiva, en la que una casilla intenta expandir su campo a sus vecinas.
- **Implementación**:
  - Se programa el analizador sintáctico en C, que abre un fichero y guarda la información de éste en una estructura interna.
  - Con esta estructura se genera una rejilla lo suficientemente grande para representar todo el mundo. Además se da valor a cada celda dependiendo de si está ocupada o no.
  - Desde la casilla elegida como origen, expandimos el gradiente hacia sus casillas inmediatamente vecinas, usando la función recursiva.
- **Pruebas**:
  - Probamos que el campo se extiende correctamente en un entorno vacío.
  - Probamos que el campo se extiende correctamente en un entorno con obstáculos.
- **Planificación de la siguiente iteración**:
  - Se ve que la función recursiva es extremadamente lenta. En la siguiente iteración se intentará hacerla iterativa. Además el frente no se propaga correctamente, no se genera un frente de onda que se expande circularmente desde el origen.
  - Puesto que se pretende hacer iterativa hay que pensar cómo almacenar la información de las casillas a las que se expande el campo. Hay que ver qué estructura usar para hacer ésto.

Capítulo 3

Plataforma de desarrollo

En este capítulo vamos a explicar la plataforma software sobre la que ha sido desarrollado este proyecto, cuya implementación comentaremos en el siguiente capítulo. Presentaremos también el robot sobre el que se ejecutará, y comentaremos brevemente algunas bibliotecas de apoyo y herramientas útiles que hemos usado. Por último, presentaremos los distintos simuladores utilizados.

3.1. Robot Pioneer

El robot que utilizamos como referencia de este proyecto es el Pioneer3x. Este robot lo comercializa la empresa norteamericana ActivMedia[1] y dispone de soporte técnico activo. El robot dispone de un equipo sensorial para medir el estado de su entorno y unos motores que le permiten moverse.

Este modelo de la gama Pioneer viene equipado con un procesador de 18 MHz Hitachi H8S/2357 con 32Kb RAM y 128Kb Memoria flash. Tiene una autonomía de 5 horas y es capaz de adquirir una velocidad máxima de 360º/s. Soporta un máximo de 23 Kg de carga.

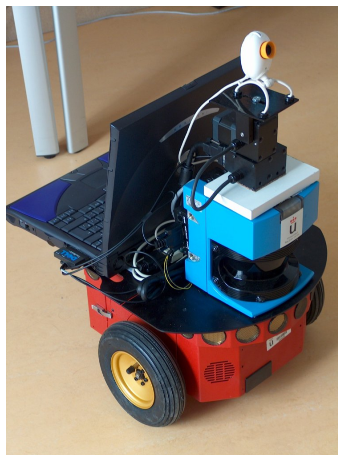


Figura 3.1: Robot Pioneer sobre el que se basa el proyecto

[1] http://www.activmedia.com

El Pioneer dispone de una corona de 16 sensores de ultrasonido que rodean al robot y lleva incorporados unos odómetros (*encoders*) asociados a las ruedas para saber cuánto han girado éstas. En nuestro proyecto usaremos estos odómetros porque nuestro modo de navegación se basa en la posición del robot. Además le hemos incorporado un sensor láser y una cámara de visión tal y como se muestra en la figura 3.1. Los actuadores principales del robot son dos motores de continua, cada uno asociado a una rueda, que dotan al robot de un movimiento diferencial tipo tanque.

Por último, comentar que el robot lleva a bordo un ordenador portátil conectado a la red exterior mediante un enlace inalámbrico, con una tarjeta de red 802.11 que le proporciona una velocidad de 11Mbps en sus comunicaciones. De esta forma el programa de control puede correr a bordo del portátil o en cualquier otro ordenador mediante la comunicación wi-fi. Esta última característica es posible gracias a la arquitectura JDE que nos permite una comunicación mediante el modelo cliente-servidor.

3.2. Arquitectura y modelo de programación con JDE

JDE es un entorno de programación para escribir aplicaciones de robots, programas que generan en el robot comportamiento autónomo. Consta de dos partes: dos servidores, que dan acceso a los sensores y actuadores del robot real o un simulador a través de mensajes, y una jerarquía de esquemas, que se comunican entre ellos a través de variables. Este entorno ha sido creado por el grupo de Robótica de la URJC [Plaza, 2003]. JDE se encuentra a disposición de todo el mundo debido a su carácter de software libre, pues tiene licencia GPL [2].

Típicamente, para ejecutar cualquier programa en el robot que utilizara los sensores o motores debía ejecutarse obligatoriamente en el portátil a bordo del robot. Para superar esta limitación nació la arquitectura de servidores y clientes de JDE, tal y como ilustra la figura 3.2.

Otra característica ventajosa que heredan los servidores y clientes de JDE es que el servidor otos se puede conectar indistintamente al simulador de Saphira/ARIA, al simulador Player/Stage, o a la plataforma real. De este modo los clientes que implementan alguna técnica de control con JDE se pueden probar en el simulador. Los datos sensoriales corresponden al entorno simulado y la situación en que se encuentre en él el robot simulado. Esto es muy útil para depuración. Además el mismo cliente podrá ejecutarse sin recompilar nada en el robot real. Una limitación que hereda es que ese simulador no incluye visión.

La tarea de los servidores es ofrecer acceso a los sensores y a los comandos motrices a cualquier programa cliente que los requiera, incluso remotamente. Cada servidor se encarga de ciertos sensores y actuadores que existen en la máquina y ofrece su funcionalidad al resto de clientes mediante una *API de mensajes*.

Así el servidor `otos` se encarga del acceso a los motores, sensores de proximidad (como infrarrojos, láser y sónar) y los sensores de voltaje. El servidor `oculo` se encarga de los sensores de imagen (cámaras) y, en caso de existir, del movimiento del cuello mecánico.

Entre un cliente y un servidor se establece una conexión *tcp* por la cual se comunican los dos. Este modo de programación bajo JDE tiene ciertas desventajas como el

[2] http://gsyc.escet.urjc.es/jmplaza/software.html

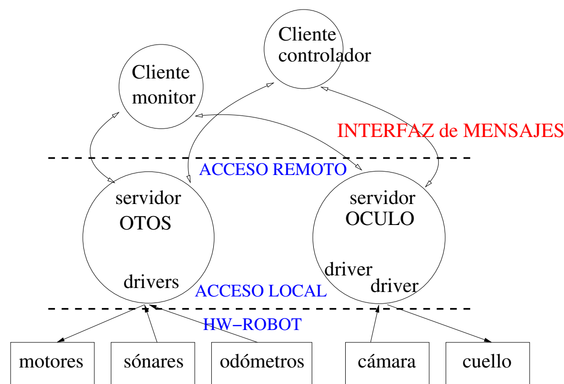


Figura 3.2: Arquitectura software con servidores y clientes de JDE

aumento de los retardos entre que se lee el dato sensorial y que éste llegue, a través de la red, al programa donde se realiza el procesamiento. Otro inconveniente es cierta desincronización de las medidas.

La programación de aplicaciones directamente sobre los servidores JDE obliga a que la propia aplicación se encargue de enviar y recibir mensajes hacia/desde los servidores. Esto implica que el programador gaste cierto esfuerzo en crear los búfferes de recepción y de envío oportunos. Además debe tener en cuenta el tiempo de cómputo de su programa dedicado a las comunicaciones.

La programación en esquemas libera a la aplicación de esta tarea, ya que incorpora varios *esquemas de servicio* que se encargan de las comunicaciones: recoger de los servidores la información de los sensores del robot, y enviar a los motores las órdenes de movimiento a través de los mismos servidores. Este método de programación con los esquemas de JDE se representa en la figura 3.3 y es el usado en nuestro proyecto.

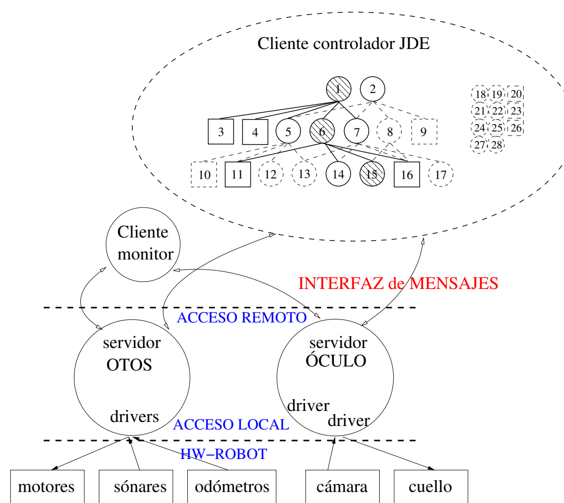


Figura 3.3: Arquitectura software del sistema programado con esquemas

La programación con esquemas JDE define la aplicación robótica como un conjunto de esquemas que se ejecutan simultáneamente y en paralelo. Los esquemas no son más que procesos que funcionan concurrentemente y se encargan de una tarea sencilla y concreta. La ejecución simultánea de varios esquemas dan lugar a un *comportamiento*.

Existen dos tipos de esquemas en JDE: los esquemas perceptivos y los esquemas motores o de actuación. Los esquemas perceptivos se encargan de producir y almacenar

información que puede ser leída por otros esquemas. Esta información puede provenir de medidas sensoriales o de la información elaborada por otros esquemas. Los esquemas motores o de actuación toman decisiones sobre los motores o activación de otros esquemas de nivel inferior a partir de la información que generan los esquemas perceptivos. Ejemplos de estos son el esquema perceptivo *Gradient Builder*, o el esquema motor *Follow Grid*, desarrollados para este proyecto, como veremos en el siguiente capítulo.

Los esquemas se pueden organizar en niveles de tal forma que los esquemas de un nivel inferior son activados o desactivados por los esquemas del nivel superior. De esta organización podemos inferir la existencia de esquemas padre y esquemas hijo. Es probable que un padre tenga varios hijos y es el mismo padre el que se encarga de activar y desactivar a sus hijos. El padre en ciertas ocasiones puede hacer de árbitro entre sus hijos actuadores para decidir quién gana y toma el control. Un ejemplo de esto lo veremos en el siguiente capítulo, cuando expliquemos cómo el esquema *Gradient Planning* despierta y arbitra entre sus esquemas hijos *Follow Grid* y *VFF*.

El paso de información de unos esquemas a otros se produce mediante variables globales, lo que denominamos como un *API de variables*. Estas variables son utilizadas por varios esquemas a la vez por lo que podemos tener concurrencia. El uso de semáforos evitaría las condiciones de carrera, pero JDE opta por no usar semáforos para no producir más retardos en las lecturas y apuesta por no proteger estas variables de información. Esto es así puesto que se asume que, aunque se puedan producir condiciones de carrera esporádicas, que se manifestarían como una lectura errónea de sensores o un comando ruidoso, el alto refresco que ofrecen los servidores minimiza la incidencia de ésas.

Este proyecto hará uso de las ofrecidas por el esquema *Sensation Otos*, que pone a disposición del resto varias estructuras para leer los datos sensoriales del robot. Una de ellas es `us[16]`, se trata de un array de 16 posiciones que ofrecen las medidas de los 16 sensores de ultrasonido. Otra estructura importante es `laser[180]` en la que se guardan las 180 medidas que es capaz de procesar el láser. También dispone de información de odometría y situación del robot en la estructura `robot[5]`.

Respecto a las variables de actuación, el esquema *Motors* pone a disposición del resto de esquemas las variables `v`, que comanda el valor de la velocidad lineal y `w`, que lo hace con la velocidad angular.

3.2.1. Servidor Otos

El servidor otos reúne los servicios de los sensores de proximidad como sónares, láser y táctiles. Además ofrece los datos de odometría que generan los *encoders* de los motores de tracción del robot, tanto la posición como la velocidad estimada a partir de ellos. También entrega las medidas del sensor de voltaje de la batería y permite enviar comandos de movimiento a la base. Más en concreto, usaremos los datos de odometría para la navegación basada en posición, mientras que los datos del láser se usarán para la detección de obstáculos imprevistos y/o dinámicos.

3.2.2. Servidor Oculo

El servidor oculo auna las funciones asociadas a la unidad pantilt y a la cámara. Con ello permite a los clientes mover la unidad pantilt a voluntad especificándole ángulos objetivo y pone a disposición de los clientes el flujo de imágenes obtenidas con la cámara. Estas se ofrecen a los clientes a través de un servicio de imágenes bajo demanda. Se ha dado soporte a imágenes en niveles de gris e imágenes RGB. No ha

sido necesaria la utilización de Oculo en este proyecto, pues no utilizaremos la visión para nada.

3.3. Bibliotecas Gridslib y Fuzzylib

La bibliotecas auxiliares Gridslib y Fuzzylib se han usado para proporcionar distintas funcionalidades a los esquemas programados.

La biblioteca Gridslib aporta funcionalidad para crear y manejar rejillas de ocupación. Implementa varias técnicas de construcción de mapas como la regla de Bayes, la regla Dempster-Shafer (teoria de evidencia), rejillas borrosas o rejillas histógramicas. Esta biblioteca genera una rejilla cuadrada con unas celdas regulares de tamaño predeterminado. Tanto el tamaño de la rejilla como el de la celda es especificado en un fichero de texto, que la biblioteca lee. Además proporciona funciones para crear, reubicar, actualizar valores, etc..

En nuestro proyecto, la rejilla será útil para almacenar el valor del campo, y para representar el mapa del escenario. Cada celda de la rejilla almacenará un valor que según su valor representará un espacio libre, un obstáculo, un valor del campo asociado a un punto del espacio, etc.

La biblioteca Fuzzylib aporta funcionalidad para usar controladores borrosos. La lógica borrosa es básicamente una lógica multievaluada que permite valores intermedios para poder definir evaluaciones convencionales como sí/no, verdadero/falso, negro/blanco, etc. Las nociones como “más bien caliente” o “poco frío” pueden formularse matemáticamente y ser procesados. Así podemos tener valores intermedios entre verdadero y falso como “probable” o “muy poco probable”. El controlador borroso hace uso de un fichero de reglas del tipo

```
IF THEN ELSE
```

, que ofrece un valor de salida según sea el valor de entrada.

En este proyecto usaremos un controlador borroso para controlar las velocidades linear y angular del robot, que lo pilotarán en su navegación a través de su entorno, como explicamos en el capítulo 4.

3.4. Simulador SRIsim

El simulador SRIsim es un simulador de robots desarrollado en *SRI*[3]. Trabaja en conjunción con *ARIA [ActivMedia, 2002]*[4], que es una biblioteca escrita en C++ con licencia GPL de *ActivMedia Robotics*. *ARIA* es una plataforma software para escribir aplicaciones robóticas que incluye soporte para multiprogramación y soporta muchos sensores, actuadores y robots. Es muy conveniente para desarrollar programas para robots. SRIsim no tiene licencia GPL, sin embargo su código fuente se le puede pedir a su autor, Kurt Konolige. Podemos ver una captura de pantalla del simulador en la figura 3.4.

[3]http://www.sri.com/
[4]http://www.activmedia.com/

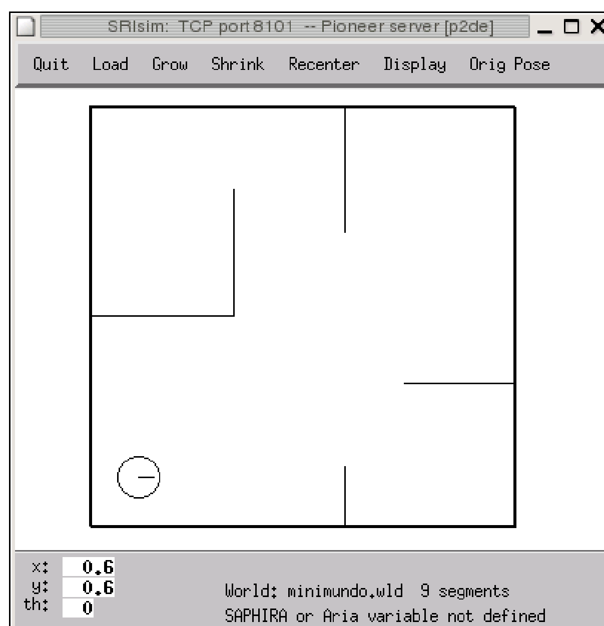


Figura 3.4: Interfaz gráfica de SRIsim

SRIsim simula el comportamiento de sensores reales como pueden ser los sensores sonar, láser, de odometría, de batería, etc; así como de actuadores, pudiendo dar valores de velocidad lineal y angular. Estos sensores y actuadores son los mismos de los que dispone un robot Pioneer.

En lo que respecta a los robots, SRIsim es un simulador realista, esto es, es capaz de simular los errores propios de un robot real introduciendo ruido en las medidas de sus sensores. Es capaz además de simular varios modelos de robots, con distintas configuraciones sensoriales, pudiendo elegir el que más nos convenga.

Puesto que para la navegación de nuestro robot hemos supuesto que sabemos en cada momento la localización del robot, hemos de anular el error simulado que SRIsim introduce en los odómetros. Para ello ha habido que acceder al código fuente, y poner a cero unos parámetros que introducían un error aleatorio en la medida de los sensores de odometría. Tras recompilar de nuevo, tenemos un simulador en el que los *encoders* dan una localización exacta del robot.

SRIsim es capaz de simular cualquier entorno que se pueda definir como intersección de segmentos. Lo cual lo hace muy apropiado para simular entornos de oficinas, que suelen ser una serie de habitaciones cuadradas conectadas entre sí por pasillos rectos. Los ficheros que definen el entorno a simular son ficheros de texto, que siguen una sintaxis predeterminada. En este fichero se indica el tamaño del entorno y la posición dentro de este entorno en que el robot está situado, así como su orientación. Las paredes vienen definidas como tuplas numeradas de cuatro puntos con el formato (X_{inicio},Y_{inicio},X_{final},Y_{final}). Podemos ver el fichero de texto que ha generado el mundo mostrado por la figura 3.4.

```
width 5000
height 5000
OriginPad 4999 4999
position 590 590 0

Start Line 1700 4000 1700 2500
```

```
AttachID 2
   1700 4000 1700 2500
End

Start Line  0000 2500 1700 2500
AttachID 3
   0000 2500 1700 2500
End

Start Line 5000 1700 3700 1700
AttachID 4
   5000 1700 3700 1700
End

Start Line 3000 0 3000 700
AttachID 5
   3000 0 3000 700
End

Start Line 3000 3500 3000 5000
AttachID 6
   3000 3500 3000 5000
End


Start Line 5000 0 0 0
AttachID 7
   5000 0 0 0
End

Start Line 0 0 0 5000
Attach ID 8
   0 0 0 5000
End

Start Line 0 5000 5000 5000
Attach ID 9
   0 5000 5000 5000
End

Start Line 5000 5000 5000 0
Attach ID 10
   5000 5000 5000 0
End
```

Este formato de fichero tendrá que ser digerido por nuestro algoritmo para saber cómo es el escenario a través del que tendrá que guiar la navegación del robot. Más concretamente, hemos construido un mundo virtual homólogo al lado izquierdo de la 1ª planta del edificio departamental II. Este mundo ha sido construido siguiendo el plano del arquitecto. Como podemos ver en la captura de la figura 3.5, es un mundo que representa las puertas y paredes, pero no representa el mobiliario, o las posibles personas que se pueden mover dentro de él.

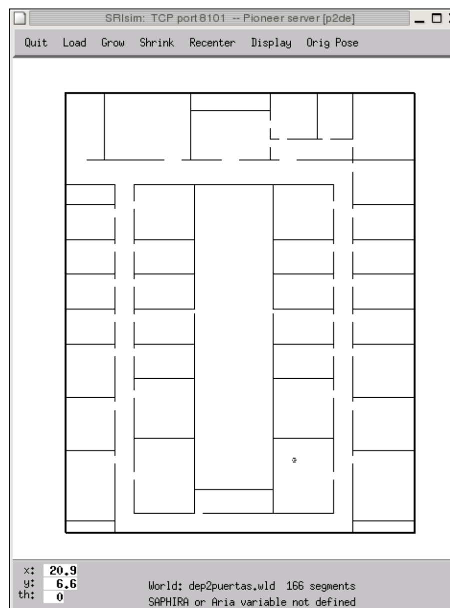


Figura 3.5: Mapa del lado izquierdo de la 1ª planta del Edificio departamental II para SRIsim

3.5. Player/Stage

La plataforma Player/Stage [Brian P. Gerkey, 2003][5] es otro entorno de programación de aplicaciones robóticas. Proporciona herramientas de código abierto que simplifican el desarrollo de controladores, particularmente para sistemas multi-robot. El entorno proporciona el servidor de dispositivos robóticos *Player*, que da acceso a sensores y actuadores desde programas clientes, y el simulador de múltiples robots *Stage*. Queremos además usar esta plataforma desde JDE.

Player es un servidor de dispositivos basado en sockets que permite el control de una amplia variedad de sensores y actuadores robóticos. Los clientes se conectan a *Player* y se comunican con los dispositivos intercambiando mensajes sobre un socket *TCP*. Esta abstracción permite neutralidad de ubicación, un programa cliente puede tomar control desde cualquier máquina que tenga conectividad a la red. *Player* sigue el modelo *UNIX* de tratar los dispositivos como ficheros. Así para recibir observaciones sensoriales, el cliente abre el dispositivo apropiado en modo lectura y lee esas medidas. De la misma forma, para comandar órdenes, deberá abrirlo en modo escritura, y escribir los comandos. Como resultado de su arquitectura basada en red, *Player* permite a cualquier cliente, localizado en cualquier lugar de la red, acceder a cualquier dispositivo; un robot podría llegar a ver a través de los ojos de otro.

Stage es capaz de simular una población de robots móviles, sensores y objetos del entorno. Todos los sensores y actuadores son accesibles a través de la interfaz estándar de *Player*. Los clientes no notan ninguna diferencia entre los dispositivos reales y su equivalente simulado en *Stage*. El hecho de simular varios robots a la vez, nos va a permitir tener obstáculos dinámicos, que no aparecen en el mapa. Así podremos tener un robot siguiendo el algoritmo de navegación desarrollado en este proyecto y otro teleoperado que se interpondrá en su camino para que sea evitado.

[5] http://playerstage.sourceforge.net/

Stage puede además simular cualquier entorno, ya que recibe este como una imagen en formato PNM. La novedad que supone representar el entorno con una imagen, en vez de como una colección de segmentos, es que tenemos libertad para representar entornos irregulares o curvos. Con el simulador SRIsim, sólo podíamos tener entornos de oficinas o basados en segmentos. Con *Stage* podemos simular desde una mesa de billar, a una ciudad como Madrid, como muestra la figura 3.6.

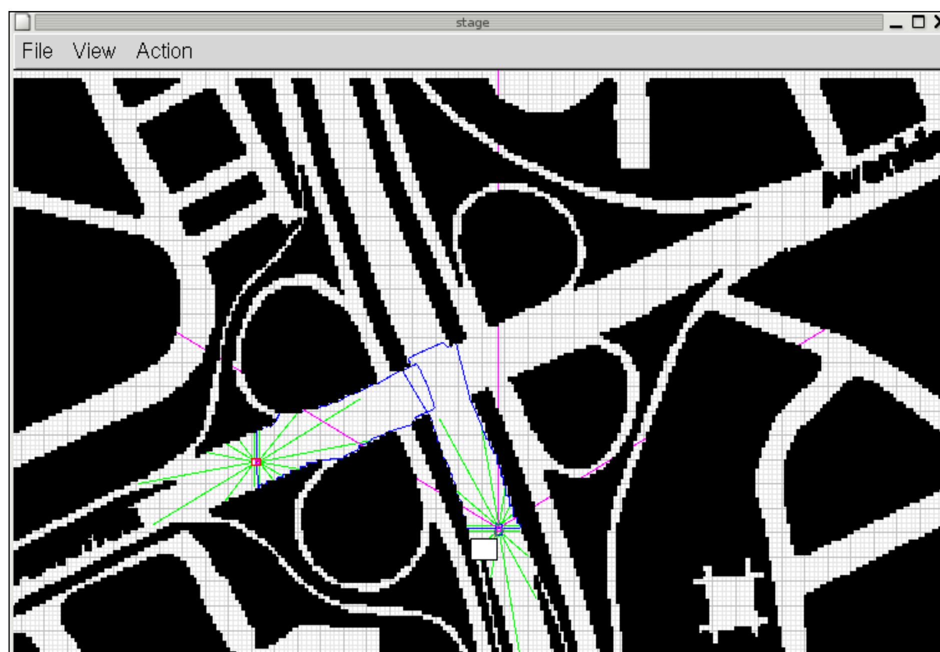


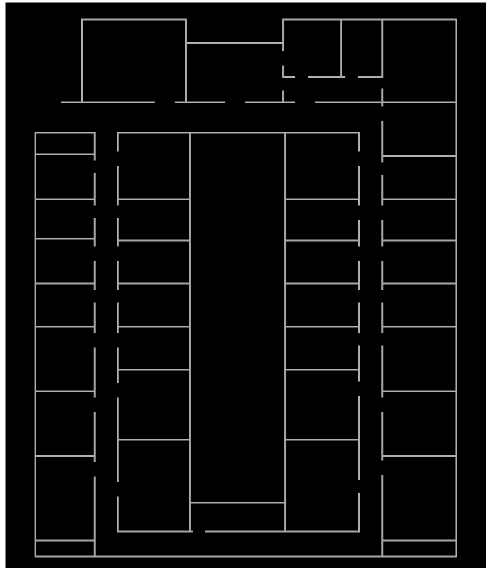
Figura 3.6: Stage simulando una parte de Madrid con 2 robots

Un entorno como el mostrado por la figura 3.6 es imposible representarlo con el simulador SRIsim. Para representar los mundos de *Stage* tendremos que reconocer el formato en que está escrito éste, para poder tratarlo. El formato de los ficheros PNM que significa *Portable aNy Map* está pensado para cubrir las 6 variaciones de los *bit/gray/pixel maps*. Estos mapas también se conocen como PNM, PGM y PPM. El fichero es simplemente un fichero de texto en el que vienen indicados el número mágico, el número de pixels de ancho de la imagen, el número de pixels de alto de la imagen, el número de colores representado y, por último, el valor de cada byte de la imagen. El número mágico indica el formato del fichero, tenemos los siguientes números:

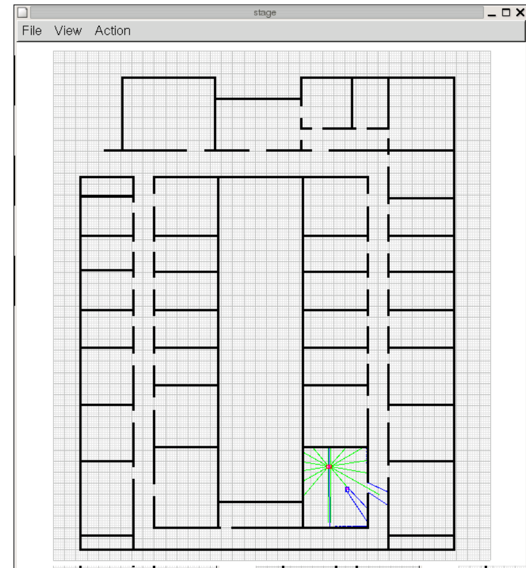
- P1, si el fichero es ASCII y en Blanco/Negro.
- P2, si el fichero es ASCII y en escala de grises.
- P3, si el fichero es ASCII y en color.
- P4, si el es binario y en Blanco/Negro
- P5, si el es binario y en escala de grises.
- P6, si el es binario y en color.

Se ha desarrollado un mapa de la 1ª planta del edificio departamental II en formato PNM a fin de que pueda ser usado por *Stage*. Podemos ver este mapa en la figura 3.7(a) y su representación en *Stage*(figura 3.7(b)), así como parte de la estructura de este fichero PNM.

```
P5
# CREATOR: The GIMP's PNM Filter Version 1.0
271 316
255
\uffff\uffff\uffff\uffff\uffff\uffff\uffff\uffff\uffff\uffff\uffff\uffff
```



(a)



(b)

Figura 3.7: Fichero PNM del departamental II (a), y su representación en *Stage*

Aparte de la amplia libertad de entornos a simular con *Stage*, también se pueden simular éstos a distintas escalas, pues podemos indicar el número de milímetros que ocupa cada pixel; y usando uno, o múltiples robots. Todo esto se le indica a *Stage* mediante un fichero de texto, que dice qué mapa cargar, a qué escala y con cuántos robots. Esto nos viene especialmente bien, ya que, como comentamos antes, usando más de un robot podemos simular obstáculos dinámicos o imprevistos. Por ejemplo haciendo que un robot se cruce en el camino del otro, a fin de depurar la navegación local. Podemos ver el fichero que genera el mundo mostrado por la figura 3.7(b).

```
include "usc_pioneer.inc"

bitmap
(
   file "stage_dep2_puertas.pnm"
   resolution 0.1425
)

usc_pioneer_truth (color "red" name "robot1" port 6665
                   pose [ 24.613 8.507 0.000 ]
                   mcom())

usc_pioneer (color "blue" name "robot2" port 6666 pose [ 28.950 7.700 90.000 ])
```

En este fichero se dice que el mundo a cargar es el del fichero "stage_dep2_puertas.pnm" y se representará con una escala de 0.1425 mm/pixel. Además se crearán 2 robots Pioneer, uno rojo situado en el punto (24.613, 8.507) con una orientación de 0 grados y otro azul situado en el punto (28.950, 7.700) con orientación de 90 grados. Hace falta mencionar que la coordenada (0,0) del mundo *Stage* es la esquina inferior izquierda.

3.5.1. Soporte JDE para Player/Stage

Como hemos visto la plataforma *Player/Stage* nos aporta unas características de las que carece SRIsim, como simular una población de robots, no uno sólo, o simular entornos no basados en segmentos. Sin embargo la plataforma JDE carecía de soporte para Player/Stage.

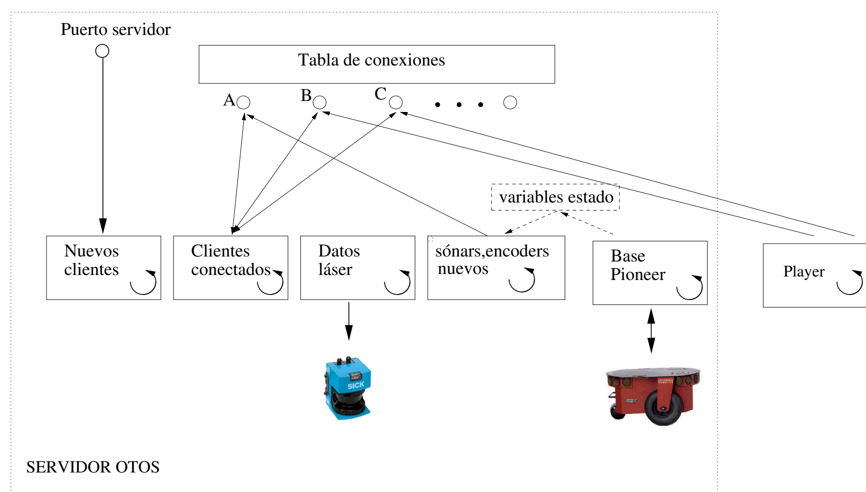


Figura 3.8: Implementación del servidor Otos con soporte para Player/Stage

A fin de superar esta limitación, se ha programado paralelamente al algoritmo de navegación el soporte de JDE. Para ello se ha implementado dentro del servidor *Otos* otra hebra, dedicada a conectarse a *Player* para recibir las medidas sensoriales y comandar las órdenes de actuación, como muestra la figura 3.8. Esta hebra simplemente hace una traducción del formato de mensajes de *Otos*, al formato de mensajes entendido por *Player*. Para ello se ha usado una biblioteca proporcionada en el mismo paquete que *Player/Stage*, la biblioteca *Libplayerc*. Esta biblioteca ofrece una *API de funciones* para construir un cliente que se entienda con *Player*. Así, usándola, traducimos las medidas sensoriales recibidas desde *Player* al formato *Otos*. Igualmente las ordenes de actuación comandadas por *Otos* son traducidas al formato *Stage* a fin de que el robot se mueva idénticamente a como lo haría sobre el robot real.

Tras hacer esto, el servidor Otos se puede conectar indistintamente al robot real, o a un robot simulado, bien sea sobre SRIsim o sobre Player/Stage. Para esto Otos lee de un fichero de configuración *otos.conf* en el que está indicado a qué debe conectarse. Se ha añadido a este fichero de configuración una nueva regla, que le dice si se debe conectar a *Stage*:

```
#stage <player_host> <player_port>
stage localhost 6665
```

Capítulo 4

Descripción informática

En este capítulo explicaremos la solución desarrollada, que satisface los objetivos y los requisitos y restricciones explicados en el capítulo 2, y que usa la plataforma comentada en el capítulo 3. Para ello detallaremos brevemente el algoritmo utilizado y describiremos en detalle cómo ha sido programado en forma de esquemas JDE. Explicaremos cada uno de los esquemas y como se relacionan unos con otros.

4.1. Aproximación al método del gradiente

Hemos hablado en la introducción de esta memoria que la navegación es un problema crítico para los robots autónomos. Como vimos en el capítulo 2, el objetivo de este proyecto es dotar a un robot de un algoritmo de navegación, combinando navegación global con navegación local. Para resolver el problema de la navegación global, hemos optado por usar el método del gradiente [Konolige, 2000], que garantiza una trayectoria mínima entre los puntos a viajar. Además, esta trayectoria no sigue la distancia euclídea mínima, sino que incorpora los obstáculos en su cálculo de la trayectoria.

El método del gradiente se basa en generar un frente de onda que recorre el espacio libre del escenario, desde el punto destino hasta la posición donde se encuentra el robot, que es el punto de partida de la trayectoria a seguir. Según se propaga este frente de onda, va asignando valores crecientes al punto del espacio por el que pasa. Por construcción, el espacio libre tiene un valor inicial de 0. Si el frente de onda llega a un punto del espacio que ya ha sido visitado, el frente muere ahí, pues ese punto ya tiene un valor asignado y, por construcción, éste es menor que con el que se ha llegado ahora. El frente de onda se expande hasta llegar a la posición del robot, u ocupar todo el escenario.

La cercanía de un punto a un obstáculo, hace que este punto tenga un valor inicial mayor que 0, que crecerá cuanto más cerca esté del obstáculo, ya que los obstáculos han generado su propio campo, con un frente de onda propio. Este campo de obstáculos ha sido propagado al reves, decreciendo según se aleja del obstáculo.

El frente de onda, al propagarse, suma el valor a asignar a ese punto del espacio, a su valor por defecto. Ésto evitará que el robot se acerque a los obstáculos. Si los obstáculos no tuvieran un campo asociado, las trayectorias seguidas irían raspándolos. Con el campo generado por los obstáculos se aumenta la seguridad para el robot.
Una vez se ha generado el campo, la navegación es tan simple como ver el campo de los puntos alrededor de la posición actual, y viajar hacia el punto cuyo valor del campo sea

menor que en el que estamos. Esta técnica permite generar una ruta, desde el punto donde está situado el robot al destino a viajar, que no es más que seguir el *gradiente* del campo escalar generado. Además, por construcción esta ruta es de distancia mínima y única, e incorpora los obstáculos a ella. Sin embargo no siempre seguiremos esta ruta estrictamente, como tendremos un algoritmo de navegación local, habrá momentos en que éste saque al robot de la ruta óptima, por ejemplo a aparecer un obstáculo imprevisto. Esta técnica global, sin embargo, nos asegura que llegaremos al objetivo, pues simplemente, consultaremos el valor del campo, y seguiremos el gradiente desde ese punto, que, por construcción, nos llevará al objetivo.

Resumiendo, esta técnica propaga un campo escalar de valores crecientes desde el objetivo a viajar hasta la posición del robot. Este campo se suma a un campo asociado a los obstáculos, que alejará al robot de estos. Cuando el campo ha terminado de expandirse, se sigue el gradiente de éste para llegar al objetivo. Por construcción, el camino seguido es la ruta óptima.

4.2. Diseño General

A la hora de programar este comportamiento, vamos a hacerlo sobre la plataforma JDE y en lenguaje C, como comentamos en el capítulo 2. Así, el diseño general del comportamiento podríamos resumirlo como una pequeña jerarquía muy sencilla de esquemas JDE, que distribuye todo el comportamiento en tareas o esquemas sencillos, que ejecutan alternativamente. En esta jerarquía un esquema padre, *Gradient Planning*, tras ser despertado a través de la interfaz gráfica, se encarga de capturar el escenario y lo pone a disposición de sus hijos a través de una rejilla.

Entre los hijos distinguimos un esquema perceptivo,*Gradient Builder*, que se encarga de generar y expandir el campo actualizando la rejilla. Al terminar este esquema su ejecución, *Gradient Planning* despierta a sus otros hijos. Éstos son los esquemas actuadores, *Follow Grid y VFF (Virtual Force Field)* y ejecutan concurrentemente. El esquema *Follow Grid* intenta llevar siempre el control, excepto cuando se activa *VFF*. Esto sucede cuando las variables sensoriales reflejan un obstáculo cercano que ha invadido una zona de seguridad. Cuando esto sucede, ambos esquemas intentan llevar el control de robot. Para evitar esto, el control se le otorga a *VFF* puesto que es más prioritario no chocar.Tanto *Gradient planning* como *VFF* hacen uso de las variables proporcionadas por la plataforma para comandar el movimiento del robot. Podemos ver el diseño general en la figura 4.1.

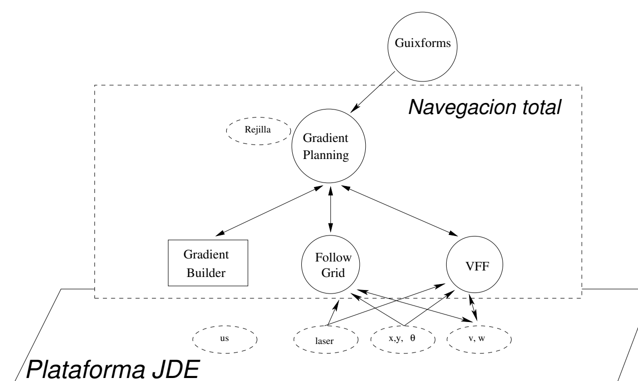


Figura 4.1: Jerarquía de esquemas del comportamiento

4.3. Esquema *Gradient Planning*

Este esquema es el que se encarga de controlar todo el comportamiento programado, para lo que activará o desactivará a cada uno de sus esquemas hijos según convenga. Las funciones que este esquema realiza son:

1. Capturar información del mundo
2. Generar la rejilla donde se almacenará el mundo
3. Despertar a sus hijos
4. Permanecer despierto por si debe arbitrar entre sus hijos

El esquema es lanzado por el usuario a través de la *GUI* de JDE, que es controlada por el esquema *Guixforms*. Una vez es lanzado, procede a efectuar sus funciones.

Para obtener la información del mundo, bien sea un escenario SRIsim, bien un escenario Player/Stage, el esquema utiliza un analizador sintáctico que abre el fichero de descripción del mundo, y lo procesa, línea por línea, obteniendo los datos necesarios. Si el simulador a utilizar es el SRIsim, la información que se captura es el alto y el ancho del mundo en milímetros, las mínimas y máximas posiciones en X e Y, la posición y orientación inicial del robot, y los puntos inicial y final que definen cada segmento del mundo. Esta información se guarda en un registro:

```
typedef struct
{
  int x_inicio;
  int y_inicio;
  int x_final;
  int y_final;
}Tsegmento;

typedef struct
{
  int ancho;
  int alto;
  int dimension;
  float resolucion;
  int x_max;
  int y_max;
  int x_min;
  int y_min;
  int n_segmentos;
  float x_robot;
  float y_robot;
  float theta_robot;
  Tsegmento segmentos[MAX];
}Tmundo;
```

Si el simulador es Player/Stage, la información capturada es el número de pixels que ocupa la imagen de alto como de ancho, la resolución con que el simulador representará esa imagen y la posición y orientación inicial del robot. De nuevo esta información se guarda en el registro anterior.

El esquema se encarga también de generar una rejilla con el tamaño necesario para abarcar todo el escenario, usando la biblioteca `Gridslib`, que ya fue comentada en el capítulo 3. Para ello usa el alto y el ancho del mundo para crear una rejilla cuadrada con el mayor de estos valores. Una vez creada la rejilla, la anclamos al mundo, es decir, la situamos sobre un punto del mundo. El punto de anclaje es el punto medio del mundo, puesto que sabemos el ancho y el alto de éste, podemos calcularlo fácilmente. Así hacemos coincidir la celda central de la rejilla con este punto. Esto es muy importante para que la rejilla sea compatible con las lecturas instantáneas de `us` y `laser`, que tienen que detectar coherentemente los obstáculos incluidos en el mapa como paredes, puertas, etc. Una vez hecho esto, se procede a rellenar la rejilla usando la información de ocupación del mapa.

Si el simulador es SRIsim, la forma de rellenar la rejilla es utilizando la información capturada de los segmentos. Puesto que las coordenadas de cada punto del mapa son absolutas, lo único que hay que hacer es una transformación, para saber a qué celda de la rejilla corresponde una coordenada. Por cada segmento, se recorre este con un paso igual a 1/3 del tamaño de celda. Se calcula en qué celda cae ese punto y se marca la celda como ocupada. Para saber en qué celda cae cada punto de espacio, hacemos la siguiente transformación :

$$Pos_x = (X_{inicio} + paso)Grid.size + X_{inicio} \tag{4.1}$$
$$Pos_y = (Y_{inicio} + paso)Grid.size + Y_{inicio} \tag{4.2}$$
$$Celda = (Pos_y)Grid.size + Pos_x \tag{4.3}$$

Podemos ver esto visualmente en la figura 4.2, en la que vemos 2 segmentos en color azul y encima de éstos, una rejilla. Estos segmentos son recorridos con un paso de 1/3 de celda, que es cada una de las divisiones rojas. Por cada una de estas divisiones, marcaríamos la celda en la que cae como ocupada.

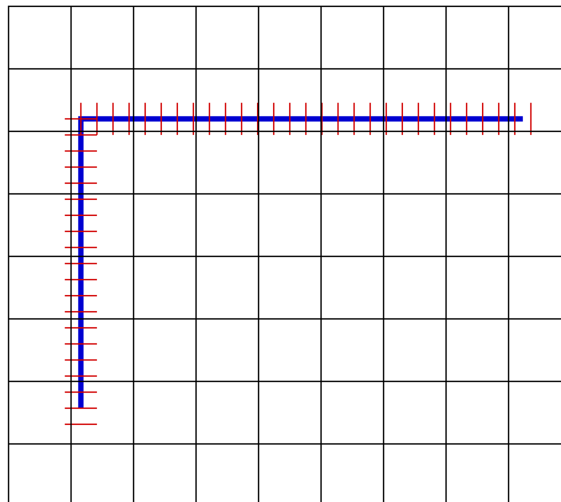


Figura 4.2: División de cada segmento en pasos.

Si el simulador es Player/Stage, lo que hacemos es recorrer cada celda viendo en a qué pixel se corresponde de la imagen. Puesto que la imagen es un *bitmap* en formato PNM, el píxel sólo puede tener dos valores, o blanco, que denota un obstáculo, o negro, que denota espacio libre, como vimos en la figura 3.7(a). Para cada celda, sabemos qué coordenadas del espacio representa, por lo que podemos también pasarlo a un píxel determinado de la imagen. Esto es así ya que sabemos que para Player la

coordenada (0,0) corresponde a la esquina inferior izquierda de la imagen. También sabemos el número de píxels que hay tanto a lo alto como a lo ancho de la imagen, y el último dato que nos falta es saber el número de centímetros que ocupa cada píxel, que es la resolución capturada. Con estos datos y siguiendo los cálculos mostrados por las ecuaciones 4.4, 4.5 y 4.6 sabemos a qué píxel corresponde cada celda. Esta celda se marca como ocupada o libre, según el valor del píxel.

$$Pos_y = \frac{(celda/Grid.size)Grid.Resolucion}{Mundo.resolucion} \quad (4.4)$$

$$Pos_x = \frac{(celdaMODGrid.size)Grid.resolucion}{Mundo.resolucion} \quad (4.5)$$

$$Pixel = (Pos_y Mundo.columnas) + Pos_x \quad (4.6)$$

Al final del proceso obtenemos una rejilla con celdas marcadas como ocupadas o libres, que representa fielmente el entorno ofrecido por el simulador, independientemente de usar SRIsim o Player/Stage. Esta rejilla se pone a disposición de los esquemas hijos, a través de una variable visible para todos ellos:

```
Tgrid *gradient_planninggrid;
```

Una vez ha capturado la información del mundo en la rejilla, despierta a su hijo *Gradient Builder*, que se encarga de generar y propagar el campo. Cuando este hijo acaba, le duerme otra vez y despierta a los hijos *Follow Grid* y *VFF*, esperando sin hacer nada a que terminen. Es posible que en algún momento deba arbitrar entre sus hijos, cuado se den las situaciones en que ambos esquemas quieran obtener el control a la vez. Ésto sólo sucederá cuando se cumplan las precondiciones de activación del esquema *VFF*. Éstas son que un obstáculo esté dentro de una "zona de seguridad", como veremos más adelante cuando detallemos este esquema. Si ésto sucede, se utiliza una función de arbitraje, que decide qué esquema hijo va a tomar el control. Esta función simplemente prioriza el comportamiento a seguir, siendo más prioritario el evitar el choque, por lo que se cede el control al esquema *VFF*

El esquema *Gradient Builder* se desactiva por medio de la GUI. Cuando esto pasa, duerme a sus hijos antes de dormirse él. Si es despertado, despierta a sus hijos, que continúan su ejecución donde la habían dejado. No se repite el proceso de generar la rejilla de nuevo.

4.4. Esquema *Gradient Builder*

Este esquema es el que se encarga de generar el campo y expandirlo a lo largo del escenario, que está representado en la variable `gradient_planninggrid`, y es puesta a disposición por el esquema padre *Gradient Planning*. Para ello se ha programado una función iterativa que desde un punto, o puntos iniciales, crea un frente de onda que se expande hasta que se cumple una condición de parada.
En esta función, la propagación se ha materializado como una lista ordenada de frentes de onda, en cuyos nodos van a estar almacenadas las celdillas que están a una "distancia" concreta del origen. Cada nodo de la lista va a representar el frente de onda asociado a esa distancia, que lo forma cada una de las celdas almacenadas en él.

Para crear y expandir el campo, partimos del nodo inicial de la lista, que será aquel que tenga distancia 0. Para cada celdilla almacenada en este nodo, se le asigna el valor de la distancia, se ven sus vecinas y se calcula qué distancia se les va a asignar, d+1 o d+1.4 (figura 4.4) siempre y cuando no hayan sido visitadas ya, o sean un obstáculo.

Ésto garantiza que el camino a cada celda sea mínimo, ya que si una celda ya ha sido visitada, no se sobreescribirá su valor. Cada una de las vecinas se introducen en el nodo de la lista que les corresponda. Una vez se ha hecho esto con todas las celdas del nodo, este nodo se borra, y se repite el proceso con las celdas del siguiente nodo de la lista.

Así, visitando todas las celdas de un nodo, es decir el frente de onda asociado a esa distancia, se va actualizando los valores del campo y creando el siguiente frente de onda. Podemos ver el pseudocódigo del algoritmo en la imagen 4.3, así como una representación gráfica de cómo funciona éste en la imagen 4.4.

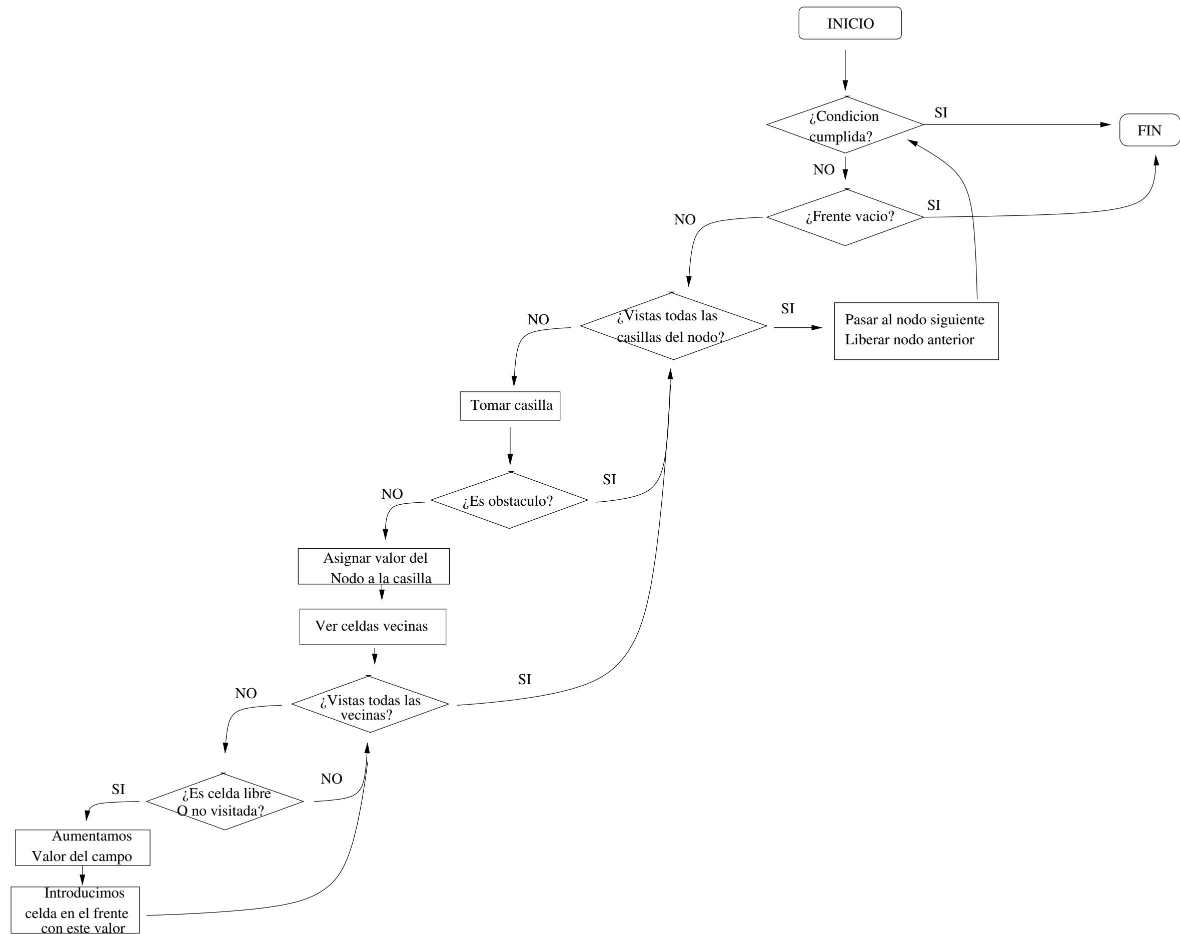


Figura 4.3: Pseudocódigo del algoritmo de expansión del frente.

Parte de las estructuras creadas para representar el frente de onda son:

```
typedef struct
{
  int fila;
  int columna;
  float estado;
}Tcasilla;

typedef struct nodo
{
  float distancia;
  Tcasilla casillas[10000];
  int casillasmax;
  struct nodo *siguiente;
```

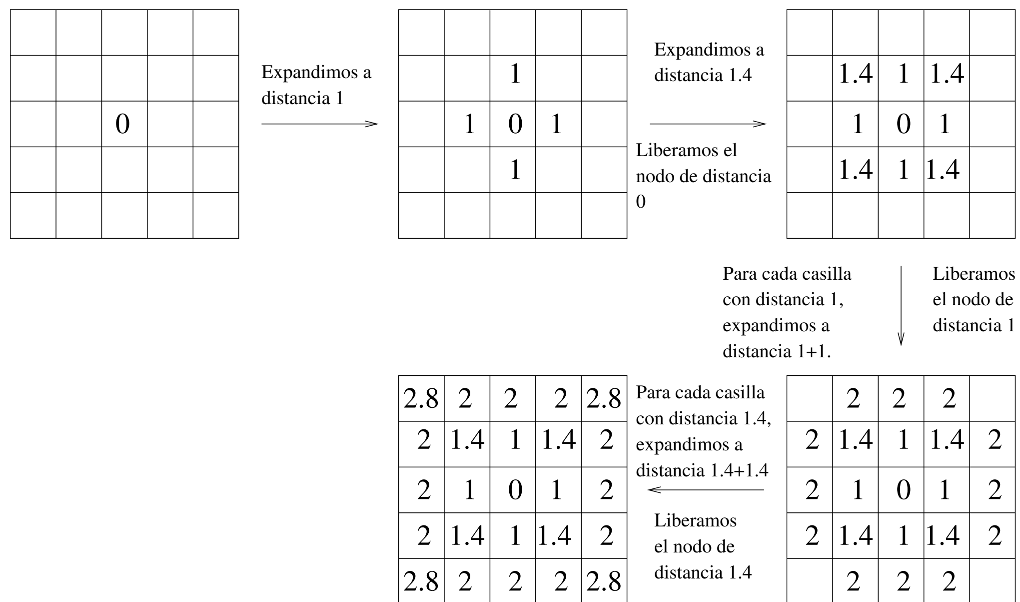


Figura 4.4: Representación de la expansión del frente en la rejilla.

```
}Tnodo;

Tnodo *lista; /* frente de onda*/
```

Y las funciones creadas para ingresar una celda en un nodo y propagar el campo son:

```
void ingresar_celdilla_en_frente(int columna,int fila, float estado,
                                 float dist)
void propagar (Tnodo *frente, int obstaculo)
```

El esquema *Gradient Builder*se ejecuta en varias fases ordenadas:

1. Crear un campo asociado a los obstáculos: Se crea una lista en cuyo nodo inicial se introduce cada celda marcada como obstáculo.

2. Expandir campo de obstáculos: Siguiendo el algoritmo antes descrito expandimos el frente de onda de los obstáculos hasta una distancia predeterminada. Esto nos creará el *campo de obstáculos*, que actuará como seguridad, alejando a robot de éstos (figura:4.6(a)-figura 4.6(c)). Este campo penalizará los movimientos cercanos a los obstáculos y se sumará al campo hacia destino, que se crea posteriormente. Este campo de obstáculos es invertido, para que los valores más altos estén cerca de los obstáculos y los más bajos lo estén más alejados de éstos.

3. Elegir destino y origen: Sobre la interfaz gráfica de la aplicación se captura el punto al que queremos que viaje el robot. Este punto gráfico es transformado a un punto del espacio real, es decir, las coordenadas pinchadas en la ventana de la GUI, a una celda de la rejilla. Una vez elegido el destino, se hace lo mismo con la posición del robot. Para hacer esto el esquema está sincronizado con la GUI y, por tanto, con el esquema *Guixforms*. Éste último es el encargado de visualizar la interfaz gráfica y capturar las coordenadas del ratón en el momento de pinchar sobre la pantalla y las transforma a coordenadas absolutas del escenario. Tras un simple cálculo, podemos saber a qué celda de la rejilla corresponden esas coordenadas.

4. Expandir campo desde destino a origen: Crea una lista nueva con el destino como único punto en el nodo inicial. Expande el campo hacia destino usando la función descrita anteriormente, hasta que el frente llegue a la celda marcada como origen (figura:4.6(d)-figura 4.6(j)). Conviene resaltar que la función de expansión del frente es la misma usada para generar el frente de los obstáculos. Lo único que cambia es la condición de parada, que en este caso es llegar a una celda particular, y en el anterior era expandir hasta una distancia cualquiera. Mientras se expande este campo hacia destino, crea a su vez el campo total, que es la suma de este campo hacia destino con el campo de obstáculos antes creado.

5. Generar ruta de referencia: Para ello usamos una función recursiva, que busca desde el origen el camino hasta el destino. La función ve el valor del campo de la celdilla en la que está el robot, mira cual de las vecinas tiene el menor valor de campo, y se invoca recursivamente para esa otra celda. La función termina cuando llega a una celda con valor 0, que por construcción es el origen del campo, y, por tanto, a donde se quiere viajar (figura:4.6(k)). Esta ruta es meramente orientativa, y nos sirve para visualizar la corrección del campo. El robot seguirá esa misma ruta, o una muy parecida. Cabe destacar el hecho de que el campo sea sumado al de los obstáculos aporta seguridad al robot al alejarlo de los obstáculos, como se ve en la ruta, que sale por el centro de la puerta. Si ésto no fuera así la ruta y, por tanto, el robot iría raspando la pared más cercana al objetivo, como se puede ver en la figura 4.5

El algoritmo creado calcula el camino de menor distancia entre dos puntos, pero incorporando los obstáculos a éste. Como se puede ver en la imagen 4.5, el punto destino está muy próximo al origen, sin embargo hay una pared entre ellos. Si siguiéramos el camino con distancia mínima euclídea, éste nos llevaría a través de la pared. Sin embargo, este algoritmo al incorporar los obstáculos hace al robot atravesar la puerta, caminar por el pasillo, y entrar en el despacho para llegar al punto objetivo.

(a) (b)

Figura 4.5: Ruta mínima generada con frente de obstáculos(a) y sin él (b)

Este esquema se ejecuta tan sólo una vez, y cuando acaba levanta un *flag* que indica al esquema padre que despierte a sus esquemas hermanos *Follow Grid* y *VFF*

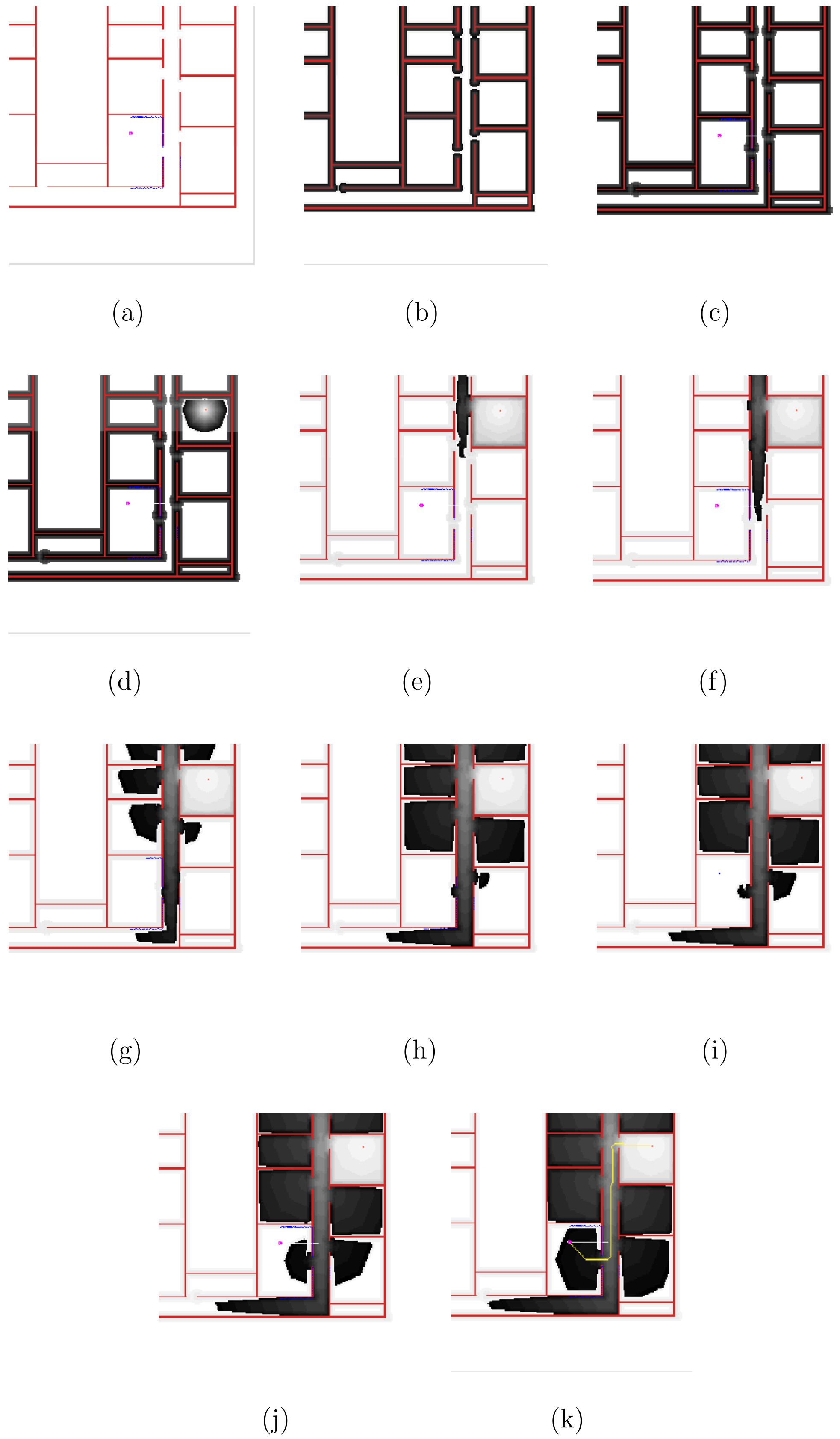


Figura 4.6: Mapa sin campo (a), Mapa con el campo de los obstáculos propagándose (b-c), Mapa con el campo propagándose (d-j) y Mapa con el campo propagado y la ruta de referencia (c).

4.5. Esquema *Follow Grid*

Este esquema se encarga de la navegación global, de pilotar al robot desde el punto en que se encuentra, hasta el destino seleccionado, utilizando para ello el campo previamente calculado por el esquema *Gradient Builder*.

Para ello materializa un controlador borroso, que nos proporciona un control reactivo, que se adapta a múltiples situaciones. El controlador borroso nos consigue suavidad en el movimiento, y que éste sea rápido. Además ha de poder compensar la inercia que lleve el robot, pues no es lo mismo frenar con una velocidad baja, que hacerlo con una velocidad elevada. Así mismo ha de tener en cuenta aspectos de seguridad como la cercanía a algún posible obstáculo. Este controlador ha sido programado utilizando la biblioteca `Fuzzylib`, como se comentó en el capítulo 3.

Para todo esto el controlador borroso programado acepta como entradas:

1. `angulo`: La diferencia entre el ángulo que lleva el robot, y el ángulo al que queremos viajar, según el campo. Para hallarla, se mira el valor del campo asignado a la celda en la que se encuentra el robot, y se decide a qué celda debería moverse al robot. Esta decisión se toma mirando el valor de sus celdas vecinas, y elije aquella celda que tenga el menor valor de ellas. No se mira el valor de las celdas con vecindad 1, sino el de las celdas con vecindad 2, lo que consigue un mejor comportamiento, como veremos en próximo capítulo. Una vez se ha elegido una celda a la que viajar, se ha de calcular cómo hacerlo. Para ello se asigna un ángulo a cada una de las celdas vecinas, según la figura 4.7. La diferencia entre este ángulo y el de orientación del robot es lo que se le pasa al controlador como entrada.
   Esta variable determinará en gran medida la velocidad de giro, ya que giraremos más rápido cuanto mayor sea la diferencia entre esa orientación deseada y la orientación actual del robot.

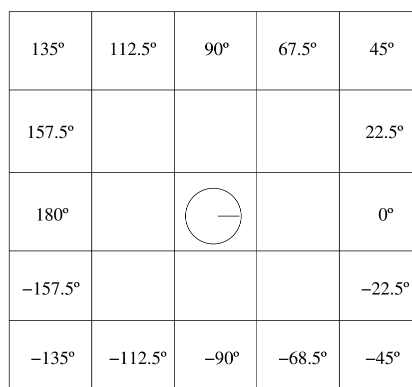


Figura 4.7: Esquema de ángulos para las 24 vecinas de la celda ocupada por el robot

2. `v_actual`: El valor actual de la velocidad lineal. Esta variable se tomará como referencia para saber si hemos de subir o bajar la velocidad lineal.

3. `w_actual`: El valor actual de la velocidad angular. Esta variable se tomará como referencia para saber si hemos de subir o bajar la velocidad angular.

4. `cerca`: El valor del campo en la posición actual. Esta variable nos indica la cercanía o lejanía al destino. Cuánto más lejos estemos de él, más rápido intentaremos ir.

5. `anticipo`: Una señal de obstáculo lejano. Cuando los sensores detectan un obstáculo a una cierta distancia esta variable hace que nos vayamos frenando, a fin de tener mayor tiempo de reacción para evitar una posible colisión.

Según los valores de estas variables de entrada, el controlador borroso recomienda unos valores de salida para la velocidad lineal y la angular, que son los que comandarán el movimiento del robot.

Todo el peso de las decisiones de movimiento para seguir el gradiente del campo lo lleva el controlador borroso, por lo que las reglas que conforman éste son críticas. Un buen controlador borroso conllevará movimientos más rápidos y suaves que otro peor. En el siguiente capítulo presentaremos las pruebas realizadas con varios controladores y cómo las diferentes variantes repercuten en el rendimiento de la navegación.

A continuación comentamos aspectos relevantes del controlador borroso finalmente elegido. En primer lugar mostraremos las etiquetas borrosas definidas sobre las variables utilizadas:

```
# Variables de entrada:

etiqueta angulo alto_pos =  90 120 150 180
etiqueta angulo medio_pos = 30 45 70 100
etiqueta angulo bajo_pos =  10 15 20 30
etiqueta angulo nulo = -5 0 0 5
etiqueta angulo bajo_neg = -30 -20 -15 -10
etiqueta angulo medio_neg = -100 -70 -45 -30
etiqueta angulo alto_neg  = -180 -150 -120 -90

etiqueta w_actual alta_pos = 40 80 100 100
etiqueta w_actual media_pos = 10 20 26 40
etiqueta w_actual baja_pos = 0 5 5 10
etiqueta w_actual nula = 0 0 0 0
etiqueta w_actual baja_neg = -10 -5 -5 0
etiqueta w_actual media_neg = -40 -26 -20 -10
etiqueta w_actual alta_neg = -100 -100 -80 -40

etiqueta v_actual nula = 0 0 0 0
etiqueta v_actual baja = 0 20 40 60
etiqueta v_actual media = 60 120 140 200
etiqueta v_actual media_alta = 200 400 600 800
etiqueta v_actual alta = 800 1500 1800 2000

etiqueta cerca mucho = 1 1 1 1
etiqueta cerca medio = 2 2 2 2
etiqueta cerca poco = 0 0 0 0

etiqueta anticipo si = 1 1 1 1
etiqueta anticipo no = 0 0 0 0

# Variables de salida:
#        velocidad_angular en grados/seg, con signo
#        velocidad_traccion en milímetros/segundo, con signo
```

```
etiqueta velocidad_angular alta_pos = 40 80 100 100
etiqueta velocidad_angular media_pos = 10 20 26 40
etiqueta velocidad_angular baja_pos = 0 5 5 10
etiqueta velocidad_angular nula = 0 0 0 0
etiqueta velocidad_angular baja_neg = -10 -5 -5 0
etiqueta velocidad_angular media_neg = -40 -26 -20 -10
etiqueta velocidad_angular alta_neg = -100 -100 -80 -40

etiqueta velocidad_traccion nula = 0 0 0 0
etiqueta velocidad_traccion baja = 0 20 40 60
etiqueta velocidad_traccion media = 60 120 140 200
etiqueta velocidad_traccion media_alta = 200 400 600 800
etiqueta velocidad_traccion alta = 800 1500 1800 2000
```

Asignamos los valores que tomará cada etiqueta de cada variable de entrada, así una diferencia angular de 115º será asignado al valor borroso "alto_pos" de la variable borrosa "angulo". Igualmente asignamos valores para las variables de salida. Dependiendo del valor asignado por las reglas a una variable, se le asigna un valor comprendido entre el mínimo y el máximo de esa etiqueta. Podemos ver las reglas para la etiqueta `velocidad_angular` gráficamente en la figura 4.8.

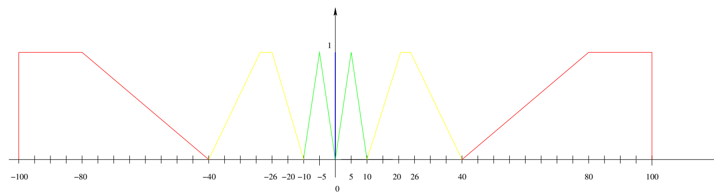


Figura 4.8: Controlador borroso para la velocidad angular

Las reglas de control que deciden el valor de salida de la velocidad angular son:

```
IF (angulo=nulo) THEN (velocidad_angular=nula)
IF (angulo=bajo_pos) THEN (velocidad_angular=media_pos)
IF (angulo=medio_pos) THEN (velocidad_angular=media_pos)
IF (angulo=alto_pos) THEN (velocidad_angular=alta_pos)
IF (angulo=bajo_neg) THEN (velocidad_angular=media_neg)
IF (angulo=medio_neg) THEN (velocidad_angular=media_neg)
IF (angulo=alto_neg) THEN (velocidad_angular=alta_neg)
```

Estas reglas siguen una idea simple para decidir el valor de la velocidad angular: cuanta mayor sea la diferencia entre el ángulo al que queremos girar (indicado por el gradiente del campo) y el ángulo en el que estamos, mayor deberá ser la velocidad de giro para compensarlo. Así aplicamos la máxima velocidad de giro cuando esta diferencia sea catalogada como máxima, y giraremos a una velocidad media en el resto de los casos, excepto cuando no debamos girar, es decir, cuando la diferencia sea catalogada como nula. Las reglas para cuando la etiqueta *angulo* toma valores negativos son completamente análogas, pero tomando valores negativos para la velocidad de giro.

Las reglas que deciden el valor de salida para la velocidad lineal son:

```
IF (angulo=nulo) AND (cerca=poco) AND (anticipo=no)
    THEN (velocidad_traccion=alta)
IF (angulo=nulo) AND (cerca=poco) AND (anticipo=si)
    THEN (velocidad_traccion=media_alta)

IF (angulo=nulo) AND (cerca=medio)
    THEN (velocidad_traccion=media)
IF (angulo=nulo) AND (cerca=mucho)
    THEN (velocidad_traccion=baja)

IF (angulo=bajo_pos) AND (v_actual=alta)
    THEN (velocidad_traccion=nula)
IF (angulo=bajo_pos) AND (v_actual=media_alta)
    THEN (velocidad_traccion=media)
IF (angulo=bajo_pos) AND (v_actual=media)
    THEN (velocidad_traccion=media)
IF (angulo=bajo_pos) AND (v_actual=baja)
    THEN (velocidad_traccion=media)
IF (angulo=bajo_pos) AND (v_actual=nula)
    THEN (velocidad_traccion=baja)

IF (angulo=medio_pos) AND (v_actual=alta)
    THEN (velocidad_traccion=nula)
IF (angulo=medio_pos) AND (v_actual=media_alta)
    THEN (velocidad_traccion=baja)
IF (angulo=medio_pos) AND (v_actual=media)
    THEN (velocidad_traccion=baja)
IF (angulo=medio_pos) AND (v_actual=baja)
    THEN (velocidad_traccion=baja)
IF (angulo=medio_pos) AND (v_actual=nula)
    THEN (velocidad_traccion=baja)

IF (angulo=alto_pos) THEN (velocidad_traccion=nula)
```

Estas reglas asignan la velocidad máxima cuando la diferencia de ángulos es nula. En caso contrario se baja la velocidad según crece la diferencia, pues es más prioritario girar rápido antes que avanzar. Aun así hay reglas que impiden alcanzar la máxima velocidad aun cuando la diferencia de ángulos sea nula. Uno de los casos es cuando se haya detectado un obstáculo en la lejanía, indicado con un *si* en la variable *anticipo*. Otro es cuando nos acercamos al destino, indicado por la variable *cerca*. Cuanto más cerca se esté del destino, más despacio se irá, para evitar "pasarse de frenada". Las reglas para cuando la etiqueta *angulo* toma valores negativos son completamente análogas.

Así pues, este controlador borroso es el encargado de dirigir el movimiento del robot siguiendo el gradiente del campo, según los valores de salida de las velocidades. Puesto que la naturaleza de un esquema es comportarse iterativamente, el controlador borroso se invoca en cada iteración del esquema. Este controlador es complementado con un "control de tracción" que impide que las aceleraciones sean demasiado bruscas. Esto impide que el robot se mueva a tirones, consiguiendo un movimiento más suave. El control de tracción se activa cuando la diferencia entre la velocidad deseada por el controlador borroso y la velocidad actual es mayor que un cierto umbral. Así, en vez

de asignar la velocidad ofrecida por el controlador, se asigna un valor más cercano a la velocidad actual, como se puede ver en la ecuación (4.7). Éste control de tracción hace que las aceleraciones sean menos bruscas, limitando los efectos de la inercia y proporcionando un mayor control en las posibles frenadas que se puedan dar, lo que genera trayectorias más suaves, como se justificará en el siguiente capítulo.

$$IF \quad (V_{deseada} - v_{actual} > 100) \quad THEN(V_{deseada} = v_{actual} + 100) \tag{4.7}$$

El controlador borroso lleva implícito en sus reglas un control de velocidad, que limita ésta según nos acercamos al objetivo. Para ello usa la información propia del campo, pues el valor de éste para el punto en el que estamos nos informa de la distancia al objetivo. Según sea este valor, el controlador recomienda una velocidad más alta o más baja.

Este esquema está en competición directa con el esquema hermano *VFF*. La decisión sobre cuál de los dos toma realmente el control es resuelta por la función de arbitraje del padre *Gradient Planning*. El esquema *Follow Grid* intenta tomar siempre el control del robot, ya que dirije al robot por el camino más corto y evitando obstáculos que aparecen en el mapa siguiendo el gradiente del campo. Sólo cederá el control cuando se active su hermano *VFF*, pues éste es más prioritario, por lo que el padre le concederá la ejecución a él.

4.6. Esquema VFF

Este esquema se encarga de implementar el algoritmo de navegación local, que pilotará al robot de manera que evite obstáculos cercanos. Para ello implementa la técnica de Campos de Fuerza Virtuales o *Virtual Force Fields (VFF)* [J. Borenstein, 1989]. Este algoritmo se basa en seguir unas fuerzas ficticias. El objetivo a viajar ejerce una fuerza atractiva sobre el robot en la dirección del destino. Cada obstáculo percibido ejerce también una fuerza repulsiva sobre el robot, mayor cuanto más cerca esté éste. El robot se moverá en la dirección que le indique la suma vectorial de la fuerza repulsiva y la atractiva. Este algoritmo combina la tendencia de avanzar hacia el destino a la vez que evita obstáculos de una forma meramente reactiva.

Así pues, este esquema es el encargado de pilotar localmente al robot, o, en otras palabras, de la evitación de obstáculos. Sólo se activa cuando un obstáculo penetra en una zona de seguridad, esto es, cuando los sensores detectan un obstáculo a una determinada distancia. La zona de seguridad se ha definido como una distancia a partir de la cual hay peligro de colisión con un obstáculo y hay que proceder a sortearlo. Como podemos ver en la figura 4.9(a), la zona de seguridad se define rectangular, pues nos interesa más reaccionar si el obstáculo está en frente que si está en uno de los lados del robot. Si la zona de seguridad hubiera sido definida redonda, como la figura 4.9(b), un obstáculo paralelo al movimiento del robot haría perturbar éste, que intentaría sortear innecesariamente dicho obstáculo. La zona de seguridad se ha definido con una distancia lateral de 350 mm y una distancia frontal de 500 mm desde el centro del robot.

$$laser[i] = 350 \quad si \quad 0 < i < 45|135 < 180$$
$$laser[i] = 500 \quad si \quad 45 \leq i \leq 135$$

Si un obstáculo entra dentro de la zona de seguridad, el esquema se activa, llamando a la función de arbitraje de *Gradient Planning*, que le da como vencedor, y *VFF* toma

el control del robot.

(a) (b)

Figura 4.9: Zonas de seguridad para VFF:cuadrada(a) y redonda(b)

Este esquema calcula en cada iteración las fuerzas repulsivas de los obstáculos percibidos por los sensores y las suma vectorialmente. Estas fuerzas se calculan como el cociente de la distancia medida por el láser entre una constante "gravitacional" K. La fuerza atractiva viene dada por el propio campo. La dirección de ésta es el ángulo asociado a la casilla a la que debemos viajar, y su módulo constante. La suma vectorial de ambas fuerzas proporciona al esquema la dirección a seguir para evitar el obstáculo y, a la vez, seguir avanzando hacia al objetivo.

$$X_{rep} = \sum_{i=0}^{180} \cos(i) * (laser[i]/K) \tag{4.8}$$

$$Y_{rep} = \sum_{i=0}^{180} \sin(i) * (laser[i]/K) \tag{4.9}$$

$$X_{Total} = X_{rep} + X_{atrac} \tag{4.10}$$

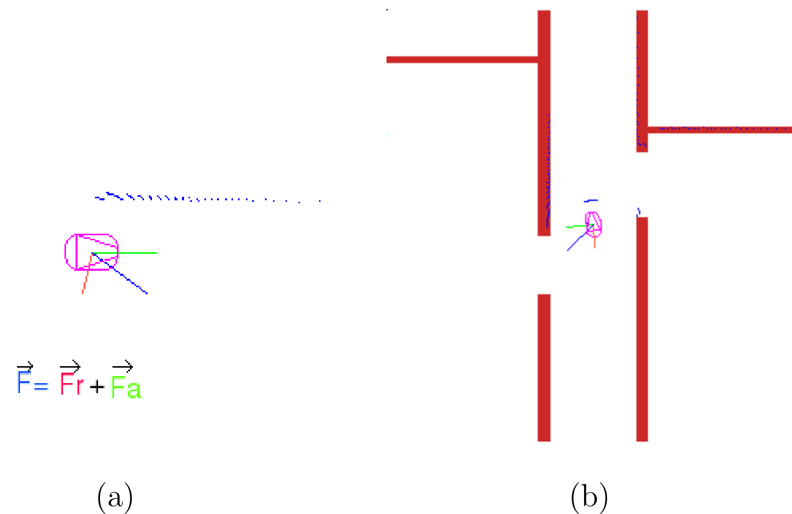
$$Y_{Total} = Y_{rep} + Y_{atrac} \tag{4.11}$$

Podemos ver un ejemplo de esto en la figura 4.10(a), donde la línea verde es la fuerza atractiva, marcada por el propio campo, la línea roja es la fuerza repulsiva, generada por la suma vectorial de la fuerza ejercida por cada punto detectado por el láser. La línea azul es la resultante de la suma vectorial de las dos anteriores, y hacia donde deberá dirigirse el robot. En la figura 4.10(b) vemos otra situación, en la que hay un obstáculo dinámico justo enfrente del robot. Vemos que la fuerza resultante obliga en este caso a retroceder al robot.

Esta forma de navegación local nos garantiza que vamos a poder esquivar los obstáculos cercanos, como pueden ser las paredes y, además, todos aquellos obstáculos no previstos, ya que éstos, aunque no aparezcan en el mapa, son detectables por los sensores del robot. Así este algoritmo de pilotaje permite reaccionar ante obstáculos imprevistos o dinámicos, evitando la colisión, y proporcionando un compromiso entre evitar obstáculos y avanzar hacia el destino.

Una vez se ha obtenido la dirección deseada de avance, usaremos para comandar las velocidades del robot un controlador borroso muy similar al de *Follow Grid*. Éste acepta como entradas:

1. `ángulo`: La diferencia entre el ángulo a viajar y el actual. Sin embargo el algoritmo no ofrece un ángulo, ofrece una fuerza resultante. Para hallar este ángulo hemos de realizar un pequeño cálculo trigonométrico. Puesto que sabemos el módulo



(a) (b)

Figura 4.10: Visualización de fuerzas que comandan el esquema VFF

de la fuerza repulsiva, podemos hallar sus componentes en cada eje. Sumando éstas a la fuerza atractiva, tenemos las componentes de la fuerza resultante. Para hallar el ángulo simplemente utilizamos la función arcotangente.

$$\theta = atan(Y_{Total}, X_{Total}) \quad (4.12)$$

2. `v_actual`: La velocidad lineal actual.

3. `w_actual`: La velocidad angular actual.

4. `peligro`: Esta señal advierte que el obstáculo está demasiado cerca, por lo que se debe girar sin avanzar, para no colisionar. Se activa cuando la distancia al obstáculo supera un cierto umbral, que lo hemos definido como 320 mm desde el centro del robot.

```
#define seguridad 320
```

La salida que ofrece el controlador son las velocidades lineales y angulares necesarias para evitar el obstáculo. Las reglas de este controlador borroso son muy parecidas al del esquema *Follow Grid*.

```
IF (ángulo=nulo) THEN (velocidad_angular=nula)
IF (ángulo=bajo_pos) THEN (velocidad_angular=media_pos)
IF (ángulo=medio_pos) THEN (velocidad_angular=media_pos)
IF (ángulo=alto_pos) THEN (velocidad_angular=alta_pos)
IF (ángulo=bajo_neg) THEN (velocidad_angular=media_neg)
IF (ángulo=medio_neg) THEN (velocidad_angular=media_neg)
IF (ángulo=alto_neg) THEN (velocidad_angular=alta_neg)
```

Estas reglas eligen el valor de salida para la velocidad angular. Como el controlador anterior, sigue la misma idea, girar más rápido cuanto mayor sea la diferencia entre ángulos.
Las reglas que eligen el valor de salida para la velocidad lineal son:

```
IF (ángulo=nulo) THEN (velocidad_traccion=alta)

IF (ángulo=bajo_pos) AND (v_actual=alta) AND (peligro=bajo)
       THEN (velocidad_traccion=baja)
IF (ángulo=bajo_pos) AND (v_actual=media) AND (peligro=bajo)
       THEN (velocidad_traccion=media)
IF (ángulo=bajo_pos) AND (v_actual=baja) AND (peligro=bajo)
       THEN (velocidad_traccion=media)
IF (ángulo=bajo_pos) AND (v_actual=nula) AND (peligro=bajo)
       THEN (velocidad_traccion=baja)

IF (ángulo=alto_neg) THEN (velocidad_traccion=nula)

IF (ángulo=bajo_pos) AND (v_actual=alta) AND (peligro=alto)
       THEN (velocidad_traccion=nula)
IF (ángulo=bajo_pos) AND (v_actual=media) AND (peligro=alto)
       THEN (velocidad_traccion=nula)
IF (ángulo=bajo_pos) AND (v_actual=baja) AND (peligro=alto)
       THEN (velocidad_traccion=nula)
IF (ángulo=bajo_pos) AND (v_actual=nula) AND (peligro=alto)
       THEN (velocidad_traccion=nula)
```

Igualmente las reglas de este controlador siguen la idea del anterior, dar más prioridad al giro que a avanzar si la diferencia de ángulos crece. Estas reglas además tienen en cuenta si un obstáculo está demasiado cerca, por lo que sólo se deberá girar sin avanzar, para evitar colisionar con éste. Sólo se muestran las reglas para el valor de *ángulo=bajo_pos*, pues las reglas para el resto de valores de esa etiqueta son idénticas.

Este controlador borroso comanda el movimiento para alejarse de los obstáculos a la vez que se acerca al objetivo final. Puesto que el primer objetivo de este esquema es evitar obstáculos, la velocidad punta que recomienda es bastante más limitada que la recomendada por el esquema *Follow Grid*, a fin de tener más tiempo para poder esquivar los posibles obstáculos. Esto lo determina el controlador borroso. Una vez se ha evitado el obstáculo, la zona de seguridad ya no está ocupada, por lo que las condiciones que se daban para que tomara el control, no se cumplen. Esto hace que el esquema se duerma, lo que devuelve el control a *Follow Grid*.

4.7. Esquema *Guixforms*

Este esquema es el encargado de dibujar y controlar la interfaz gráfica de la aplicación. Esta interfaz está programada usando la biblioteca Xforms y se ejecuta como un esquema más de JDE.

Este esquema se encarga de tomar los datos sensoriales proporcionados por la plataforma JDE y los dibuja en el canvas. También dibuja en el canvas la rejilla de ocupación que crea el esquema *Gradient Planning*. Este esquema no ha sido programado desde cero, ya que se ha modificado el proporcionado por JDE para crear un *canvas* más grande, quitar funciones innecesarias para nuestro proyecto y dotar a la interfaz de los elementos genuinos para nuestra aplicación. Al ser un esquema más, se ejecuta iterativamente, y en cada iteración comprueba el estado de los elementos de la interfaz, actuando de una forma u otra según sea éste.

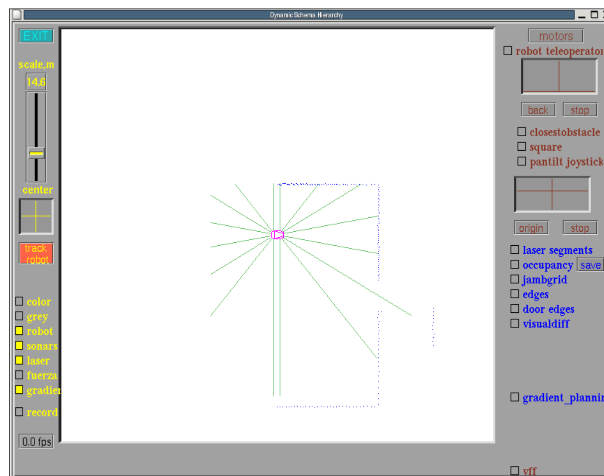


Figura 4.11: Interfaz gráfica de la aplicación

Desde este esquema, y pulsando en el botón adecuado, podemos visualizar las medidas de distintos sensores, comandar los actuadores o poner en marcha o dormir el comportamiento programado. Además, a través de la interfaz proporcionada por este esquema, podremos indicarle a la aplicación el destino a viajar, pinchando sobre un punto del mapa, desde donde se expandirá el campo para llevar el robot a ese punto.

Los botones que activan los esquemas sensoriales más reseñables son:

- `Laser`: Muestra u oculta las medidas sensoriales recogidas por el laser.
- `Sonars`: Muestra u oculta las medidas sensoriales recogidas por los sensores Sónar.
- `Robot`: Muestra u oculta al robot.
- `Fuerza`: Muestra u oculta las fuerzas virtuales que comandan el esquema *VFF*
- `Gradiente`: Muestra u oculta los valores del campo de navegación.

Los botones que activan los esquemas motores más reseñables son:

- `Motors`: Activa o desactiva los motores.
- `Teleoperator`: Simula un joystick para teleoperar el robot. Este teleoperador controla las velocidades lineal y angular del robot. La velocidad lineal se controla moviendo el "punto de mira" en sentido vertical, más alta mientras más arriba esté éste. La velocidad angulas se controla moviéndolo en sentido horizontal, girando a un lado o al otro según lo movamos a la izquierda o a la derecha.
- `Gradient Planning`: Activa o desactiva el comportamiento programado en este proyecto.

Capítulo 5

Resultados Experimentales

En el capítulo anterior hemos comentado la solución final que hemos programado. En este capítulo explicaremos qué pruebas y experimentos hemos realizado para validar y mejorar la implementación en el camino hacia esta solución, según los distintos prototipos desarrollados iterativamente, como vimos en el capítulo 2. Para ello primero comentaremos una ejecución típica del algoritmo. Luego presentaremos y explicaremos la necesidad del campo antiobstáculos. Por último comentaremos algunas mejoras añadidas al algoritmo final y cómo influyen en su comportamiento.

5.1. Ejecución típica

En la figura 5.1 vemos una ejecución típica del algoritmo de navegación. Como se comentó en el capítulo 4, éste se divide en varios esquemas que son los que comandan el movimiento. El campo ha sido propagado desde el punto objetivo hasta la posición del robot en ese momento. El campo de cada punto del espacio se representa con un color. Los colores más próximos al blanco son los puntos más cercanos al destino, y los colores más próximos al negro los más lejanos, con toda la escala de grises entre medias de estos puntos. La ruta de referencia hallada siguiendo el gradiente puro se representa como un camino amarillo. La ruta realmente descrita por el algoritmo de pilotaje, se representa como un camino de dos colores, verde si el que comandaba el movimiento es el esquema *Follow Grid*, y rojo si el que lo hacia era el esquema *VFF*.

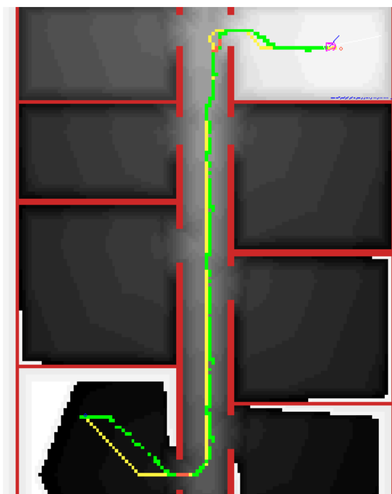


Figura 5.1: Ejecución típica de navegación.

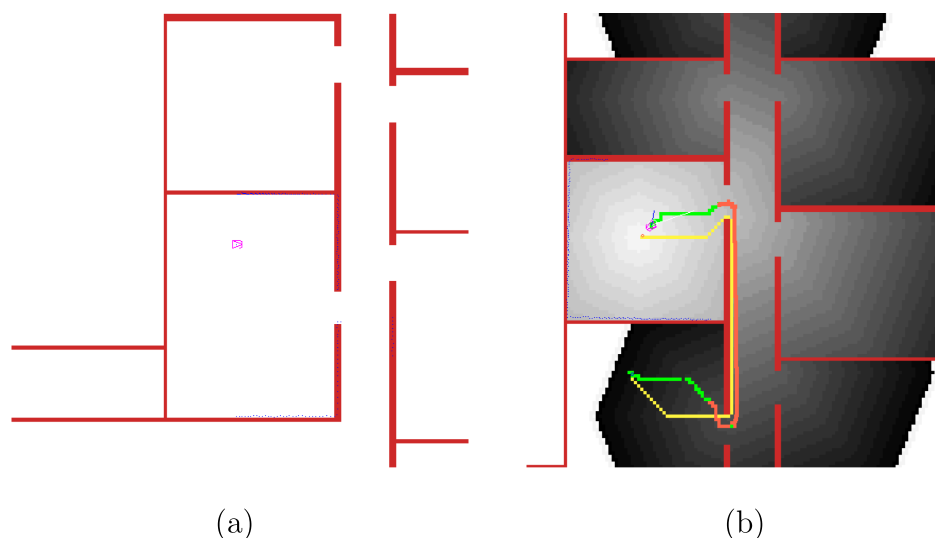
Así según la 5.1 vemos que el esquema *Follow Grid*, encargado de la navegación global, es el que toma en control en la mayoria del tiempo. *VFF* sólo se activa en puntuales ocasiones en las que se pasa demasiado cerca de un obstáculo, o en pasajes estrechos como pueden ser las puertas. Con ésto conseguimos una navegación deliberativa combinada con una reactiva, que se activa sólo cuando realmente se necesita.

Respecto a la velocidad alcanzada, hemos conseguido que el robot alcance los 1800 mm/seg o 1.8 m/seg de velocidad punta. La velocidad media de los trayectos está en torno a 1.3 m/s, consiguiendo, por tanto, una velocidad elevada con una gran seguridad en el movimiento, ya que el algoritmo de evitación de obstáculos toma el control en muy pocas ocasiones, lo que significa que podemos ir a altas velocidades con la certeza de no colisionar con obstáculos previstos por el mapa.

En distintos experimentos hemos comprobado que cuando el robot se encuentra con muchos obstáculos la velocidad de crucero se ve disminuida para poder evitarlos mejor, pués como comentamos en el capítulo anterior, el controlador borroso de *VFF* no permite coger velocidades demasiado elevadas. Si nos encontramos con muchos obstáculos dinámicos o imprevistos, evidentemente, la velocidad media baja, pues es más prioritaria la seguridad ante las posibles colisiones que la velocidad.

5.2. Campo antiobstáculos

En el capítulo 4 se comentó que los obstáculos tenían asociados un campo propio, que se sumaba al de propagación. Éste campo impedía que la ruta óptima pasara demasiado cerca de los obstáculos, proporcionando una mayor seguridad al movimiento. Se han realizado múltiples experimentos para ver a qué distancia máxima habría que propagar este campo asociado a los obstáculos. Un campo demasiado pequeño haría que el robot pasara demasiado cerca de los obstáculos. Experimentalmente se ha llegado a una condición que optimiza este campo. En las figuras 5.2, 5.4 y 5.3 podemos ver distintos experimentos hechos para localizar el punto de equilibrio.

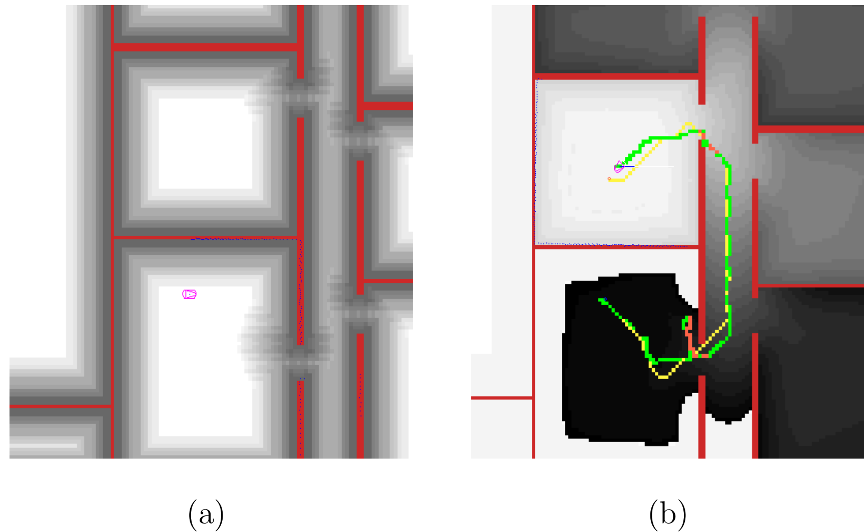


(a) (b)

Figura 5.2: Mapa con campo nulo (a) y ruta seguida (b).

En la figura 5.2 se usa un campo demasiado pequeño, por lo que la ruta lleva al

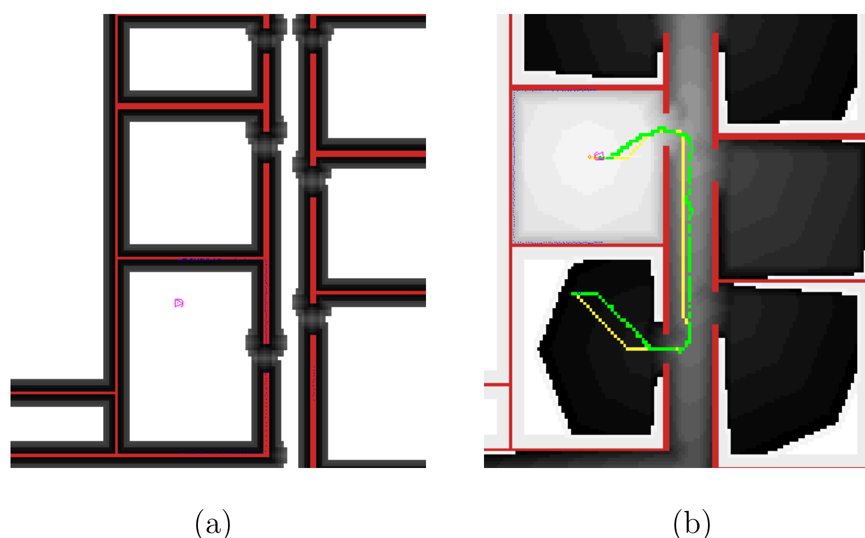
robot muy próximo a la pared. Como se puede observar el algoritmo de pilotaje evita la colisión contra las paredes cediendo el control al esquema *VFF*. Esto lleva al robot a su objetivo, pero de una manera más lenta que si tomara el control *Follow Grid*, ya que la seguridad de evitar una colisión es más importante que la velocidad en el movimiento.



(a) (b)

Figura 5.3: Mapa con campo alto (a) y ruta seguida (b)

En la figura 5.3 se usa un campo de obstáculos muy grande. Este campo abarca la totalidad de los pasillos, lo que ralentiza la expansión del campo, pero asegura que la ruta óptima pase por el centro de éstos. Sin embargo, como se puede ver en la figura 5.3(b), el algoritmo de pilotaje tiene problemas para salir por las puertas, realizando un rizo al intentar por la primera puerta. Esto es así porque la suma del campo de obstáculos y el campo propagado toma un valor demasiado elevado, lo que el algoritmo entiende que es "peligroso" pasar por ahí e intenta encontrar otro camino.



(a) (b)

Figura 5.4: Mapa con campo medio (a) y ruta seguida (b)

En la figura 5.4 vemos el campo usado finalmente. Este campo es lo suficientemente potente para alejar al robot de los obstáculos, pero no tanto como para que la suma con el campo propagado produzca problemas como los vistos en 5.3. Vemos que el algoritmo de pilotaje se ajusta casi perfectamente a la ruta de referencia. Experimentalmente se ha visto que propagar el campo de los obstáculos hasta una distancia de 17 celdas proporciona el mejor rendimiento velocidad-seguridad.

5.3. Evitación de obstáculos imprevistos

Uno de los objetivos de este proyecto era evitar obstáculos que no aparecieran en el mapa. Para poder probar que esto se cumple satisfactoriamente se han realizado múltiples pruebas con obstáculos dinámicos. Éstos obstáculos se han conseguido utilizando el simulador ofrecido por Player/Stage, como se comentó en el capítulo 3. Las pruebas realizadas han consistido en tener un robot manejado por nuestro algoritmo de navegación y otro teleoperado manualmente que se le cruzaba en su camino para probar que no colisionaban entre si.

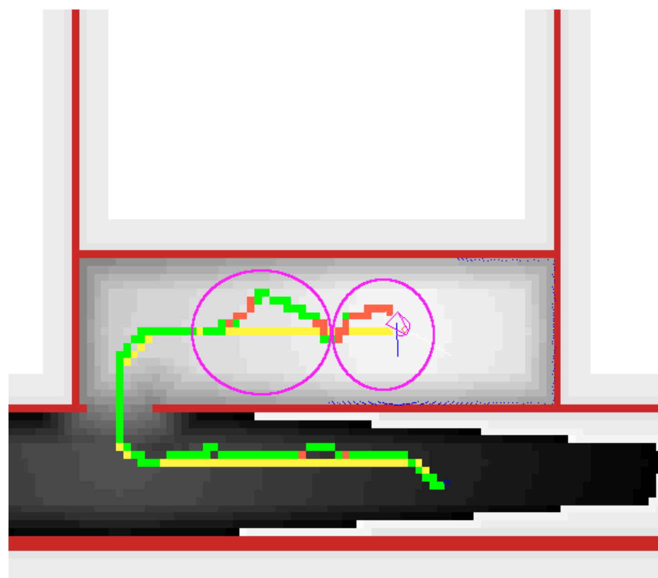


Figura 5.5: Ruta afectada por un obstáculo imprevisto

En la figura 5.5 podemos ver una situación en la que el esquema *VFF* ha tenido que tomar el control, para evitar un obstáculo imprevisto. En dicha figura, vemos como se han evitado 2 obstáculos consecutivos, al llegar al objetivo. *VFF* toma el control, lo que gráficamente se ve como una trayectoria en color rojo. Vemos que tras evitar el primer obstáculo, el algoritmo de pilotaje lleva al robot a la ruta óptima, pero se desvía de ésta de nuevo porque se ha situado otro obstáculo imprevisto. Así *VFF* debe entrar de nuevo en acción para evitarlo.

En la figura 5.6 podemos ver otra situación similar a la anterior. En ésta han actuado tres obstáculos imprevistos, uno antes de salir de la habitación origen, otro antes de entrar en la habitación de destino, y otro último dentro de esta habitación. De nuevo podemos ver como *VFF* toma el control del pilotaje y evita los obstáculos desviándose de su ruta. Podemos ver en esta imagen que los obstáculos se han simulado con otro robot, que se interponía en el camino del que estaba guiado por nuestra aplicación,

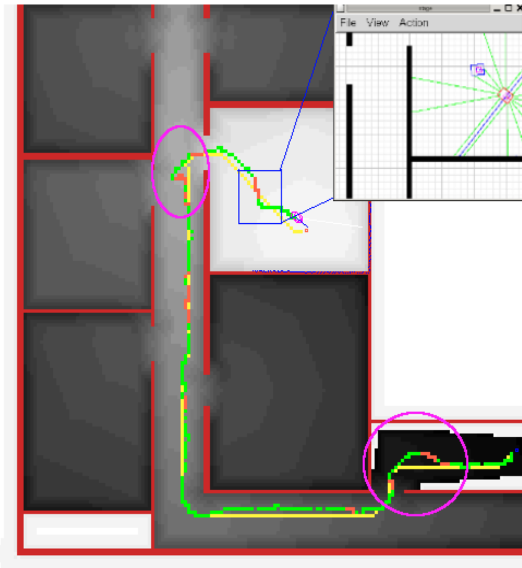


Figura 5.6: Ruta afectada por un obstáculo imprevisto

como se ve en la imagen, que muestra el entorno de Stage, en el que un robot Pioneer azul teleoperado, ha sido el que ha provocado el desvío observado a un Pioneer rojo, que estaba guiado por nuestra aplicación.

5.4. Mejoras de la navegación

En esta sección comentamos tres mejoras que hemos realizado al algoritmo original de pilotaje, para conseguir una mayor suavidad en su movimiento.

Las tres mejoras que se han realizado han sido:

1. Usar vecindad-2 (5x5) para ver cual es la siguiente celda a la que moverse en vez de vecindad-1 (3x3).

2. Control de tracción para evitar aceleraciones bruscas.

3. Limitador de velocidad al llegar al objetivo.

Las baterías de pruebas que hemos realizado han probado el algoritmo bajo distintas condiciones, y eliminando cada una de las mejoras al algoritmo cada vez, para ver cómo influía ésta en el comportamiento final. Las pruebas han sido realizadas sobre 5 rutas distintas.

Cada prueba muestra el campo entre el destino a viajar y el origen donde estaba situado el robot. Como antes, la ruta de referencia está marcado como un camino amarillo. La ruta realmente seguida está marcada como un camino de colores, verde si ha sido el esquema *Follow Grid* el que comandaba el movimiento, y roja, si era *VFF* el que lo hacía.
La bondad de una ruta será elegida por 3 parámetros.

- El tiempo tardado en recorrerla. Será mejor cuanto menor sea éste.

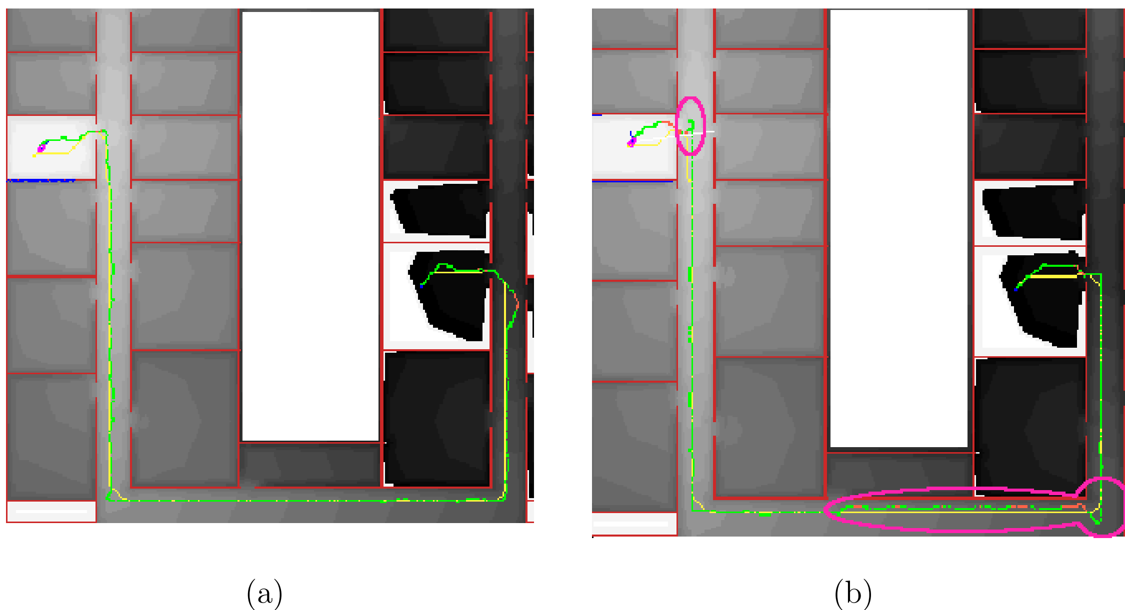
- El coste acumulado por la ruta seguida. Por coste acumulado entendemos la suma de cada uno de los valores de las celdas visitadas en el camino hacia el objetivo.

Este valor es similar a la distancia recorrida, pero penaliza el paso por zonas cercanas a los obstáculos.

- La diferencia entre este coste acumulado y el de la ruta de referencia, calculada por el descenso de gradiente puro.

Tras ver los resultados que mostraremos a continuación, podemos llegar a la conclusión que tener estas tres ayudas activadas a la vez proporciona un movimiento más estable y suave, que se ajusta a la ruta de referencia, aun a sacrificio tal vez de la velocidad, pero ganando en estabilidad y seguridad.

5.4.1. Vecindad-1 VS Vecindad-2



(a) (b)

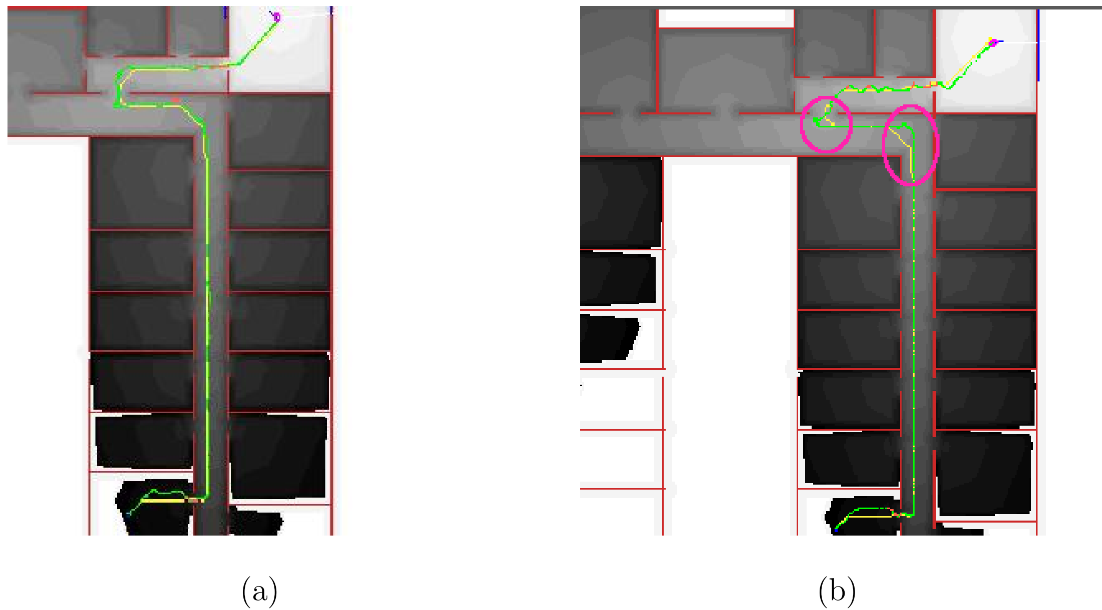
Figura 5.7: Ruta seguida usando Vecindad 2 (a), y Ruta usando Vecindad 1 (b)

En las figuras 5.7 y 5.8 vemos dos ejemplo típico de estas pruebas. En las figuras 5.7(a) y 5.8(a), se ha seguido la ruta usando vecindad-2, mientras que en las figuras 5.7(b) y 5.8(b) se ha seguido vecindad-1. En ambos casos el resto de mejoras están activas.

Como podemos ver, en 5.7(b) la ruta seguida es más inestable, produciéndose oscilaciones cerca de la pared en el pasillo inferior. También vemos como se pasa de frenada al intentar entrar en la habitación donde está el objetivo, teniendo que retroceder. Estas mismas oscilaciones las vemos en 5.8(b), en el pasillo superior, así como la pasada de frenada.

Sin embargo si nos fijamos en 5.7(a), vemos que la ruta seguida se ajusta más a la ruta ideal, no oscilando como lo hace en 5.7(b). De igual manera en 5.8(a) vemos que se ajusta mejor a la ruta de referencia, de tal manera que no se producen oscilaciones ni pasadas de frenada.

En la tabla 5.4.1 observamos la bondad de una ruta respecto a otra de manera más objetiva para cuantificar cuánto afecta la mejora al algoritmo.



(a) (b)

Figura 5.8: Ruta seguida usando Vecindad 2 (a), y Ruta usando Vecindad 1 (b)

	Tiempo (seg)	Coste de referencia	Coste real	Diferencia
Vecindad 2 (fig 5.7(a))	164	221678	229396	03,48 %
Vecindad 1 (fig 5.7(b))	212	216753	230090	06,15 %
Vecindad 2 (fig 5.8(a))	183	119290	124542	04,40 %
Vecindad 1 (fig 5.8(b))	189	118929	137098	15,27 %

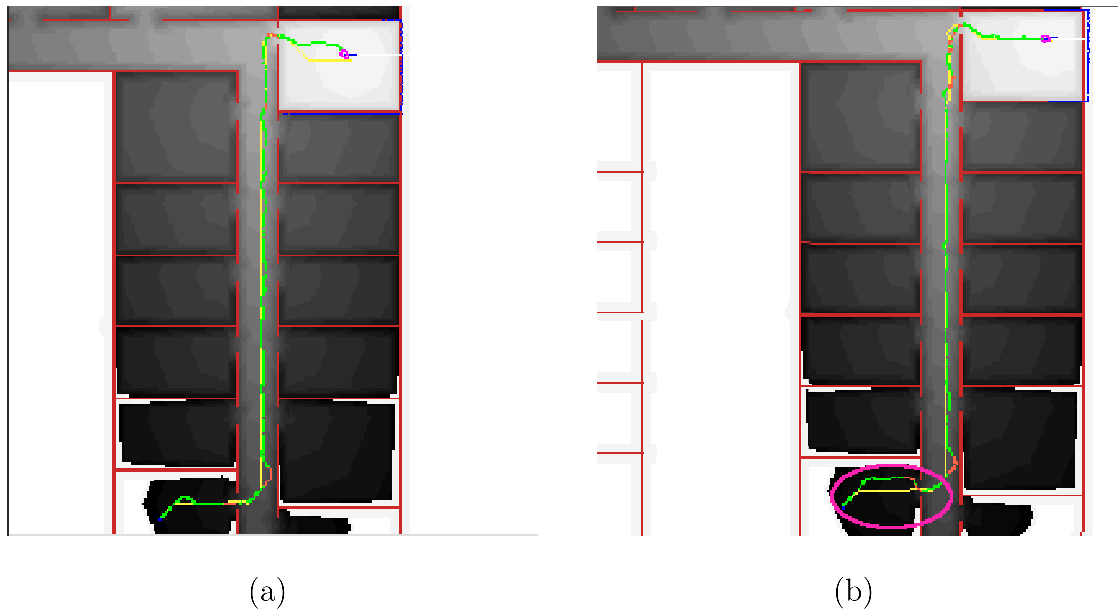
Cuadro 5.1: Tabla resumen de las Rutas de prueba

Estos experimentos son los más representativos de todos los realizados, y a partir de ellos podemos sacar la siguiente conclusión: El utilizar vecindad 2 en vez de vecindad 1 proporciona una mayor seguridad y estabilidad al movimiento del robot. Puesto que se decide dónde ir con una celda de "antelación", en caso de un giro brusco el robot tiene más tiempo para reaccionar, puede frenar antes y empezar a girar también antes, por lo que su movimiento es más suave. Tener un mayor horizonte ayuda a tomar una mejor decisión en el movimiento, resultando éste más suave y preciso.

5.4.2. Control de tracción

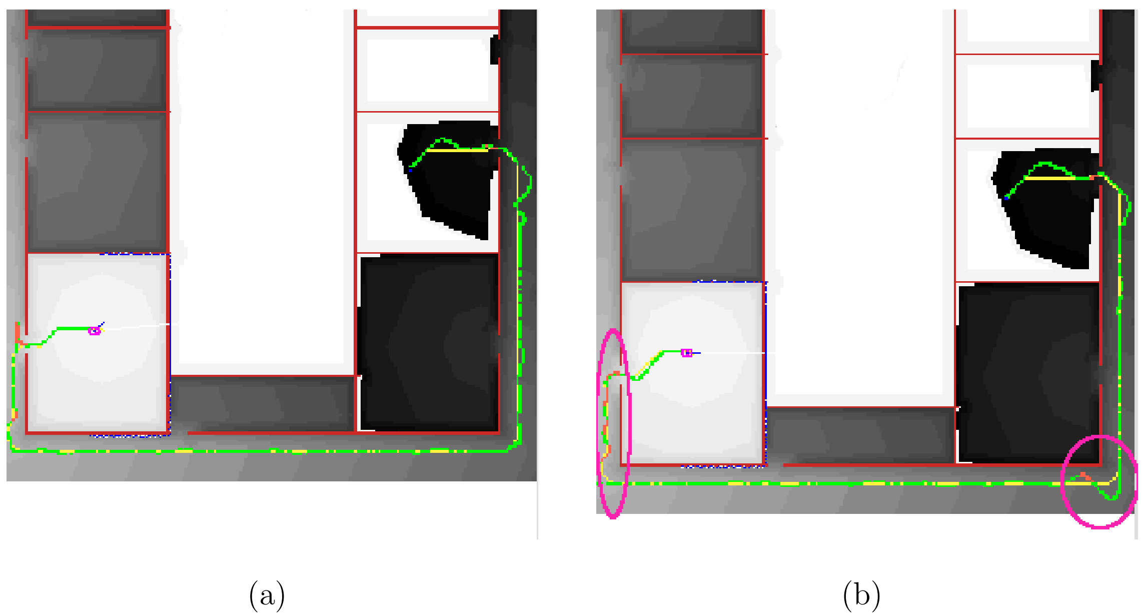
En las figuras 5.9 y 5.10 vemos dos ejemplo típico de estas pruebas. En las figuras 5.9(a) y 5.10(a), se ha seguido la ruta usando el control de tracción, mientras que en las figuras 5.9(b) y 5.10(b) no se ha usado. En ambos casos el resto de mejoras están activas.

Como podemos ver en la figura 5.9(b) el arranque del robot es un arranque "descontrolado". Esto es porque alcanza la velocidad máxima en un corto espacio y tiene que frenar muy bruscamente, produciéndose un arranque demasiado brioso. Además, como podemos observar en la figura 5.10(b), las frenadas sin el control de tracción son más inexactas, teniendo que corregir bruscamente la dirección del robot. Igualmente vemos que también puede crear oscilaciones en el movimiento, ya que acelera y frena



(a) (b)

Figura 5.9: Ruta seguida con control de tracción (a), y Ruta sin control de tracción (b)



(a) (b)

Figura 5.10: Ruta seguida con control de tracción (a), y Ruta seguida sin control de tracción (b)

bruscamente, descontrolando el movimiento del robot, como se puede ver en la figura 5.10(b), al llegar al objetivo.

Control de tracción	Tiempo (seg)	Coste de referencia	Coste real	Diferencia
Con (fig 5.9(a))	127	72832	82747	13,61 %
Sin (fig 5.9(b))	160	74837	90542	20,98 %
Con (fig 5.10(a))	213	131986	148286	12,35 %
Sin (fig 5.10(b))	170	133143	158464	19,02 %

Cuadro 5.2: Tabla resumen de las Rutas de prueba

A la luz de los datos mostrados por la tabla 5.4.2 podemos sacar las siguientes conclusiones: Usando el control de tracción vemos que la trayectoria seguida en las figuras 5.9(a) y 5.10(a) es más acorde con la ideal. El arranque no es tan brusco y las frenadas están más apuradas. Esto es así porque el control de tracción limita la aceleración, evitando que ésta sea demasiado brusca y haciéndola más contínua.

El utilizar el control de tracción evita pues aceleraciones y desaceleraciones demasiado bruscas, y por tanto, proporciona también un movimiento más estable. Así el robot ve limitada su aceleración neta, pero gana en estabilidad. Este control limita los efectos de la inercia, pues con una aceleración fuerte es más difícil frenar. Sin embargo si se acelera poco a poco para llegar a la velocidad máxima, si en cualquier momento se ha de frenar, ésto será más fácil.

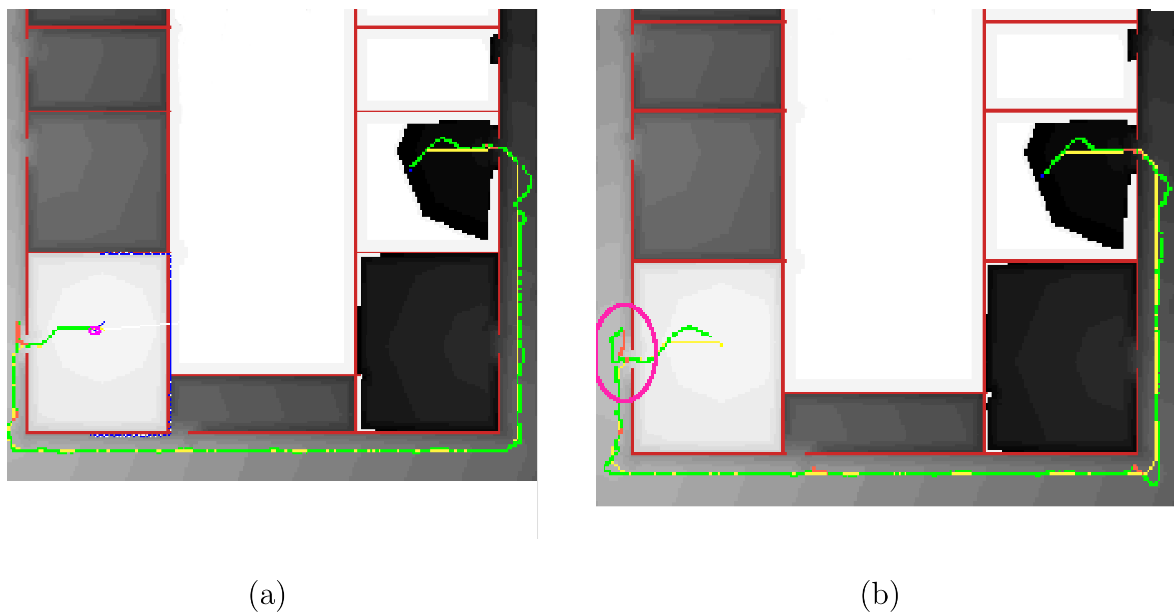
5.4.3. Limitador de velocidad

En las figuras 5.11 y 5.12 vemos dos ejemplo típico de estas pruebas. En las figuras 5.11(a) y 5.12(a), se ha seguido la ruta usando el limitador de velocidad, mientras que en las figuras 5.11(b) y 5.12(b) no se ha usado. En ambos casos el resto de mejoras están activas.

Observamos en la figura 5.11(b) que, sin el control de velocidad, llegamos a las inmediaciones del objetivo demasiado rápido, por lo que el algoritmo de pilotaje debe dar la vuelta para entrar en el despacho adecuado. Además en la figura 5.12(b) vemos que el robot pasa de largo del objetivo, teniendo que dar la vuelta para llegar a él. Ésto es porque la velocidad es demasiada, y no le da tiempo al robot a frenar lo suficiente en las inmediaciones del objetivo. Como se puede observar en las figuras 5.11(a) y 5.12(a) usando el limitador de velocidad, esto no sucede.

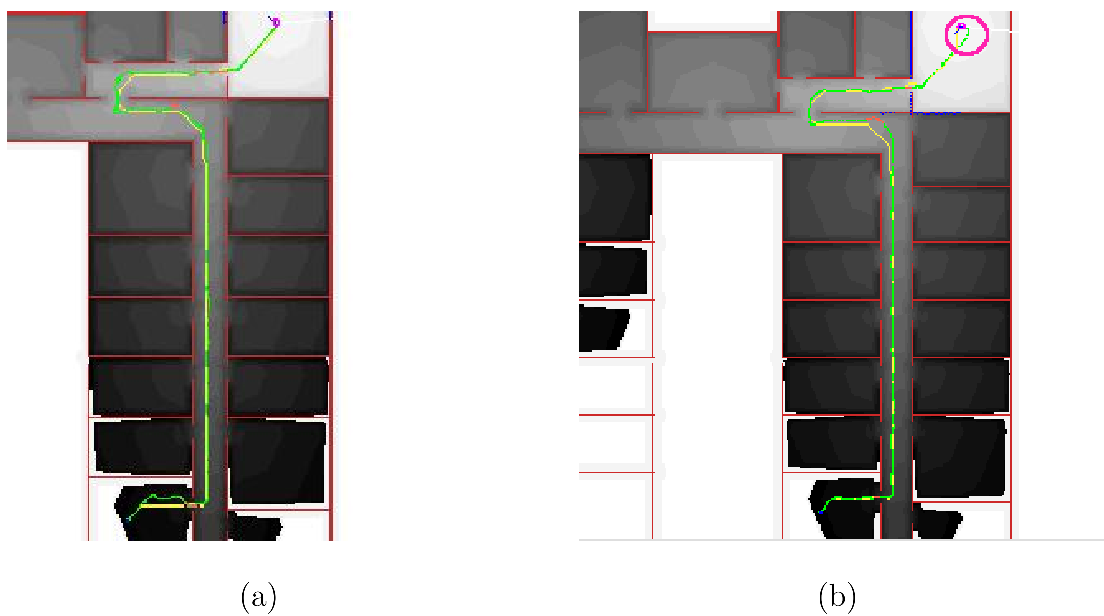
El limitador de velocidad no es más que el robot sepa que está llegando al objetivo, y baje su velocidad. Ésto asegura que el robot no se va a pasar el punto objetivo por ir demasiado rápido. Al descender su velocidad sacrificamos tiempo, pero ganamos seguridad en el movimiento.

A la vista de esta tabla, podemos ver numéricamente lo que ya se apuntaba viendo las capturas de cada ruta: Que las ayudas ajustan mejor a la ruta calculada matemáticamente siguiendo el gradiente del campo escalar. El porcentaje de desviación es menor con todas las ayudas activadas, por lo que es la ruta que más ajusta a la óptima.



(a) (b)

Figura 5.11: Ruta con limitador de velocidad (a), y Ruta sin limitador de velocidad (b)



(a) (b)

Figura 5.12: Ruta seguida con limitador de velocidad (a), y Ruta sin limitador de velocidad (b)

Limitador de velocidad	Tiempo (seg)	Coste de referencia	Coste real	Diferencia
Con (fig 5.11(a))	213	131986	148286	12,35 %
Sin (fig 5.11(b))	208	133222	160998	20,85 %
Con (fig 5.12(a))	183	119290	86187	8,49 %
Sin (fig 5.12(b))	187	120676	136946	13,48 %

Cuadro 5.3: Tabla resumen de las Rutas de prueba

Capítulo 6

Conclusiones y trabajos futuros

Una vez descrita la solución propuesta para este comportamiento, y algunos experimentos relevantes, terminamos esta memoria resumiendo los principales aportes de este proyecto y sus posibles líneas futuras de continuación.

6.1. Conclusiones

Como conclusión global destacamos que se han cumplido los objetivos expuestos en el capítulo 2 de esta memoria. Veámoslos uno a uno:

- *Desarrollar un algoritmo de navegación global.* Para ello hemos implementado el algoritmo de navegación usando el método del Gradiente [Konolige, 2000]. Como se comentó en el capítulo 4 de esta memoria se ha programado una estructura de esquemas para que expanda el gradiente desde el punto objetivo deseado hasta la posición del robot y haga navegar a éste siguiendo la información de este campo escalar.

- *Desarrollar un algoritmo de navegación local.* Para ello se ha implementado el algoritmo de navegación local *VFF* [J. Borenstein, 1989]. Tal y como se vio en el capítulo 4 de la memoria, se ha programado este algoritmo en un esquema, que se encarga de llevar al robot lejos de los obstáculos a la par que avanza hacia el destino.

- *Combinar ambos algoritmos para tener una navegación completa.* De nuevo en el capítulo 4 explicamos cómo se ha conseguido ésto, que no ha sido más que combinar ambos comportamientos como esquemas de JDE. Las precondiciones de actuación de estos esquemas son disjuntas, pués *Gradient Planning* intenta tomar el control siempre y *VFF* se activa sólo cuando un obstáculo rebasa una cierta zona de seguridad. Por esto puede darse el caso que ambos esquemas quieran ejecutar a la vez. Si esto sucede se cede el control al esquema *VFF* pues es más prioritario evitar una colisión. Las pruebas mostradas en el capítulo 5 nos han servido para validar la corrección de los esquemas creados.

- *Ampliar JDE de tal manera que acepte la plataforma de simulación de robots Player/Stage.* Para resolver esto se ha modificado el servidor Otos, añadiéndole una nueva hebra para la comunicación con *Player/Stage*, como se vio en el capítulo 3. Esta nueva hebra se encarga de traducir las medidas sensoriales recibidas desde *Player* al formato *Otos*. Igualmente las órdenes de actuación recibidas por *Otos* son traducidas al formato *Stage* para materializarlas.

Estos objetivos tenían implícitos una serie de requisitos que debían de ser cumplidos también. Éstos requisitos eran:

- *Trabajar sobre simuladores.* Este requisito es necesario ya que necesitamos saber en todo momento la posición exacta del robot. Los sensores de posición de los robots reales no son totalmente exactos, introduciendo incertidumbre en la posición final del robot. Este requisito es cumplido por la solución propuesta, que funciona tanto en el simulador SRIsim, como en el simulador *Player/Stage*.

- *Desarrollar el comportamiento usando la plataforma JDE.* Este requisito impone el lenguaje de programación, que es C, así como la programación en esquemas. La solución propuesta está programada en C y es una jerarquía de esquemas, que ejecutan bajo JDE.

- *La navegación ha de ser vivaz.* Este requisito se ha cumplido satisfactoriamente, ya que el robot toma decisiones de movimiento en tiempo real, evitando obstáculos móviles. Los esquemas *Follow Grid* y *VFF* realizan una iteración cada 100 milisegundos, por lo que tenemos 10 iteraciones por segundo.

Recapitulando, se ha programado un algoritmo de navegación total para un robot Pioneer. Este algoritmo calcula la ruta óptima hasta un objetivo y conduce al robot a través de su entorno evitando obstáculos previstos e imprevistos, combinando algoritmos de navegación global y local. Este programa funciona sobre los dos simuladores usados por el grupo de robótica. Además se ha ampliado el servidor Otos para que soporte el nuevo simulador *Player/Stage*. Este simulador ha sido necesario usarlo porque es capaz de simular objetos dinámicos y con ello, obstáculos imprevistos para nuestro algoritmo de navegación local.

En cuanto a los conocimientos aportados por el proyecto, podemos destacar los conocimientos sobre la técnicas utilizadas actualmente para sistemas de navegación, tanto globales como locales. Igualmente se han adquirido conocimientos de lógica borrosa, necesarios para crear y utilizar controladores borrosos usados en los esquemas de pilotaje.

En cuanto a las herramientas utilizadas, hemos aprendido a utilizar la plataforma JDE para el desarrollo de comportamientos autónomos sobre el robot Pioneer. Además, este proyecto ha dotado de nuevas posibilidades a esta arquitectura como el soporte para el simulador *Player/Stage*.

Este proyecto ha servido para entender y experimentar por uno mismo, las fases en las que se divide un trabajo de esta importancia. Nos ha acercado hacia el mundo laboral y las implicaciones que conlleva esto, ya que en ocasiones nos hemos encontrado con problemas de mayor o menor importancia, y que hemos resuelto utilizando otras técnicas o mediante experimentos y pruebas afinando lo más posible el comportamiento.

Además del trabajo reflejado por esta memoria, me gustaría destacar el trabajo que se realizó en el periodo de formación previo al proyecto. En el periodo se realizó un proceso de aprendizaje de la plataforma a utilizar así como de la programación de distintos esquemas motores y perceptivos. En este sentido, otro mérito de este proyecto ha sido el esquema perceptivo *VisualDiff*, desarrollado en este periodo y que ha sido incorporado al repertorio de JDE. Éste es un esquema perceptivo, que realiza un filtro de movimiento. Procesa la imagen ofrecida por una cámara a través del servidor Oculo de tal manera que sólo se muestra aquellos píxels que se han movido. Este es un esquema que puede ser utilizado en aplicaciones de seguridad, o detección de movimiento.

6.2. Trabajos futuros

Puesto que éste es un proyecto de fin de carrera de una Ingeniería Técnica, no ha sido posible la realización de todas las mejores y optimizaciones que pasaron por la mente del autor de este proyecto, a fin de no alargarlo innecesariamente. En esta sección se detallan algunas mejoras posibles que en un futuro se podrían realizar sobre este proyecto.

- Este proyecto ha resuelto el problema de la navegación global usando únicamente simuladores. Esto ha sido así puesto que necesitamos saber en todo momento la posición del robot, y este problema no ha sido resuelto por el grupo de robótica todavía. El siguiente paso lógico es llevar el algoritmo de navegación desarrollado al robot real. Para ello ha de combinarse éste algoritmo con otro de localización, que estime continuamente la posición del robot.

- Se puede usar esta misma técnica del gradiente para realizar una navegación local basada en ella. El robot crearía mapas locales a partir de sus medidas sensoriales y propagaría un campo para moverse a través de este entorno cercano percibido. Esto evitaría el tener que conocer a priori el mapa de su escenario, ya que el propio robot se construiría un mapa de su entorno y sería capaz de navegar por él, evitando obstáculos de la misma manera que ahora.

Bibliografía

[ActivMedia, 2002] ActivMedia. Aria reference manual. *Technical Report version 1.1.10, ActiveMedia Robotics*, 2002.

[Benítez, 2004] Raúl Benítez. Sistema de localización basado en la detección de segmentos para robot móviles. *Proyecto fin de carrera, URJC*, 2004.

[Brian P. Gerkey, 2003] Andrew Howard Brian P. Gerkey, Richard T. Vaughan. The player/stage project: Tools for multi-robot and distributed sensor systems. *Proceedings of the international conference on Advanced Robotics.*, pages 317–323, 2003.

[Calvo, 2004] Roberto Calvo. Comportamiento sigue persona con visión direccional. *Proyecto fin de carrera, URJC*, 2004.

[Crespo, 2003] María Angeles Crespo. Localización probabilística en un robot con visión local. *Proyecto fin de carrera, Universidad Politécnica de Madrid*, 2003.

[GSyC, 2004] GSyC. Tema de navegación del temario de robotica. *Universidad Rey Juan Carlos-PFC*, 2004.

[Herencia, 2004] Ricardo Ortiz Herencia. Comportamiento sigue persona con visión. *Proyecto fin de carrera, URJC*, 2004.

[J. Borenstein, 1989] Y. Koren J. Borenstein. Real-time obstacle avoidance for fast mobile robots. *IEEE Journal of Robotics and Automation*, 1989.

[Konolige, 2000] Kurt Konolige. A gradient method for realtime robot control. *IROS*, 2000.

[Lobato, 2003] David Lobato. Evitación de obstáculos basada en ventana dinámica. *Proyecto fin de carrera, URJC*, 2003.

[López, 2005] Alejandro López. Navegación global utilizando grafo de visibilidad. *Proyecto fin de carrera, URJC*, 2005.

[Plaza, 2003] José María Cañas Plaza. Jerarquía dinámica de esquemas para la generación de comportamiento autónomo. *Tesis doctoral, Universidad Politécnica de Madrid*, 2003.

[Plaza, 2004] José María Cañas Plaza. Manual de programación de robots con jde. *URJC*, pages 1–36, 2004.

[Pérez, 2004] Lía García Pérez. Navegación autónoma de robots en agricultura: un modelo de agentes. *Tesis doctoral, Universidad Complutense de Madrid*, 2004.