INGENIERÍA INFORMÁTICA

Escuela Técnica Superior de Ingeniería Informática

Curso académico 2007-2008

Proyecto Fin de Carrera

Navegación y autolocalización de un robot guía de visitantes

Tutor: José María Cañas Plaza
Autor: Julio Manuel Vega Pérez

Dentro de un alma absolutamente libre de
pensamientos y emociones ni siquiera el
tigre encuentra espacio para clavar sus
fieras garras.

Si la misma brisa pasa sobre los
pinos en la montaña, sobre los
robles en el valle; ¿Por que dan un
sonido diferente?

Ningún pensamiento, ninguna reflexión,
perfecta vacuidad; sin embargo dentro
algo se mueve, siguiendo su propio curso.

El ojo la ve,
pero las manos no pueden apresarle
la luna en la corriente.

Nubes y brumas
son transformaciones medioambientales;
por encima de ellas, eternamente brilla el Sol y la Luna.

La victoria es para aquel,
que incluso antes del combate,
no tenga pensamientos sobre sí mismo,
y que more en la no-conciencia del Gran Origen.

Un sacerdote taoísta

A mi familia, sin la cual nada
de esto tendría sentido

Agradecimientos

Quisiera dedicar este proyecto a mucha gente, de los cuales seguro que se me escapa alguno... en tal caso que se den por aludidos y me perdonen.

Mis más sinceros agradecimientos al tutor de este proyecto, Jose María, por su tiempo, dedicación, esfuerzo, conocimientos facilitados y, sobretodo, por su paciencia conmigo. La primera persona en darme la bienvenida a esta universidad, y con la que espero continuar aprendiendo mucho tiempo.

Por otro lado, es de agradecer el buen ambiente que se respira dentro del Grupo de Robótica. En especial quiero nombrar a Victor, por su gran apoyo desde que lo conozco; a Antonio, por los grandes ratos que hemos pasado dentro y fuera del laboratorio; a David, porque es todo un personaje; a Redo y Javi, por lo mucho que me han enseñado; y en definitiva a todos aquellos con los que he compartido este tiempo. Gracias a todos, porque hacéis que esto sea realmente un *grupo*.

A mis amigos de Cáceres, con los que compartí los mejores momentos universitarios. Jose, como un hermano; Javi y Sevi, un dúo de *buena gente*; Mateo, qué buenos ratos; Yoli, comprensible como ella sóla; Pepe, Montse, Cristina, Alicia,... y más gente que seguro que se me está quedando en el tintero.

Y a la familia, ni que decir tiene que estas páginas van por y para ellos. Su apoyo incondicional desde siempre, especialmente desde que estoy en la distancia; la misma que me separa de mis amigos de *toda la vida*: Jona, Francisco, Moi, Lucas, Juanfran... en fin, aquéllos que, aunque no sepan muy bien de qué va todo esto, siempre están ahí.

Espero que os guste;
esto también es vuestro

Resumen

Los robots móviles de servicio constituyen una rama de investigación activa dentro de la robótica. Un robot móvil se caracteriza por realizar una serie de desplazamientos y por llevar a cabo una interacción con distintos elementos de su entorno de trabajo, de acuerdo a su programación. Elaboran la información que reciben a través de sus propios sistemas de sensores.

Un robot de servicio frecuente es el *robot guía*. El propósito primordial de estos robots es llevar de un sitio a otro a personas que desconocen tal lugar. En este proyecto hemos desarrollado los componentes software necesarios para un robot guía que opera en entornos de oficina: componentes de navegación y de autolocalización. Autolocalización es la habilidad del robot para estimar su posición en un mapa. Por navegación se entiende planificar un camino hacia el destino y evitar obstáculos.

La planificación de ruta se ha resuelto mediante un método que emplea campos de potenciales (*Gradient Path Planning* o *GPP*), asociando un coste de viaje a cada punto del espacio y navegando siguiendo el gradiente de ese campo. Con el fin de sortear obstáculos imprevistos hemos implementado varios algoritmos, entre ellos uno de fuerzas virtuales (*Virtual Forces Field* o *VFF*), repulsivas de los objetos y atractiva del destino, que nos concretarán una fuerza directora del movimiento del robot. Lo hemos mejorado añadiendo una *ventana de seguridad* que rodea al robot en todo momento. Por último, hemos abordado el problema de *localización* mediante estimación probabilística, utilizando las imágenes de una cámara que tiene el robot mirando al techo, empleando para ello un filtro de partículas clásico (condensación) y un mapa visual.

Hemos realizado múltiples experimentos sobre el robot real *Pioneer* para afinar los algoritmos y validarlos en un entorno de interiores como es el Departamental II de esta universidad, donde se dan situaciones usuales como el constante movimiento de personas.

Índice general

Índice de figuras

Capítulo 1
Introducción

Estos seres no dormían nunca, como no lo hace el corazón del hombre. Como no tenían un gran sistema muscular que debiera recuperarse de sus fatigas, la extinción periódica que es el sueño era desconocida para ellos. No parecen haber conocido lo que es el cansancio. En nuestra Tierra jamás pudieron moverse sin hacer grandes esfuerzos; sin embargo, estuvieron en movimiento hasta el último minuto. Cumplían veinticuatro horas de labor durante el día, como quizá lo hagan en la Tierra las hormigas.

H. G. Wells, *La guerra de los mundos*

En este primer capítulo se va a dar una visión global del contexto del presente proyecto. Comienza con una reseña histórica de los inicios de la robótica, repasando sus orígenes industriales y recorriendo algunos ejemplos de sus aplicaciones en la actualidad. A continuación hacemos un breve recorrido por las aplicaciones desarrolladas en robótica móvil autónoma; campo en el cual se encuadran los robots guía, cuyos prototipos más importantes también expondremos. Finalmente, describimos el contexto inmediato de otros trabajos desarrollados en esta misma línea por el Grupo de Robótica y que suponen los precursores directos de este proyecto.

1.1. Robótica

La palabra *robot* fue usada por primera vez en el año 1921, cuando el escritor checo Karel Capek (1890 - 1938) estrena en el teatro nacional de Praga su obra Rossum's Universal Robot (R.U.R.). Su origen es la palabra eslava *robota*, que se refiere al trabajo realizado de manera forzada.

Los robots tienen apariencia variada; no obstante, un robot siempre está compuesto básicamente por *sensores*, que captan información del entorno; los *actuadores*, cuyo propósito es interactuar con el entorno según la información percibida; y los *procesadores*, que hacen de intermediarios entre sendos componentes, recibiendo la información dada por los sensores y comandando convenientemente a los actuadores.

Muchas películas han mostrado a los robots como máquinas dañinas y amenazadoras. Sin embargo, películas más recientes, como la saga de *La guerra de las galaxias* desde 1977, retratan a robots como *C3PO* y *R2D2* como ayudantes del hombre. *Número 5* de *Cortocircuito* y *C3PO* realmente tienen apariencia humana. Estos robots que se fabrican con apariencia humana se llaman *humanoides*. En la actualidad la robótica ha avanzado en los modelos *humanoides*, tales como *Q-RIO* de Sony, *Asimo* de Honda o *Nao* de Aldebaran, consiguiendo comportamientos bípedos avanzados tales como andar y subir y bajar escaleras con dos piernas.

Figura 1.1: Ejemplos de robots bípedos: a) Robot Asimo de Honda; b) Q-RIO de Sony; y c) Nao de Aldebaran.

Más allá de las películas, en los últimos años se ha introducido la robótica en el hogar con *robots de servicio* donde las grandes marcas como iRobot, Honda, Sony, Fujitsu, Toyota son los principales creadores de tales prototipos. Un ejemplo de estos robots es la aspiradora *Roomba* (figura 1.2). Se trata de una aspiradora automática que barre y aspira suelos sin ningún tipo de supervisión. Para ello utiliza la navegación autónoma evitando los diferentes obstáculos que encuentra, recorriendo todo el terreno disponible.

Figura 1.2: a) Aspiradora Roomba; b) Pared virtual para limitar espacios grandes.

Los robots también han hecho su aparición en el sector de entretenimiento. Este es el caso de los robots *Lego Mindstorm* como kit de construcción y programación de robots con piezas Lego. También, los perritos *Aibo* de Sony, diseñados para su uso como mascotas en los hogares, han pasado de ser simples mascotas a jugadores de fútbol en la competición mundial de robots que juegan al fútbol denominada Robocup[1].

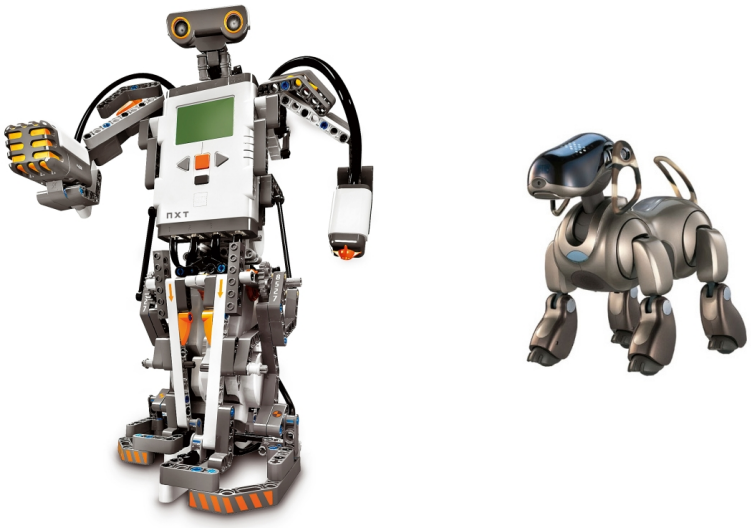


Figura 1.3: Robot Lego Mindstorm y Aibo de Sony.

Además de su irrupción en el hogar, los robots típicamente se usan para llevar a cabo tareas sucias, difíciles, peligrosas; en inglés, conocido como las tres *D* (*Dirty, Dull, Dangerous*). Por ejemplo, la industria automotriz ha tomado gran ventaja de esta nueva tecnología; en ella, los robots son programados para reemplazar el trabajo de los humanos en muchas tareas repetitivas. Así, todos los fabricantes de coches emplean brazos robotizados en mayor o menor medida; realizando labores de pintura, soldadura, ensamblaje de piezas, etc. Citaremos, por ejemplo, una de las grandes empresas de este sector como es *Toyota Motor Corporation*, ilustrada en la imagen 1.4.

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

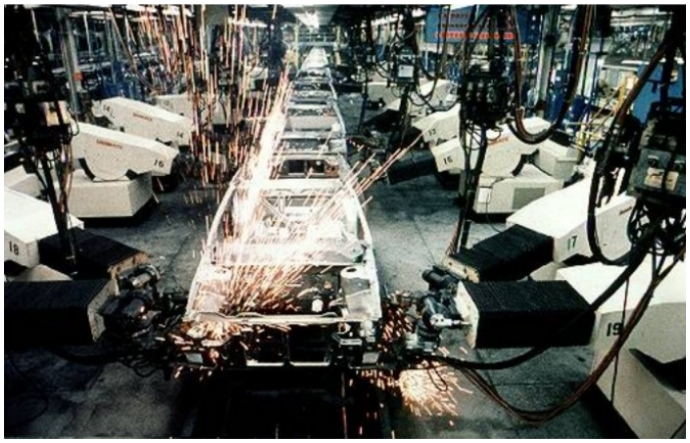


Figura 1.4: Cadena de montaje de coches de Toyota.

Otro ámbito de utilidad de los robots es la medicina. En la actualidad, se ha logrado un gran avance en los robots dedicados a la medicina, con dos compañías en particular, *Computer Motion* e *Intuitive Surgical*, que han recibido la aprobación regulatoria en América del Norte, Europa y Asia para que sus robots sean utilizados en procedimientos de cirugía invasiva mínima. Integrando la tecnología de la robótica con la habilidad y destreza del cirujano, el Sistema de Cirugía *Da Vinci* de *Intuitive Surgical* (ver figura 1.5) permite a los cirujanos intervenir y operar de una manera jamás experimentada con anterioridad. Muchos procedimientos que hoy en día se ejecutan con la técnica laparoscópica convencional pueden realizarse de manera más rápida y más fácil con el robot Da Vinci.

Figura 1.5: Robot cirujano Da Vinci en una operación.

La automatización de laboratorios y almacenes también es un área en crecimiento. Aquí los robots son utilizados para transportar muestras biológicas o químicas entre instrumentos tales como incubadoras, manejadores de líquidos y lectores.

Tanto el robot Da Vinci, como los que mencionamos a continuación son *robots teleoperados*; es decir, no son autónomos sino que son controlados remotamente por un operador humano. Cualquiera que sea su clase, los teleoperadores son generalmente

muy sofisticados y extremadamente útiles en entornos peligrosos tales como residuos químicos y desactivación de bombas. Por ejemplo, el robot que vemos en la figura 1.6 fue desarrollado tras el desastre de Chernobyl. Así, este robot tiene la principal característica de poder navegar por entornos peligrosos y difíciles de acceder por parte del hombre; normalmente cuando ocurre alguna catástrofe y los caminos *habituales* de acceso puedan quedar bloqueados.

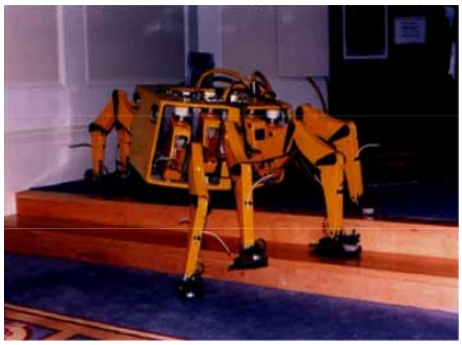


Figura 1.6: Robot Robug III.

Otro claro ejemplo de lugar peligroso para el hombre nos lo encontramos cuando hay que desactivar bombas. Para esta labor existen multitud de robots comerciales. El que mostramos en la figura 1.7-a es el *MR-5*, diseñado y comercializado por ESI (*Engineering Service Inc.*[2]). Es una plataforma controlada remotamente, con un brazo manipulador que puede alcanzar los 170 cm. de longitud y puede llegar a soportar hasta 20 kg. Pesa 250 kg. y alcanza los 2.5 km./h.

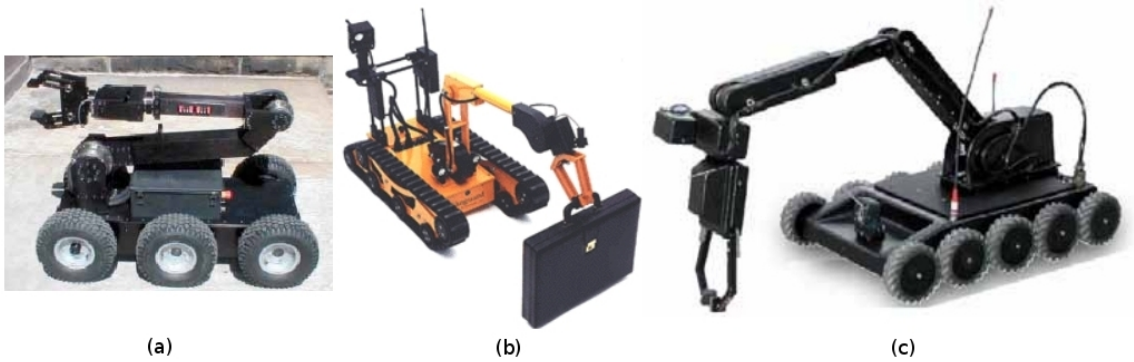


Figura 1.7: Robots desactivadores de bombas. a) Robot MR-5; b) Robot de RoboProbe; y c) Robot MURV-100.

Otro tipo de actividades en las que se centran los robots móviles es la *exploración* de sitios inalcanzables por el ser humano. Por ejemplo, los robots destinados a la exploración interplanetaria. Los robots exploradores han sido usados en misiones a

[2]http://www.esit.com

Marte y la Luna por las agencias soviética y americana para recabar datos de estos lugares. El objetivo es hacer aterrizar una nave que contiene un robot móvil con instrumental de medida que explora el lugar con mayor o menor grado de autonomía.

Un ejemplo clásico de ello serían los robots *Lunokhod 1 y 2*. Se trata de dos robots móviles exploradores enviados por la Unión Soviética en otoño de 1970 y enero de 1973. Lunokhod 1 posee 8 ruedas con 8 motores controlados independientemente, 2 antenas para comunicaciones, 4 cámaras de televisión y dispositivos especiales para la prospección del suelo lunar y realización de determinadas medidas.En la figura 1.8-a se puede ver una imagen de éste. El robot era teleoperado desde la URSS por 5 personas. El tiempo esperado de duración de la misión era de 3 meses y finalmente el robot estuvo funcionando durante 11, recorriendo más de 10 km. en total, enviando más de 20000 imágenes de televisión y realizando más de 500 tests del suelo lunar. Lunokhod 2 era una mejora de Lunokhod 1.

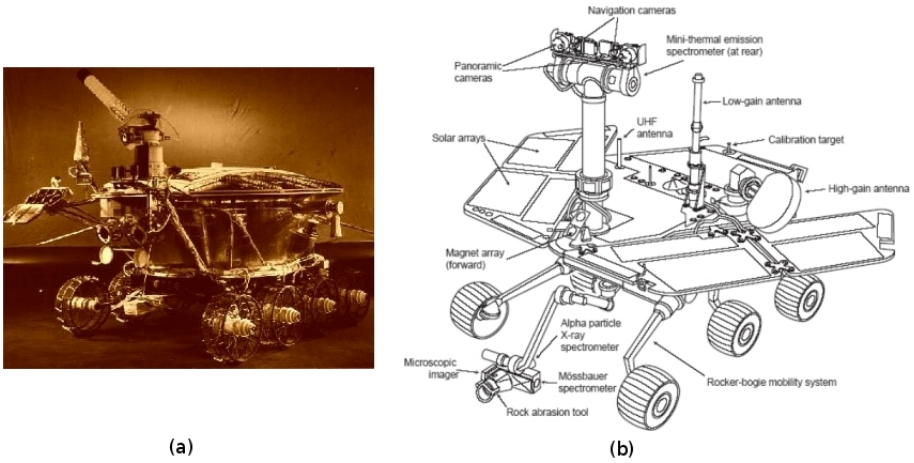


Figura 1.8: Robots exploradores interplanetarios. a) Lunokhod 1 y b) Esquema de Mars Exploration Rover.

Los robots gemelos *Spirit* y *Opportunity* llevan funcionando en Marte desde enero de 2004 y hasta hoy, como laboratorios geológicos móviles. Poseen diferentes herramientas para estudiar la superficie rocosa de Marte. Su único predecesor, el *Sojouner*, estuvo funcionando en Marte durante los meses de julio, agosto y septiembre de 1997.

1.2. Robots móviles autónomos

En la misma línea que los robots exploradores vistos anteriormente, pero en un contexto más próximo a este proyecto fin de carrera, están los robots móviles autónomos. No obstante, existen diferencias radicales entre éstos y los primeros robots vistos en la sección anterior. En primer lugar, hemos descrito los brazos *manipuladores*, mientras que ahora veremos robots *móviles*; y, en segundo lugar, también hemos

detallado robots *teleoperados*, lo cual es totalmente opuesto a los robots *autónomos* que veremos a continuación.

Los robots móviles autónomos ocupan un campo propio dentro de la robótica móvil. En ellos surgen problemas respecto a la navegación, y la autolocalización del robot en el entorno donde se mueve. Respecto al problema de navegación se distingue entre navegación local, que básicamente consiste en evitar obstáculos, siguiendo siempre un rumbo que vendrá dado por el otro mecanismo de navegación, esto es, la navegación global. Y la piedra angular de la navegación global es la autolocalización; ya que el robot necesita saber dónde está en todo momento.

Un buen ejemplo donde se dan robots móviles autónomos es la competición estadounidense *Grand Challengue*[3] (2004, 2005) y la última realizada, denominada *Urban Challengue*[4] (2007). Nació como reto por parte del *DARPA*[5] (Agencia de Investigación de Proyectos Avanzados de Defensa de E.E.U.U.); quien incluso en la última edición financió a algunos equipos con el propósito de poder licenciar alguna nueva tecnología que pueda interesarle. El objetivo en último término de este proyecto, es hacer un uso militar de esta tecnología robotizada; aunque también hay otros beneficios civiles, como el desarrollo de coches inteligentes capaces de evitar accidentes en carretera.

Figura 1.9: Competición de coches robotizados. (a) y (b) Grand Challengue, (c) Urban Challengue.

En estas competiciones los vehículos se dirigen exclusivamente de modo autónomo. Calculan la ruta previamente, con la información proporcionada a través de un mapa. Para la detección de obstáculos inesperados durante el trayecto, se suelen utilizar sistema de sensores láser similar al que empleamos en este proyecto (láser *sick*). Y en cuanto a la ubicación se suele usar *GPS* y *GIS* (localización geográfica); con esto se resuelve la localización. El principal problema no es saber donde se encuentra un vehículo, ya que hoy en día se pueden encontrar sistemas de *GPS* civiles capaces de obtener una precisión con un error menor de 50 cm. El auténtico problema es conseguir interpretar correctamente los datos del sistema de Información Geográfica (*GIS*) para

[3] http://www.darpagrandchallenge.com/
[4] http://www.urbanchallenge.com/
[5] www.darpa.mil/

no acabar perdido en la calle equivocada.

Respecto al comportamiento del vehículo, se diferencian tres aspectos: navegación, autonomía y lógica de control. Por navegación se entiende que una vez detectada la ruta correcta el *coche robot* debe seguirla y en caso de que se pierda en un momento dado, debe ser capaz de reubicarse y definir una nueva trayectoria para no saltarse ningún punto intermedio de control (*checkpoint*). Y finalmente, lo más importante es la lógica de control: el coche, el motor, el sistema *GPS*, el control del estado del vehículo,... todo depende del cerebro central.

La edición *Urban Challengue* surgió como un incremento de dificultad respecto a las ediciones de *Grand Challengue*. En las dos primeras ediciones era imprescindible navegar de forma autónoma durante unos 210 km sobre el desierto de California, ya que como mucho los únicos obstáculos que tendría que esquivar serían otros coches de la competición. La última edición propuso un reto más desafiante, ya que los coches debían sortear gran cantidad de obstáculos así como respetar las normas de circulación, al moverse por un núcleo urbano. Pues bien, tras estas ediciones aún no se sabe si el *DARPA Challenge* tendrá continuación, puesto que los organizadores consideran que la tecnología para vehículos robóticos ha alcanzado un nivel suficiente como para que sea desarrollada por compañías privadas u otras organizaciones con fines comerciales.

1.3. Robots guía

Dentro de los robots móviles autónomos nosotros nos centraremos en los *robots guía*, cuyo propósito primordial es llevar de un sitio a otro a personas que desconocen tal lugar; añadiendo el problema de localizar al robot por entornos interiores, lo cual a día de hoy es una ardua tarea candente de investigación. La navegación es una cuestión principal para estos robots. Por *navegación* se entiende la habilidad del robot para localizarse a sí mismo en un mapa; planificar un camino de navegación hacia el destino; y evitar obstáculos. Si no hay obstáculos imprevistos, al robot le basta con navegar basado en un mapa; en cualquier caso, la localización es necesaria para la navegación global. En muchos sistemas robóticos, sólo con la parte de navegación descrita el robot puede alcanzar su propósito. Sin embargo, en la aplicación que nosotros estamos tratando, hemos de considerar también el ambiente en el que el robot se moverá, con multitud de personas a las que ha de prestar su servicio.

En este campo de los robots guía hay varios trabajos realizados hasta la fecha. A continuación citaremos los más relevantes, que sirven de antecedentes para nuestro proyecto.

Urbano

Los grupos de investigación *UPM-DISAM, UPM-GTH y CACSA*[6] desarrollaron dentro del proyecto *URBANO* (CICYT 2002-2004) un robot guía para ferias y museos. Se pretendía incorporar en esta plataforma los comportamientos sociales necesarios para interactuar adecuadamente con los humanos en el desarrollo de las misiones asignadas. El Grupo de Tecnología del Habla de la UPM es el responsable del módulo de reconocimiento y síntesis de voz. UPM-DISAM desarrolla el resto de los componentes. CACSA supervisa el proyecto, asistiendo a todas las reuniones, fijando las especificaciones de la demostración y poniendo el Museo a disposición para llevar a cabo las demostraciones.

El sistema se compone de un cuerpo artificial (robot móvil, 1.10) con cierto nivel de inteligencia (autonomía), con el cual es posible interactuar a través de Internet y de manera presencial, pudiendo ser operado mediante lenguaje hablado. Los usuarios a los que va dirigido el proyecto son ciudadanos con condicionantes especiales necesitados de integración social, personas de negocios que por motivos de economía o tiempo se decantan por hacer una visita virtual, y empresas públicas y privadas, o ciudades españolas en su conjunto, que desean difundir su bagaje cultural, educativo o científico. En lo que se refiere a la interacción hablada, este proyecto desarrolla nuevas líneas tecnológicas no previstas anteriormente para la mejora de la interacción *hombre-robot* en el sistema de visita presencial o remota con ayuda automática.

Figura 1.10: Robot Urbano, realizando sus funciones de guía del museo.

Cicerobot

Es una especie de *R2D2*, algo más alto, con una licenciatura en Historia del arte y pinta de bibliotecario. Hace de guía para los visitantes del museo Agrigento Regional Archaeological Museum. Actualmente sus diseñadores (*Palermo University Robotics*[7]) están trabajando en otros modelos que actuarán de vigilante para *proteger* los tesoros arqueológicos de los ladrones...

[6] www.disam.upm.es/control
[7] http://www.difter.unipa.it

Figura 1.11: Cicerobot.

Enon, de Fujitsu

En Septiembre de 2007 se ponía en marcha este robot como guía del museo de Kyotaro Nishimura que se encuentra en la ciudad de Yugawara. En realidad, el pequeño robot nació en 2005 en los laboratorios de las firmas japonesas Fujitsu y Fujitsu Frontech[8]. Desde entonces y hasta su puesta en funcionamiento, estuvieron perfeccionando sus habilidades, aprendiendo nuevas tareas. Ya en el 2006 encontró trabajo en los grandes almacenes Jusco Park de la ciudad de Oita (en el suroeste de Japón). Enon se comportaba como un dependiente. Atendió a los clientes de viva voz, y además disponía de una pantalla táctil. Entre sus funciones estaba la de acompañar a los clientes para ayudarles a encontrar lo que buscaban. El robot también retiraba la basura.

Ahora, este robot de 1.3 metros de estatura y 50 kilos de peso ha encontrado un trabajo más estable en el museo del señor Nishimura. Enon recibe a los visitantes, los guía en un recorrido a través del museo y los despide a la salida, mostrándoles un vídeo en la pantalla de su pecho, en la que el propio Nishimura les da las gracias por haber venido. Circula a una velocidad máxima de 3 Km/h. gracias a un motor que mueve sus dos ruedas y, eso sí, no va dirigido por control remoto. Lleva unos sensores de detección de obstáculos, para moverse de forma autónoma sin ir tirando los obstáculos que se va encontrando en el museo.

[8]http://www.frontech.fujitsu.com/en/forjp/robot/servicerobot/

Figura 1.12: Robot Enon, de Fujitsu.

Enon también puede hacer sencillos gestos, como girar los hombros para indicar a los visitantes el camino a seguir. Incluso actualmente está aprendiendo a hablar en varios idiomas: japonés (también puede reconocerlo), inglés, chino y coreano. Pero sólo cosas muy básicas, como si fuese un niño de cuatro años con exquisitos modales. También puede transportar objetos (como mucho de 10 kilos de peso) y hacer de vigilante cibernético, gracias a las seis cámaras que lleva en su cabeza, cuyas imágenes pueden ser transmitidas por Wi-Fi.

SmartPal V

En la Exhibición Internacional de Robots de 2007 se presentó este robot. El SmartPal V de Yasukawa Electric Corporation[9] es un robot que mide 1.3 metros de altura y dispone de la mayoría de los sensores robóticos disponibles: está equipado con un subsistema de reconocimiento de voz, detección de objetos y sensores de proximidad. Se diferencia de los demás robots en que dispones de varias articulaciones adicionales y unidades lumbares.

[9] http://www.yaskawa.co.jp/en/

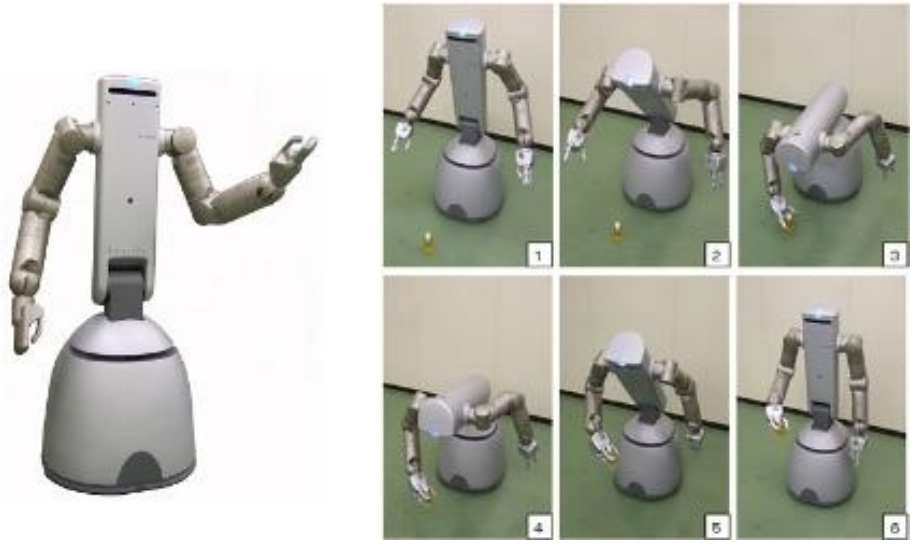


Figura 1.13: Robot SmartPal V. Recoge objetos del suelo, gracias a su articulación.

Es un robot que puede cumplir las funciones típicas de un guía turístico. También puede desempeñarse en museos y exposiciones brindando datos específicos a los visitantes. Esto es posible gracias a su sintetizador de voz y un software especializado en el reconocimiento de voz. Estas dos características le permiten mantener una conversación razonable sobre temas acotados. También puede distinguir a una persona por el timbre de su voz.

Puede desplazarse a una velocidad aproximada de 3.6 km/h, esquivando obstáculos gracias a sus cuatro cámaras CCD ubicadas en su cabeza. Estas cámaras proporcionan una gran habilidad en el reconocimiento de objetos, que puede recoger del piso con facilidad gracias a su articulación característica.

Minerva

Minerva[10] es un robot diseñado para educar y entretener a la gente en lugares públicos. El propósito del mismo es guiar a la gente a través de un museo, explicando qué es lo que están viendo durante el trayecto. Fue instalado en 1998 en el Museo Nacional de historia americana de Smithsonian. Durante sus dos semanas de operación, el robot interactuó con miles de personas, recorriendo más de 44 km. a velocidades superiores a 5.8 km/h.

[10]http://www.cs.cmu.edu/ Miverva

Figura 1.14: Robot Minerva.

Desarrollado de forma conjunta por *School of Computer Science* (Carnegie Mellon University, U.S.A.) y *Computer Science Department III* (Bonn University, Alemania). En un intento por superar al robot *Rhino*, previamente desarrollados también por ellos mismos (a mediados de 1997). Incluye importantes diferencias con éste: *Minerva* construye sus mapas, emplea un mosaico del techo para navegar, incluye una interfaz *emocional* (ver figura 1.16) para comunicar sus intenciones al usuario, etc.

Está controlado por un software implementado para la navegación autónoma de un robot, así como interactuar con personas. Resuelve los siguientes problemas:

- *Localización.* Minerva emplea dos tipos de mapas para orientarse a sí mismo: un mapa de ocupación y otro mapa de textura del techo del museo. Ambos son incorporados (manualmente teleoperando) al robot a través de sus sensores de información: láser, cámara y lecturas odométricas.

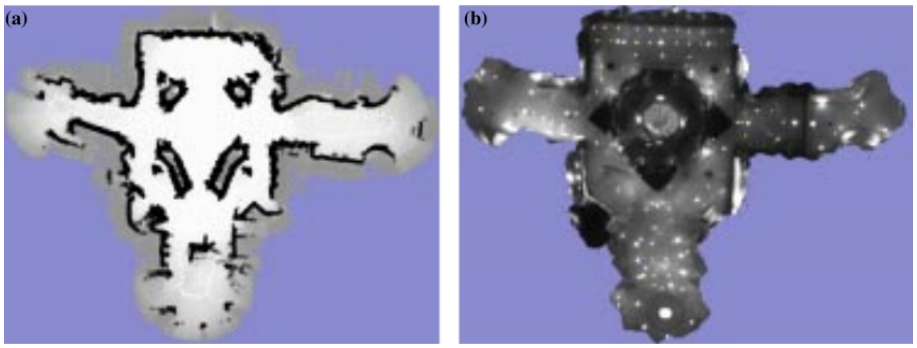


Figura 1.15: Mapa de ocupación y mosaico del techo del Museo Smithsonian.

- *Navegación en entornos dinámicos.* Los lugares públicos son normalmente ocupados por multitud de personas que no necesariamente cooperan con el sistema.

- *Navegación en entornos sin modificar.* Necesario para las operaciones del robot.

- *Interacción robot-humano.* Tal software está especialmente diseñado para interactuar con personas, o multitud de personas, que no han sido expuestas a robots previamente. De ahí que incluya una interfaz emocional, con el que poder comunicar los estados de ánimo.
- *Telepresencia virtual.* Una interfaz *web* permite a gente de todo el mundo ver lo que está viendo el robot en cualquier momento.

Figura 1.16: a) Interfaz emocional para comunicar sus estados de ánimo e intenciones; y b) Interfaz de control vía *web*.

1.4. Navegación y autolocalización de un robot guía de visitantes

En el grupo de Robótica de la Universidad Rey Juan Carlos se han realizado varios trabajos sobre temas de navegación y localización, que conforman el contexto cercano del presente proyecto.

Como proyectos relativos a comportamiento autónomo destacamos el seguimiento de una persona en los pasillos de un edificio ([Calvo, 2004]), la persecución de un congénere con el robot Pioneer ([Hidalgo, 2006]), o navegación local ([Lobato, 2003]). En cuanto a navegación global se han ido desarrollando trabajos que han utilizado diferentes técnicas. En ellos el robot necesita estar localizado en todo momento ([Isado, 2005], [López, 2005b]). En estos dos proyectos se asumía conocida la posición del robot en todo momento, es decir, la localización resuelta.

Por otro lado se han realizado trabajos sobre localización utilizando el sensor láser ([Kachach, 2005]) y visión local ([López, 2005a], [Crespo, 2003]). En ambos se han utilizado las técnicas de *filtro de partículas* y *mallas de probabilidad*. En estos proyectos se trabajaba la localización del robot asumiendo que se conoce el mapa de antemano,

el cual se almacenaba en forma de rejilla.

El pasado curso se realizó un proyecto ([Cortés, 2007]) donde se trataba de forma conjunta la navegación y la localización. Era el primer proyecto dentro del grupo que además de tratar la navegación y la localización de forma separada, se trabajaron de forma conjunta sobre simulador. Tal proyecto trató de avanzar en las técnicas de navegación, de construcción de mapas y de localización. Se acercaba a resolver el problema de navegación en el robot real.

Con este proyecto pretendemos, por tanto, dar un paso más allá y tratar la navegación y localización en el robot real *Pioneer*: navegación local, navegación global y localización. Se trata de un proceso de *integración*, desarrollo y optimización de distintas técnicas enfocado hacia una aplicación concreta: el *robot guía del Departamental II*. Mientras que otros proyectos previos se quedaron en experimentos sobre simulador; en el presente proyecto nos enfrentaremos a tratar con el robot real, construyendo por tanto algoritmos de navegación y localización lo suficientemente robustos, ágiles y rápidos como para interactuar con el mundo real en condiciones de seguridad.

En el siguiente capítulo (2) desarrollaremos los *objetivos* concretos que nos pretendemos cumplir, así como los *requisitos* subyacentes de éstos. A continuación detallaremos tanto la *plataforma* como las *herramientas* empleadas para poder satisfacer los objetivos marcados (3). Dado que es un proyecto en el que se engloban varias cualidades bien diferenciadas, las hemos separado en capítulos distintos en los que detallamos las labores de implementación de los diferentes *algoritmos* y técnicas en las que nos basamos, así como los *experimentos* llevados a cabo para depurar y optimizar los mismos. En el capítulo 4 describimos la navegación global, en el 5 nos centramos en la localización y en el 6 tratamos la navegación local. Por último haremos alusión a los resultados obtenidos sobre el robot *Pioneer* en el capítulo de *conclusiones* (7), así como una ligera perspectiva de futuro.

Capítulo 2
Objetivos

Puede ser muy interesante saber que el cuadrado de catorce es ciento noventa y seis, que la temperatura en este momento es de 28 grados centígrados, que la presión del aire acusa 750mm de mercurio, y que el peso atómico del sodio es 23, pero para esto, en realidad, no se necesita un robot.

Isaac Asimov, *Yo robot*

Una vez presentado el contexto en el cual se engloba nuestro proyecto, entraremos a detallar los objetivos concretos de este proyecto fin de carrera, así como los requisitos marcados.

El presente proyecto tiene como objetivo ensamblar un prototipo de robot móvil para ser usado como un sistema de guía del Departamental II de la Universidad Rey Juan Carlos, el cual tendrá la capacidad de detectar obstáculos cercanos que se encuentren en su trayectoria dentro de su ambiente de trabajo, y en caso de encontrar alguno decidirá sobre un camino alternativo.

2.1. Descripción del problema

Estamos interesados en diseñar un robot que navegue en tiempo real utilizando únicamente la información sensorial visual y a través de láser, así como odométrica, descartando explícitamente otro tipo de sensores como el sónar, GPS, etc.

El objetivo principal es conseguir dotar a un robot de las distintas habilidades necesarias para ejercer, por ejemplo, de robot guía. El robot ha de ser capaz de llegar al objetivo sin chocar contra ningún obstáculo, ya sea de tipo estático como una pared, o de tipo dinámico como una persona. Además, para navegar a destinos remotos, resulta imprescindible un mecanismo de localización que infiera la posición del robot en todo momento, ya que el sistema de sensores de odometría de un robot real resulta impreciso.

Hemos dividido el objetivo principal en varios aspectos o subojetivos, que se han ido desarrollando a lo largo del mismo. Estos subobjetivos son:

- *Navegación global.* Capacitar al robot para poder planificar rutas de forma óptima. De este modo el robot será capaz de, una vez conocido el mapa por donde navegará, pronosticar una trayectoria cuya distancia será la mínima para alcanzar el destino. Puesto que el mapa es conocido de antemano, la posición del robot en el mismo depende de una correcta localización (aspecto tratado en el tercer apartado).
  El usuario podrá interactuar con el robot de forma que el primero le indique el destino deseado y, a su vez, el robot sea capaz de conocer la posición actual.

- *Localización.* Algoritmo para evitar que el robot pueda desorientarse y/o perderse en cualquier momento, ya que éste ha de saber en qué posición concreta de su entorno está situado. Permitiéndole así navegar de forma autónoma durante largos periodos de tiempo.

- *Navegación local.* Conjunto de algoritmos que otorgan capacidad al robot para que pueda esquivar obstáculos que pudieran aparecer de modo imprevisto. El robot desconoce el mapa por el que se mueve, de ahí que su posición no sea tenida en cuenta y, por tanto, no requiera de un sistema de localización (como en el caso anterior).

La experimentación será un aspecto muy importante de todos los subobjetivos. Resulta imprescindible estudiar y analizar el comportamiento del robot ante diferentes situaciones reales. En un primer momento, para optimizar los algoritmos, se emplearán diferentes simuladores gráficos que nos reportarán mucha información del comportamiento de los mismos y evitarán daños en el robot. Posteriormente se implantará nuestro sistema en un entorno real, que será nuestro fin último. Concretamente, la plataforma de evaluación de la arquitectura será un robot móvil, como el robot *Pioneer* disponible en el Laboratorio de Robótica de esta universidad.

2.2. Requisitos

A continuación detallaremos los requisitos que condicionan la solución al robot guía que hemos desarrollado en este proyecto y que estarán guiados por los objetivos marcados en la sección 2.1.

Varios de los requisitos básicos para este proyecto son la utilización del lenguaje C para la programación de los algoritmos en la plataforma *JDE* (que será descrita en el capítulo 3), y sobre el sistema operativo Linux.

El requisito más importante y que nos marcará el desarrollo de los distintos algoritmos es que la solución ha de funcionar sobre el robot real *Pioneer*, con lo que se han de tener en cuenta muchas circunstancias y condiciones de adversidad, tales como errores de cálculo en las mediciones dadas por los sensores, ruido visual (imágenes percibidas incoherentes), ruido odométrico (posición estimada errónea del robot), etc.

Otro requisito importante es la *vivacidad* del robot. La solución final desarrollada ha de *responder* en tiempo real; el robot guía no puede ser lento en sus movimientos. De ahí que el robot deba responder con rápidez a las situaciones que se le planteen. Igualmente, los movimientos a parte de ser rápidos han de ser *fluidos*.

Y otro requisito también importante es la seguridad a la hora de navegar. No es concebible un robot que se choque; tanto por la seguridad del robot, como por la del entorno que lo rodea.

2.3. Metodología

Para llevar a cabo el proceso software de este proyecto nos hemos basado en un modelo de desarrollo software *incremental*. Dado que se trata de un proyecto que abarca un amplio enfoque estructural, decidimos dividir las distintas partes que conforman el mismo en sucesivos incrementos representativos de cada parte; de este modo, tanto el desarrollo (visto bajo un enfoque global), como las entregas semanales al tutor corresponden a incrementos cuya funcionalidad cumple con los objetivos marcados la semana previa.

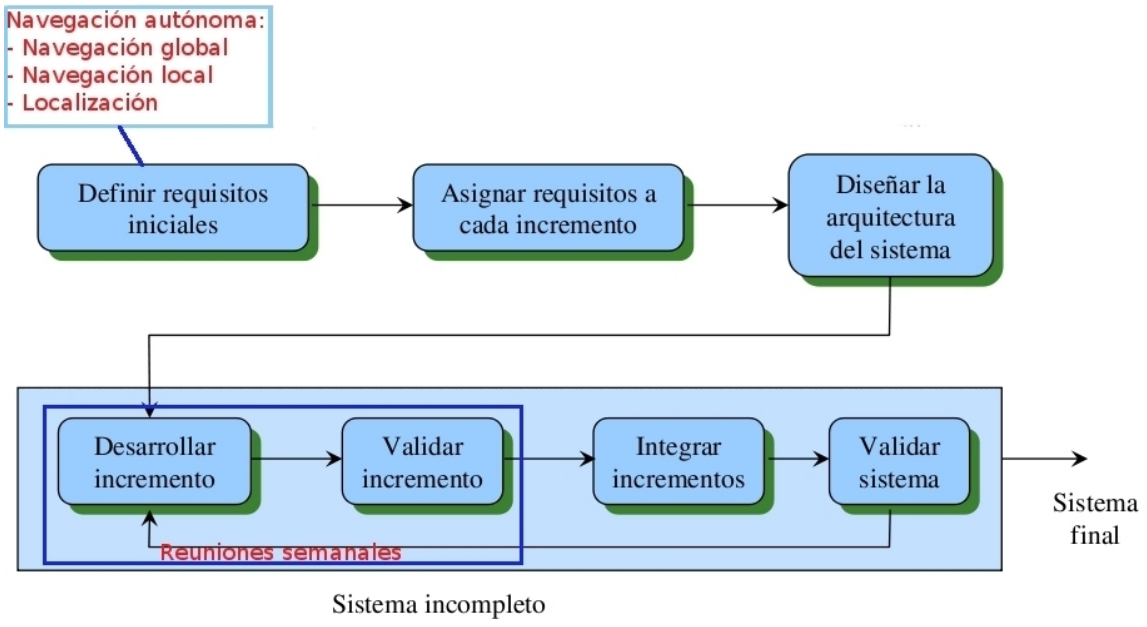


Figura 2.1: Modelo incremental de desarrollo software.

Obviamente, los requisitos de prioridad más alta se incluyen en los incrementos más tempranos. De forma que, cuando el desarrollo de un incremento comienza, sus requisitos son fijos; mientras que los requisitos de incrementos posteriores pueden ir siendo discutidos.

2.4. Plan de trabajo

Atendiendo a la metodología seguida, para el desarrollo del proyecto se han completado los siguientes *incrementos*:

- *Formación básica y familiarización con la infraestructura software.* En esta fase inicial se pretende adquirir conocimientos sobre el entorno Linux y sus herramientas, así como la familiarización con la plataforma software *jde.c* (ver capítulo 3) y estudio a fondo de la estructura de la misma. Pruebas con el lenguaje C y la biblioteca gráfica *XForms*. Utilizar el simulador *Player/Stage*, imprimir órdenes al robot *Pioneer* y aprender a manejar la información suministrada por éste.

- *Técnicas de navegación global en robots móviles.* Análisis, estudio y pruebas del algoritmo de navegación global, para que el robot pueda navegar de un sitio a otro planificando rutas de forma óptima. Estudio y simulación gráfica del algoritmo bajo la biblioteca *OpenGL*.

- *Técnicas de localización en interiores.* Análisis, estudio y pruebas de los algoritmos de localización, que permitan al robot conocer su posición en el entorno de trabajo. Adquisición y puesta en práctica de conocimientos en técnicas de *visión artificial* y otros aspectos relativos como filtros de color, geometría proyectiva, soportes de vídeo, calibración y funcionamiento de cámaras USB y Firewire.

- *Técnicas de navegación local.* Análisis, diseño y pruebas del algoritmo de navegación local para evitar obstáculos y explorar el entorno para la construcción del mapa de memoria temporal. Estudio y empleo de la biblioteca *GridsLib* para generar rejillas de información.

- *Experimentación.* En este último incremento se procedió a integrar las tres partes previamente desarrolladas. Asimismo se realizaron experimentos en conjunto de todos los algoritmos integrados primero sobre simulación y posteriormente en el robot real *Pioneer*, contrastando los resultados obtenidos.

Estos incrementos se han ido presentando al tutor del proyecto en las reuniones programadas semanalmente. Una vez finalizado todo el proceso de implementación del código, así como las respectivas pruebas y experimentos sobre el robot real, se pasa a escribir la memoria del proyecto, empleando para tal efecto unas 10 semanas de trabajo.

Igualmente es de destacar la gran cantidad de tiempo que han consumido las pruebas realizadas sobre el robot *Pioneer*, ya que en la mayoría de las ocasiones aparecían factores negativos que nos obligaban a modificar un número considerable de parámetros e instrucciones de los distintos algoritmos. Obviamente, el hecho de preparar convenientemente toda la infraestructura hardware subyacente a utilizar el robot real (cámaras, sensores láser, alineación de ruedas, modificación del entorno con balizas, etc.) es una de las tareas más laboriosas de este proyecto.

Capítulo 3
Plataforma de desarrollo

La fuerza del motor tiene que ser siempre proporcionada al peso del móvil y a la resistencia del medio en que se mueve el paso. Pero uno no puede deducir la ley de esta acción, a no ser que primero descubra la cantidad de condensación del aire cuando es golpeado por cualquier objeto movible. Esta condensación será más o menos densa, según que sea mayor o menor la velocidad del móvil que presiona sobre él. Esto aparece en el vuelo de las aves, ya que el sonido que hacen con las alas al batir el aire es más penetrante o menos, según que el movimiento de las alas sea más lento o más rápido.

Leonardo da Vinci, *Cuaderno de notas*

En este capítulo vamos a explicar la plataforma hardware (robot, láser y cámara) y software (*stage* y *jdec*) sobre la que ha sido desarrollado este proyecto, cuya implementación comentaremos en el siguiente capítulo.

Asimismo, comentaremos brevemente algunas bibliotecas de apoyo y herramientas útiles que hemos usado: *Gridslib*, para el tratamiento de rejillas de ocupación; *Fuzzylib*, que nos proporciona funcionalidad para el uso de controladores borrosos; *Progeo*, para relacionar el mundo de las imágenes (2D) con el mundo real (3D); *XForms*, para crear interfaces gráficas; y *OpenGL*, para visualizar de forma gráfica los algoritmos implementados.

3.1. Plataforma hardware

A continuación exponemos las características del robot real empleado como plataforma hardware de este proyecto; adaptado con diferentes componentes, como el sensor láser o la cámara. Sirviendo así de prototipo para un futuro robot guía.

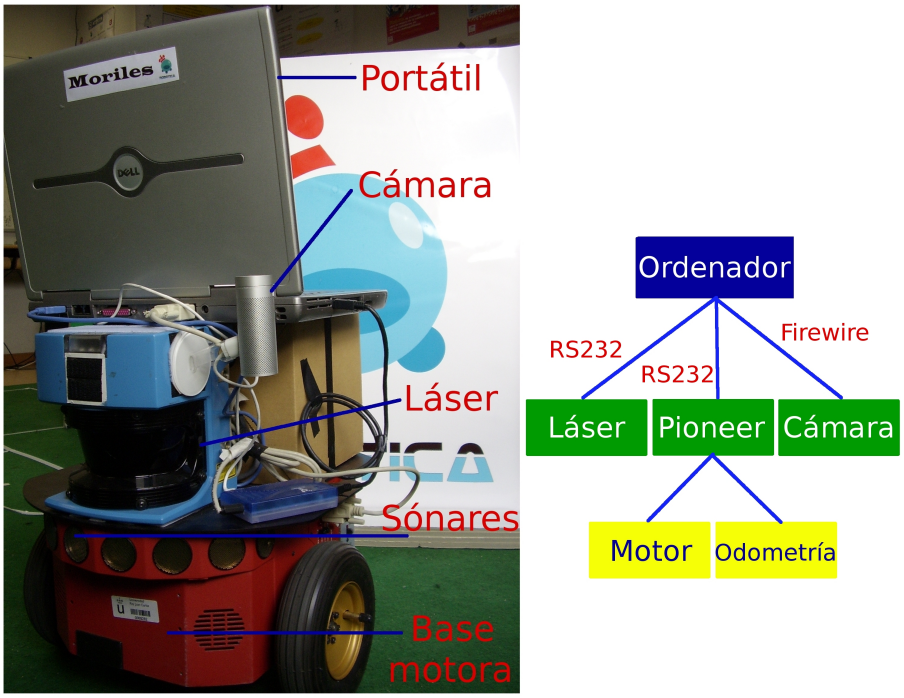


Figura 3.1: Robot Pioneer 2-DX sobre el que se centra el proyecto.

3.1.1. Robot Pioneer

El robot empleado es el *Pioneer 2-DX*, fabricado por ActivMedia Robotics (ver figura 3.1). Se trata de una versátil plataforma móvil autónoma, con dos ruedas motoras laterales y una tercera multidireccional (rueda loca), colocada en la parte trasera para lograr la estabilidad del conjunto. Las ruedas motoras están dotadas de odómetros para la medición de los desplazamientos y giros efectuados por el robot. La plataforma dispone también de un conjunto de 8 sensores de ultrasonidos (cubriendo un rango frontal de 180º) para la detección de obstáculos cercanos (a distancias entre 10 y 500 cm), y que funcionan a una frecuencia de 3 Hz cada uno. El conjunto está gobernado por un microcontrolador Siemens 88C166 a 20 MHz, con 64 KB de memoria, organizados en 32 KB de flash-ROM para almacenar el microkernel del sistema operativo que controla todos los sensores y actuadores, y otros 32 KB de RAM dinámica para los datos de operación.

La carrocería tiene 33 cm de ancho por 44 de largo y 22 de altura, sin incluir el ordenador portátil que lleva a bordo, así como el sensor láser (que comentaremos en el siguiente apartado). A pesar de lo reducido de las dimensiones comentadas, que dotan al Pioneer de una gran maniobrabilidad incluso en espacios pequeños, la situación ligeramente descentrada de las ruedas permite un diámetro de giro mínimo sobre sí mismo de 52 cm. En cuanto a las características energéticas, estos modelos operan

con tres baterías recargables de 12V que le dan una autonomía de funcionamiento de aproximadamente una hora, dependiendo de la cantidad de dispositivos conectados y del movimiento realizado. El peso total del conjunto es de 9 Kg, aunque puede soportar hasta 20 Kg de carga útil adicional.

La comunicación local entre el PC y el microcontrolador se realiza a través de un puerto serie *RS-232*. El ordenador portátil está 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*.

En nuestro proyecto necesitamos, además, la odometría o información de posición *(x,y,theta)*. Los odómetros cuentan las vueltas que dan las ruedas del robot y trabajan con una resolución en *mm/sg*; con una precisión de 9850 marcas por revolución en cada rueda. Para sacar información de posición a partir de las vueltas contadas es necesario realizar una conversión.

Una descripcion más detallada de las características del *Pioneer 2* y sus distintos componentes puede encontrarse en el manual de descripción del hardware del robot ([ActivMedia, 2002]).

3.1.2. Sensor láser

Para la navegación, hemos incorporado al robot un sensor láser (figura 3.2), de la compañía *Sick*, que mide las distancia a los obstáculos. El modelo *LMS-200* proporciona 10 barridos por segundo, con una precisión de 1 grado y 2cm. El láser ofrece perfiles del entorno mucho más nítidos y fiables que los sónares. Su principal inconveniente es el precio, que lo hace prohibitivo para robots comerciales. Se alimenta con una tensión continua de 24 Voltios, que se extraen con una placa elevadora de tensión desde los 12 Voltios internos del robot. El sensor entrega sus lecturas a través de un puerto serie convencional siguiendo el protocolo del fabricante. En general este sensor es muy fiable y preciso, aunque no detecta bien los cristales transparentes.

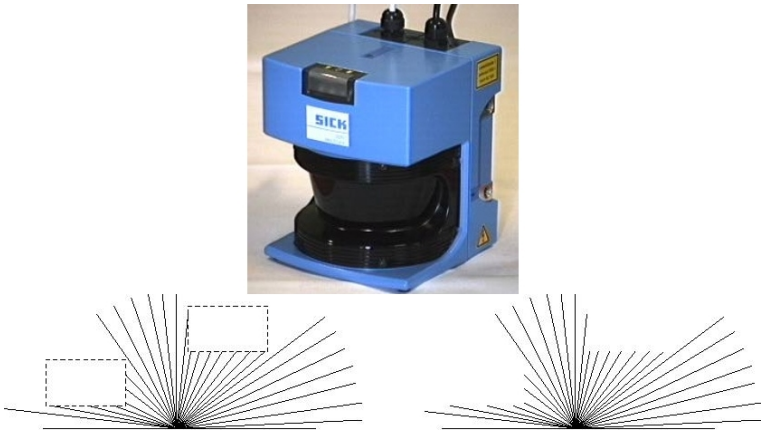


Figura 3.2: Láser *sick* y ejemplo de barrido.

3.1.3. Cámara

Para las tareas de localización, le hemos añadido una cámara *i-Sight* (de *Apple*, figura 3.3) en color para la captura de imágenes. Está conectada al portátil por el puerto *firewire*. Esta cámara presenta una resolución VGA de 320x240 píxeles, pudiendo trabajar a una frecuencia de 30Hz (fps). Otras características que ofrece son el autoiris y el autoenfoque.

Figura 3.3: Cámara *i-Sight* de Apple.

3.2. Stage

Este simulador tiene detrás una comunidad muy activa de desarrolladores y ha extendido mucho su uso en los últimos años, convirtiéndose en un estándar *de facto*. *Stage*[1] simula muchos modelos de robots, entre ellos los de base *Pioneer*, y muchos

[1] *http://playerstage.sourceforge.net*

dispositivos sensoriales y de actuación. En particular, simula los motores de la base, la odometría, los sónares y láser de nuestro robot de referencia. No simula las cámaras ni el cuello mecánico (no empleado en nuestro proyecto); aunque ambos aspectos sí son simulados por *Gazebo*. *Stage* está orientado a mundos bidimensionales, donde hay objetos estáticos, otros robots móviles o incluso cosas que se pueden empujar. La capacidad de simular varios robots en el mismo mundo permite poner obstáculos dinámicos y probar cómo el robot se comporta frente a ellos. En la figura vemos el Departamental II de esta universidad, que hemos modelado para simular el entorno por donde se moverá un futuro prototipo de robot guía.

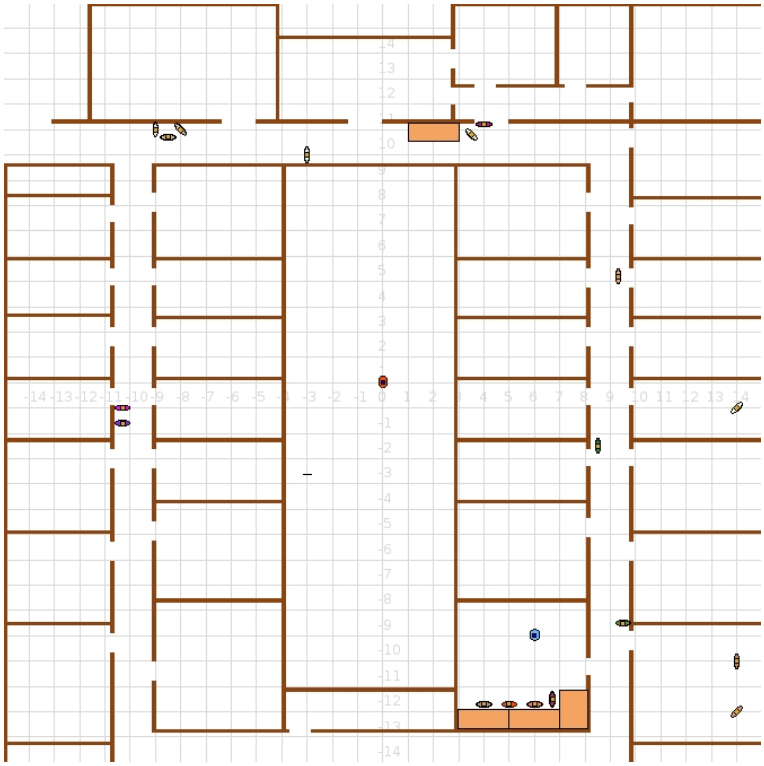


Figura 3.4: Departamental II de la *URJC*, modelado con *Stage*.

3.3. La plataforma sofwtare JDE

Jde[2] (Jerarquía Dinámica de Esquemas) es una plataforma software desarrollada íntegramente en la URJC ([Cañas Plaza *et al.*, 2007]) para facilitar la programación de robots y de aplicaciones relacionadas con la visión artificial, así como de domótica. Su origen se remonta al año 1997 como fruto de una tesis doctoral ([Cañas Plaza, 2003]). Desde entonces ha ido creciendo año tras año, con nuevas funcionalidades. Este proyecto se ha desarrollado sobre la versión 4.2.1 de *jdec*.

[2]http://jde.gsyc.es

Esta plataforma ofrece, en primer lugar, el acceso a los sensores del robot en forma de variables que las aplicaciones leen (*variables perceptivas*), y el acceso a los actuadores como variables que las aplicaciones escriben (*variables de actuación*). Para ello, la plataforma incluye drivers como por ejemplo: *pioneer*, para tratar con el robot *Pioneer* (ya sea real o simulado), así como con el láser *sick* mencionado previamente; *pantilt*, para el cuello mecánico que también se puede acoplar al robot; *firewire*, para tratar las imágenes procedentes de una cámara; o *imagefile*, para tratar imágenes procedentes de fichero.

En nuestro caso, hemos empleado el driver *pioneer* para tratar con el robot en simulador y con el robot real; *firewire*, para tratar con las imágenes que ayudan a localizar al robot en todo momento; e *imagefile* para almacenar y tratar con la *imagen-mosaico* del techo, también imprescindible en el mecanismo de localización.

En segundo lugar, proporciona un modelo basado en componentes, llamados *esquemas*, para construir las aplicaciones robóticas. La ejecución simultánea de varios esquemas dan lugar a un comportamiento. Existen esquemas de distintos tipos: *(i) perceptivos*, que se encargan de producir y almacenar información sensorial o elaborada por otros esquemas; y *(ii) de actuación*, que son los que toman decisiones sobre motores o la activación de esquemas de niveles inferiores a partir de la información que generan los esquemas perceptivos. Los esquemas pueden organizarse en niveles estableciendo una jerarquía entre esquemas padre y esquemas hijo siendo el padre el que pueda activar y desactivar a los esquemas hijo.

En este proyecto se han diseñado y programado dos esquemas de actuación (*navegacionLocal* y *navegacionGlobal*) y uno de percepción (*localizacion*), correspondientes a los tres pilares sobre los que se fundamente este proyecto; siendo el padre de todos el esquema *guiaMuseogui*.

En tercer lugar, esta plataforma ofrece soporte para el simulador *Stage*, para el robot real y la posibilidad de conectarse remotamente a ellos a través de servidores de red, manteniendo el acceso a través de variables:

- *Robot real.* Las variables representan de forma directa a los sensores y actuadores del robot Pioneer real.
- *Simulador.* Las variables representan de forma directa a los sensores y actuadores del robot Pioneer simulado.
- *Servidores.* La plataforma *jdec* también incluye los drivers *networkclient* y *networkserver*. De este modo, lanzando la aplicación *jdec* con *networkserver* se proporciona acceso remoto a los sensores y actuadores, ofreciendo así funcionalidad a los clientes (lanzando *jdec* con *networkclient*) a través de una *API* de mensajes.

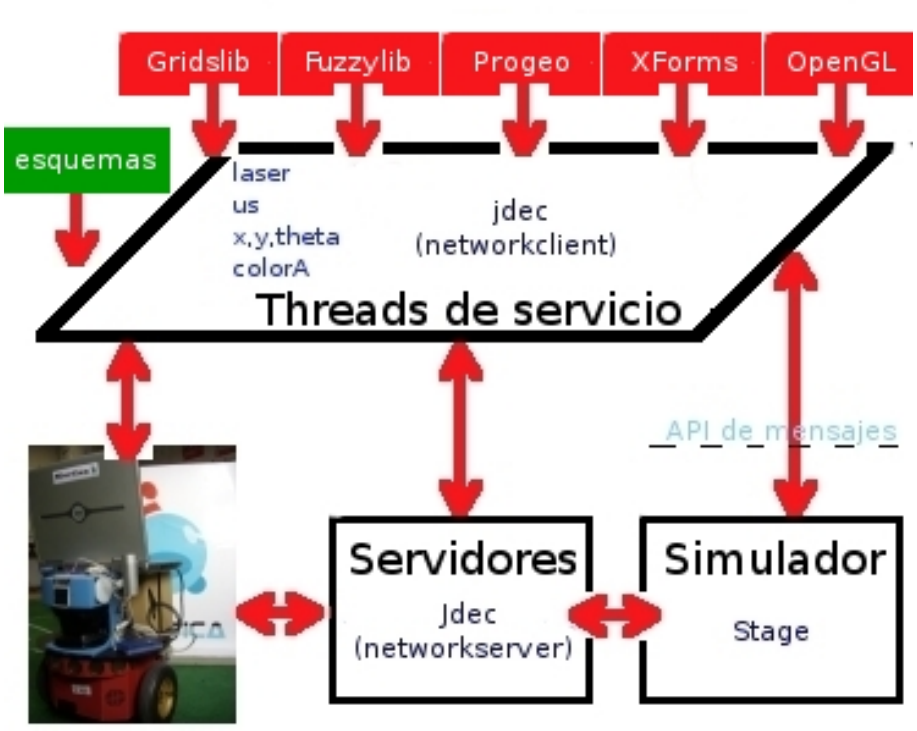


Figura 3.5: Plataforma de desarrollo del proyecto.

Por último, la plataforma también resuelve las necesidades de interfaz gráfica de las aplicaciones, típicamente empleadas para depuración. Buen ejemplo de ello son los drivers *graphics_gtk* y *graphics_xforms*, que nos ofrecen soporte para gestión de ventanas diseñas bajo *GTK* y bajo *XForms* respectivamente. De éstos, nosotros hemos empleado *graphics_xforms*.

3.4. Bibliotecas auxiliares

Junto con la plataforma *Jde* y el simulador *Stage*, en este proyecto se han utilizado también una serie de bibliotecas adicionales que detallamos a continuación. Éstas son la bibliotecas *Gridslib*, *Fuzzylib*, *Progeo*, *XForms* y *OpenGL*.

3.4.1. Gridslib

La librería *Gridslib* nos permite crear y manejar rejillas de ocupación. Esta biblioteca implementa diversas técnicas de construcción de mapas como la regla de Bayes, la regla Dempster-Shafer, rejillas histográmicas o rejillas borrosas. Esta biblioteca genera una rejilla cuadrada con celdas regulares de cierto tamaño. Estos tamaños son especificados en un archivo de configuración que la biblioteca se encarga de leer.

En nuestro proyecto esta biblioteca nos va servir para representar el mapa local de ocupación del escenario. Cada celda de la rejilla almacenará un valor que representará un punto del espacio. Estos valores pueden representar espacios vacíos, paredes, obstáculos. Además podemos asociarle un cierto valor temporal, lo que nos permitirá capacitar al robot de memoria *olvidadiza*, para no acumular ruido excesivo.

3.4.2. Fuzzylib

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 nublado*» o «*poca calor*» 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 la 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.

3.4.3. Progeo

La biblioteca *Progeo* es utilizada para realizar cálculos de geometría proyectiva. Estos cálculos permiten procesar la información de puntos geométricos en el espacio 3D para obtener sus proyecciones en un plano imagen determinado. Mediante estas proyecciones, los puntos 3D pueden ser visualizados en un monitor, empleando para ello algún gestor gráfico. De igual forma, los cálculos pueden ser realizados a la inversa, de manera que a partir de varias proyecciones de un punto, podamos obtener las coordenadas 3D del mismo.

Nosotros hemos utilizado únicamente el mecanismo de *retroproyección*, que nos permite obtener la recta de proyección que une el centro óptico de una cámara (la del mundo virtual, en nuestro caso) con el punto 3D que representa un punto 2D de su plano imagen.

Así, hemos podido establecer un sistema muy intuitivo de cara al usuario para interactuar con el mundo simulado en 3D con *OpenGL*, y obtener por ejemplo la imagen visual teórica percibida por el robot en esa posición; es decir, qué vería el robot si estuviera en tal posición. Lo cual nos ha resultado muy útil para depurar el algoritmo de localización.

3.4.4. XForms

La biblioteca *XForms*, basada en la librería *Xlib*, es utilizada por el esquema de servicio *guiaMuseogui* utilizado en este proyecto. Nos permite crear una interfaz gráfica con la que visualizar y depurar los resultados que vamos obteniendo con los distintos algoritmos. Esta biblioteca proporciona una herramienta denominada *fdesign* con la que poder crear objetos gráficos en sistemas de ventanas *X Window* como botones, diales, barras de desplazamiento, menús, canvas para visualización de *OpenGL* (que a continuación detallaremos), etc. Tal herramienta es soportada por *jde* a través del componente *graphics_xforms*.

3.4.5. OpenGL

Por último, la librería gráfica *OpenGL* nos va a permitir visualizar de forma elegante y eficiente escenas 3D o las estructuras internas de los algoritmos, por lo que nos será muy útil como método de depuración. Además, la carga computacional que teníamos con la librería *progeo* (que también se puede usar para renderizar escenas 3D) ahora se trasladará al procesador gráfico (*GPU*), ya que esta biblioteca trabaja directamente con el procesador gráfico y no con la *CPU*, como lo hace *progeo*.

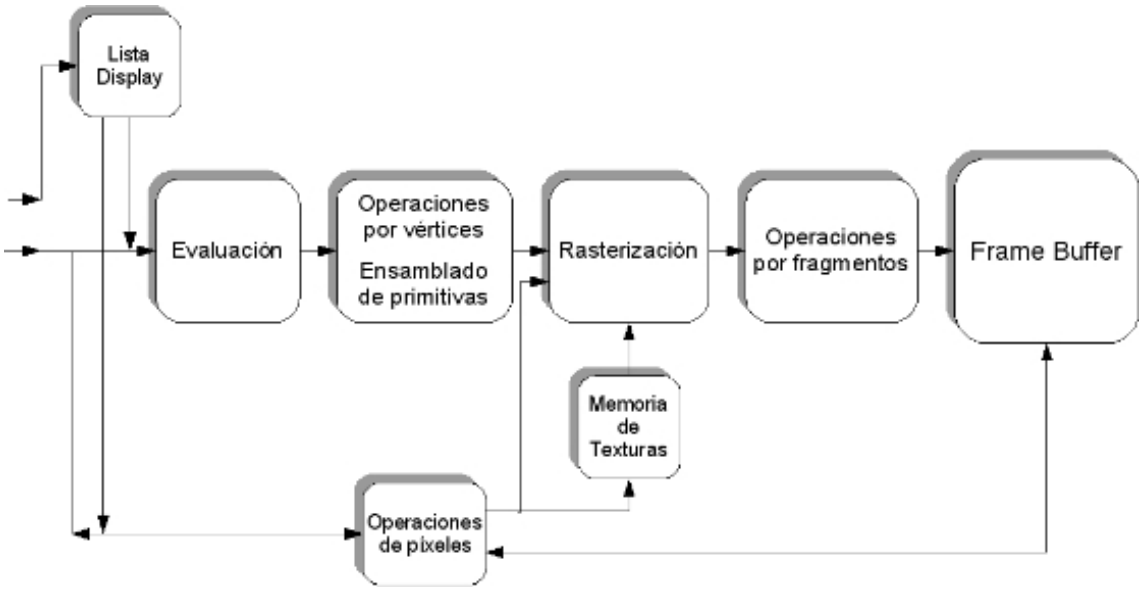


Figura 3.6: Pipeline de *OpenGL* y Lab. de Robótica, modelado con *OpenGL*.

Concretamente, hemos empleado esta librería para modelar entornos virtuales, como puede ser el Laboratorio de Robótica (figura 3.6), al que incluso le añadimos texturas a las paredes, suelo, ventanas, puertas, el techo compuesto como mosaico de varias fotografías reales, etc... Este último detalle nos ha sido de gran utilidad sobretodo para el algoritmo de localización (5), donde la piedra angular del mismo recae sobre las balizas incluidas en el techo real, detectadas visualmente por el robot.

También hemos modelado todo el departamental, pero ya sin tanto detalle; con motivo del algoritmo de navegación global (4).

Capítulo 4
Navegación global

Los viajeros tenían curiosidad de examinar la Luna durante la travesía, y para facilitar el reconocimiento de su nuevo mundo, iban provistos de un excelente mapa de Beer y Moedler, Mapa selenographica, publicado en cuatro hojas, que pasa, con razón, por una verdadera obra maestra de observación y paciencia. En dicho mapa se reproducen con escrupulosa exactitud los más insignificantes pormenores de la porción del astro que mira a la Tierra; montañas, valles, circos, cráteres, picos, ranuras, se ven en él con sus dimensiones exactas, con su fiel orientación, y hasta con su denominación propia, desde los montes Doerfel y Leibniz, cuya alta cima descuella en la parte oriental del disco, hasta el mar del Frío, que se extiende por las regiones circumpolares del Norte.

Julio Verne, *De la tierra a la luna*

Como ya hablamos en la introducción, un problema muy común al que se enfrenta un sistema móvil con un cierto grado de autonomía es el de navegar por un entorno evitando sus obstáculos hasta alcanzar una meta. Asimismo, vimos en el capítulo 2 que uno de los objetivos de este proyecto es dotar al robot de un algoritmo de *navegación global*, ya que de este modo el robot es capaz de planificar rutas de forma óptima.

La navegación global satisface la necesidad de planificar un trayecto. La *planificación* se define como la búsqueda de una ruta libre de obstáculos desde una posición inicial hasta otra final a través del entorno de trabajo del robot móvil. El destino puede ser más o menos lejano, pudiendo estar en una posición fuera del rango sensorial inmediato. Se planifica sobre un mapa topológico o métrico, mediante el uso de la información que se posee del entorno actualmente, la descripción de la tarea de navegación y algún tipo de metodología estratégica. Así, el planificador se define por el modelo del entorno y el algoritmo de búsqueda utilizado.

La segunda fase sería ejecutar o seguir la ruta planificada. El hecho de haber deliberado una ruta antes de que el robot empiece a moverse, hace que este método de navegación sea insensible a obstáculos que no estén en tal mapa y que pueden aparecer en cualquier momento. Por otro lado, para que el robot mantenga el rumbo adecuado debe saber dónde está en todo momento.

4.1. Técnicas de planificación

Los algoritmos de planificación espacial, 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 mapa a fin de calcular la mejor trayectoria posible. Este mapa puede venir *a priori*, porque se le haya proporcionado, o ser construido sobre la marcha por el propio robot a través de sus sensores.

Existen muchas técnicas de navegación global. En esta sección mostramos dos de las más representativas para ilustrar otras técnicas de navegación global distintas a la que hemos elegido. Una de ellas es la del *grafo de visibilidad* ([López, 2005b]), que utiliza los caminos que se forman al unir los vértices de los obstáculos con la condición de que no atraviesen ninguno de ellos a su paso. Otra técnica interesante son los *diagramas de Voronoi* ([Moreno *et al.*, 2004]), que utilizan los espacios que surgen al dejar los obstáculos a ambos lados a una misma distancia.

4.1.1. Grafos de visibilidad

Los grafos de visibilidad (Nilsson, 1969) 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 3D. 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 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.

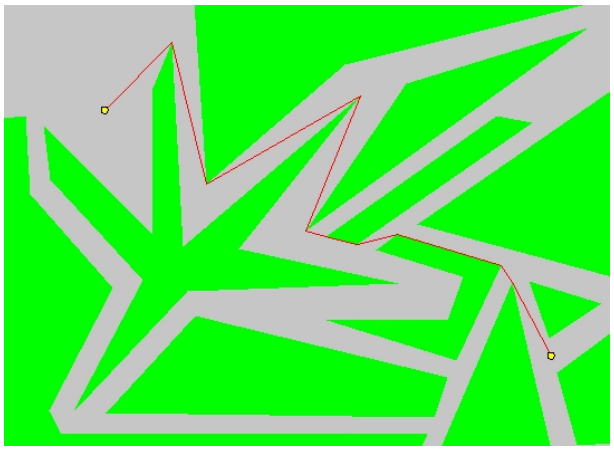


Figura 4.1: Grafo de visibilidad con 11 polígonos.

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

de camino dentro del grafo. Para construir el grafo, hay que aplicar un engorde a los polígonos para considerar el robot como un punto. La búsqueda consiste en ir asignando un coste a cada arco del grafo, por ejemplo su longitud, y se acabará encontrando una ruta desde el origen al destino siguiendo los arcos del mismo, que en este caso sería el camino de distancia mínima. El método de búsqueda podría seguir el *Algoritmo de Dijkstra*. 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, ésta se define por un conjunto ordenado de nodos del grafo. Este algoritmo fue implementado en el grupo de robótica ([López, 2005b]) y puesto en práctica en simulador.

4.1.2. Diagramas de Voronoi

El Diagrama de Voronoi ([Moreno *et al.*, 2004]) es definido como un conjunto de puntos que están equidistantes de dos o más objetos característicos. El conjunto de puntos de entrada los denotamos como $s_1, s_2, ..., s_n$. Para cada uno, s_i, definimos una función de distancia $D_i(x) = Dist(s_i, x)$. Así, la región de Voronoi de s_i es el conjunto:

$$V_i(x) = \{x | D_i(x) <= D_j(x)\} \tag{4.1}$$

El Diagrama de Voronoi particiona el espacio en tales regiones. Las intersecciones de esos caminos conforman los nodos del grafo resultante. Al igual que en la anterior técnica, sobre ese grafo puede utilizarse un algoritmo de búsqueda. La diferencia radica en que este método maximiza la seguridad al maximizar la distancia a obstáculos.

Por otro lado, hay que emplear algún mecanismo que en la fase inicial del movimiento del robot, permita a éste situarse sobre el grafo partiendo de su posición actual; ya que inicialmente no estará situado sobre ningún arco del grafo. Igualmente, hay que repetir el proceso para la salida del robot del grafo hacia su destino; que normalmente tampoco estará situado sobre el grafo.

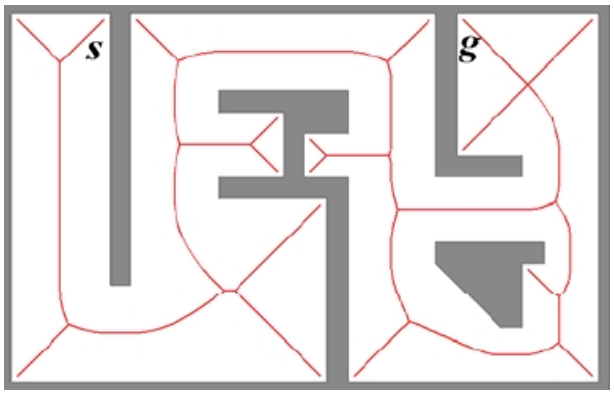


Figura 4.2: Generalización del Grafo de Voronoi.

4.2. Navegación global con *GPP*

Finalmente, para resolver el problema de la navegación global, hemos optado por usar el método del gradiente, 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.

4.2.1. Fundamentos teóricos

Para llevar a cabo el algoritmo de *Gradient Path Planning* (*GPP* a partir de ahora) nos basamos en el desarrollado por Kurt Konolige ([Konolige, 2000]), así como en previos trabajos sobre simulador que se han realizado en esta universidad años atrás ([Isado, 2005]). Tal método surge ante la necesidad de solucionar carencias de otros métodos. Esta técnica se basa en un campo virtual que representa, en cualquier punto del espacio, la distancia mínima hacia el destino (considerando que hay que sortear a los obstáculos); y se navega por su gradiente.

GPP calcula el camino óptimo hacia nuestro destino. El concepto de *óptimo* nace del hecho de ir asignado costes al camino según la distancia hacia el objetivo, y la cercanía a los obstáculos. En nuestro caso con estos dos criterios es suficiente, pero podríamos tener en cuenta otros como por ejemplo el tipo de suelo, si éste es inclinado, si es zona desconocida, o simplemente si el camino a través de tal camino es menos propicio para la correcta navegación del robot; en tales casos el coste aumentaría proporcionalmente...

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 revés, 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. Esto 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, ya que como tendremos un algoritmo de navegación local, habrá momentos en que éste saque al robot de la ruta óptima, por ejemplo al aparecer un obstáculo imprevisto. Esta técnica global, sin embargo, nos asegura que llegaremos al objetivo; el robot consulta el valor del campo, y sigue el gradiente desde ese punto que lo lleva al objetivo.

4.2.2. Esquema de navegación global

Este algoritmo se ha diseñado con un esquema denominado *globalNavigation*, como se indica en la figura 4.3.

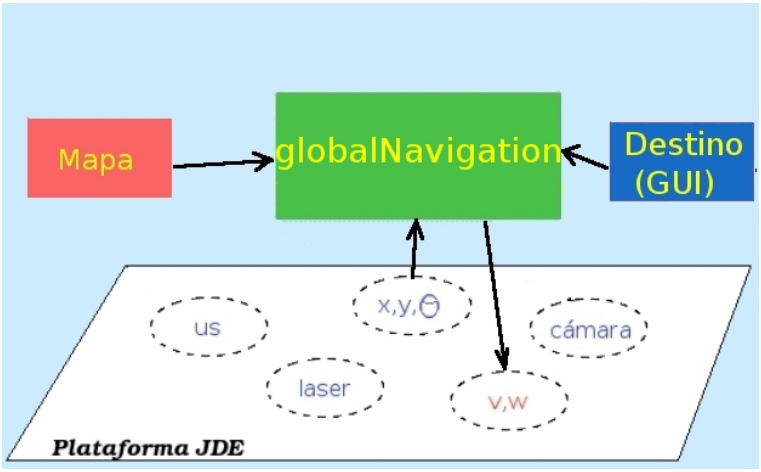


Figura 4.3: Diagrama de E/S del esquema de *GPP*.

A la hora de programar este comportamiento, el esquema se encarga de capturar el escenario, sobre el que posteriormente se genera y expande el campo actualizando la rejilla. Las funciones realizadas se describen a continuación:

1. Capturar información del mundo.
2. Generar la rejilla donde se almacenará el mundo.
3. Crear el campo y expandirlo a lo largo del escenario.
4. Pilotar al robot desde el punto en el que se encuentra hasta el destino seleccionado.

4.3. Construcción del campo virtual

A continuación mostraremos cómo asignar a cada punto del espacio un campo potencial. De este modo, *viajando* a través del gradiente de potenciales, nos aseguraremos de ir siempre por el camino más corto hacia nuestro destino.

Coste del camino

En general, el coste del camino se representa como una función, *F(P)*, donde sumamos el coste *intrínseco* de todos los puntos del espacio que forman el camino, junto con el coste *extrínseco* dado por la distancia que hay desde ese punto hasta su siguiente.

$$F(P) = \sum_{i=1}^{n} I(p_i) + \sum_{i=1}^{n} A(p_i, p_{i+1}) \tag{4.2}$$

En nuestro caso, la función I representa el coste de los puntos del espacio que corresponden a obstáculos. Asignaremos un coste muy elevado en estos casos, mientras que si no es obstáculo será un valor nulo. La longitud del camino vendrá dada por la función A, que en nuestro caso representará la Distancia Euclídea entre dos puntos del camino por los que viajará el robot. Sumando todas las distancias resultantes de esta función, obtendremos el coste total de nuestro camino a seguir.

Función de navegación

La función de navegación consiste básicamente en asignar un campo potencial a cada punto del espacio. De este modo nos evitamos cualquier posibilidad de que pudiéramos *caer* en un mínimo local. La idea principal del gradiente está en asignar la función de navegación a un punto que formará parte del camino de mínimo coste. O lo que es lo mismo:

$$N_k = min_{P_k} F(P_k) \tag{4.3}$$

Siendo *Pk* el camino que comienza en el punto *k*. Si los costes intrínsecos son cero; como es el caso de las celdas *no-obstáculo*, la función de navegación representa la distancia al punto más cercano de destino inmediato. Así, si viajamos en la dirección del gradiente de *N*, estaremos recorriendo con total seguridad la distancia más corta con respecto a nuestro objetivo.

4.4. Expansión del gradiente

Tendremos que calcular la ecuación 4.3 para cada punto de nuestro espacio. Esta operación es muy costosa computacionalmente; en tanto cuanto más grande es el espacio a tener en cuenta, así como en función del número de caminos posibles hacia el destino.

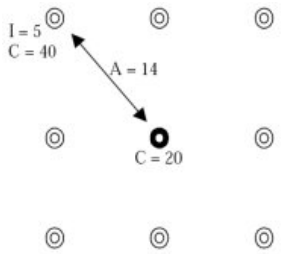


Figura 4.4: Actualización del coste de los vecinos de un punto *activo*.

```
for (i=1;i>=-1;i--) { /* Para cada vecino de la casilla actual */
  for(j=1;j>=-1;j--) {
    if ((fo+i>=0)&&(fo+i<gradientGrid->size)&&
      (co+j>=0)&&(co+j<gradientGrid->size)&&
      (gradientGrid->map[(fo+i)*gradientGrid->size+(co+j)].estado==EMPTY)) {
      /* Si el vecino está VACIO y está en la rejilla */
      if ((i+j)%2!=0) d = frente->distancia +1.; /* los vecinos transversales se les impone distancia 1 */
      else d = frente->distancia +1.414213562; /* los vecinos diagonales se les impone más distancia */
      /* Distribucion de distancias en vecinos:
                    pi 1 pi
                    1  0  1
                    pi 1 pi                  */
      ingresar_celdilla_en_frente(co+j,fo+i,gradientGrid->map[(fo+i)*gradientGrid->size+(co+j)].estado,d);
      /* para evitar la insercion multiple de la misma celdilla */
      gradientGrid->map[(fo+i)*gradientGrid->size+(co+j)].estado=TRATADO;
    }
    else if ((fo+i>=0)&&(fo+i<gradientGrid->size)&&
      (co+j>=0)&&(co+j<gradientGrid->size)&&
      (gradientGrid->map[(fo+i)*gradientGrid->size+(co+j)].estado<=dist_max)&&
      (aux[(fo+i)*gradientGrid->size+(co+j)]==1)&&
      (gradientGrid->map[(fo+i)*gradientGrid->size+(co+j)].estado>0)) {
      /* introducimos en el frente las celdillas propagadas por los obstaculos, con su valor */
      if ((i+j)%2!=0)
        d=frente->distancia+gradientGrid->map[(fo+i)*gradientGrid->size+(co+j)].estado+1.;
      else
        d=frente->distancia+gradientGrid->map[(fo+i)*gradientGrid->size+(co+j)].estado+1.414213562;

      ingresar_celdilla_en_frente(co+j,fo+i,gradientGrid->map[(fo+i)*gradientGrid->size+(co+j)].estado,d);
      gradientGrid->map[(fo+i)*gradientGrid->size+(co+j)].estado=TRATADO;
    }
  }
}
```

Figura 4.5: Actualización del coste de los vecinos de un punto de la rejilla de ocupación.

Comenzaremos aplicando un valor de 0 al objetivo. En cada iteración, aquellos puntos con valor n serán expandidos a su vecinos más cercanos, propagándoles el valor de $n+1$; siempre y cuando éstos no tengan ya un valor menor previamente asignado y no sean obstáculos. El proceso se repetirá hasta que todos los puntos tengan un valor asignado, o bien hayamos llegado al origen del cual partiremos. Cada punto p está rodeado por 8 vecinos, considerando el espacio en dos dimensiones. Para todos sus vecinos q, aplicaremos la ecuación 4.2. Si el nuevo valor de q es menor que el que ya tiene, lo reemplazamos por el nuevo calculado. Así, en un instante determinado t, tendremos un *frente de onda* activo, que estará formado por los puntos que estemos expandiendo en ese momento. Aquellos vecinos que sean actualizados en t, formarán el nuevo frente de onda en el instante $t+1$. Por ejemplo, en la figura 4.4 el nuevo coste del vecino es 39, ya que es menor que su coste original (40). Por lo que será añadido al nuevo frente de onda.

La función de navegación dada por método de expansión del gradiente, nos garantizará el camino de mínimo coste hacia el objetivo.

4.5. Expansión del frente de obstáculos

El mayor problema a la hora de utilizar el algoritmo de expansión del gradiente, es poder generar caminos que sorteen obstáculos. El algoritmo, en su forma original, hace uso de una función de navegación que minimiza el coste del camino, por lo que los puntos del espacio que rodean a los obstáculos tendrán asignado un coste intrínseco:

$$I(p) = Q(d(p)) \tag{4.4}$$

Siendo *d(p)* la distancia del punto p al obstáculo más cercano. Y Q es una función que decrece en función de su argumento. En la práctica esta función la hemos implementado como un frente de onda que parte de los *puntos obstáculo* y se expande hasta un máximo de 7 unidades. Dado que cada unidad de nuestra rejilla tiene una dimensión de 5 cm. del espacio *real*, la separación entre un obstáculo y el camino más cercano a éste (dado por la función de navegación) será de 35 cm. Así evitamos que el robot pueda colisionar, ya que tiene unas dimensiones de 42 cm. de ancho x 60 cm. de largo.

Una vez propagado el frente de obstáculos, dado que su efecto sobre la navegación del robot ha de ser negativo, deberemos invertir el campo.

Esto es debido al cómo está implementado tal algoritmo de expansión de frente. Inicialmente comenzamos con un valor de campo que iteración a iteración se irá incrementando (como un frente de onda que se expande). Así, en el caso del gradiente de distancias la propagación es correcta, ya que comenzamos desde el destino (donde habrá menos valor de campo) hasta el origen, donde el campo será máximo. Acabada la propagación, iremos persiguiendo las celdillas del campo que menor valor tenga.

De este modo, la propagación de los obstáculos se hará debidamente comenzando desde cada *celdilla-obstáculo*, hasta un determinado nivel de propagación; en nuestro caso, 7 niveles. El resultado así obtenido es un gradiente donde el obstáculo tiene el menor valor del campo y, a medida que nos alejamos, el valor es mayor. Pero esto va en contra de la lógica, ya que lo ideal es que cuánto más cerca estemos del obstáculo, mayor sea el valor del campo del gradiente. De ahí la inversión que realizamos. Podemos verlo en el siguiente fragmento de código:

```
/* Invertimos el campo, para que las celdas más próximas a los obstáculos tengan mayor peso */
if ((gradientGrid->map[i].estado!=EMPTY)&&(gradientGrid->map[i].estado!=OCCUPIED)) {
  valor=gradientGrid->map[i].estado - PESO; /* INVERSIÓN */
  gradientGrid->map[i].estado=(gradientGrid->map[i].estado-(valor*CORRECCION))+6;

  if (gradientGrid->map[i].estado>dist_max)
    dist_max=gradientGrid->map[i].estado; /* Controlamos hasta donde propagan los obstaculos */
}
```

Figura 4.6: Inversión del frente de obstáculos generado inicialmente.

4.6. Control de velocidad

Este método lleva implícito la imposición de una dirección a seguir en cada celda. La cuestión es qué velocidad de tracción y rotación hay que ordenar al robot, según la celda en la que nos encontremos. Dada la diversidad de circunstancias que podemos encontrarnos, hemos empleado la lógica borrosa para dar más realismo a la toma de decisiones de cómo actuar. Hemos de tener en cuenta cuatro factores para generar nuestras reglas de lógica:

- Diferencia entre el ángulo de la casilla a viajar y el ángulo que lleva el robot. Estará comprendido entre 0 y 180 grados.
- Velocidad de tracción actual (v).
- Velocidad angular actual (w).
- Distancia a la que estamos del objetivo. A medida que estemos más cerca, deberemos disminuir la velocidad de tracción.

4.7. Interfaz gráfico. Interacción con el usuario

Aquí expondremos los resultados a nivel gráfico, bajo el visualizador desarrollado con la librería *OpenGL*. Recreamos el entorno en el que se moverá el robot en la realidad, que en nuestro caso es el ala derecha de la primera planta del Departamental II de la Universidad Rey Juan Carlos (parte derecha de la imagen). Para poder navegar en el entorno virtual, tenemos una cámara virtual que se controla con el ratón.

Además, como ya se ha comentado, el objetivo del presente proyecto es desarrollar un prototipo enfocado hacia un robot guía para tal zona de la universidad. De ahí que podamos seleccionar el destino según el despacho deseado. Éstos aparecen numerados en la parte izquierda de la imagen.

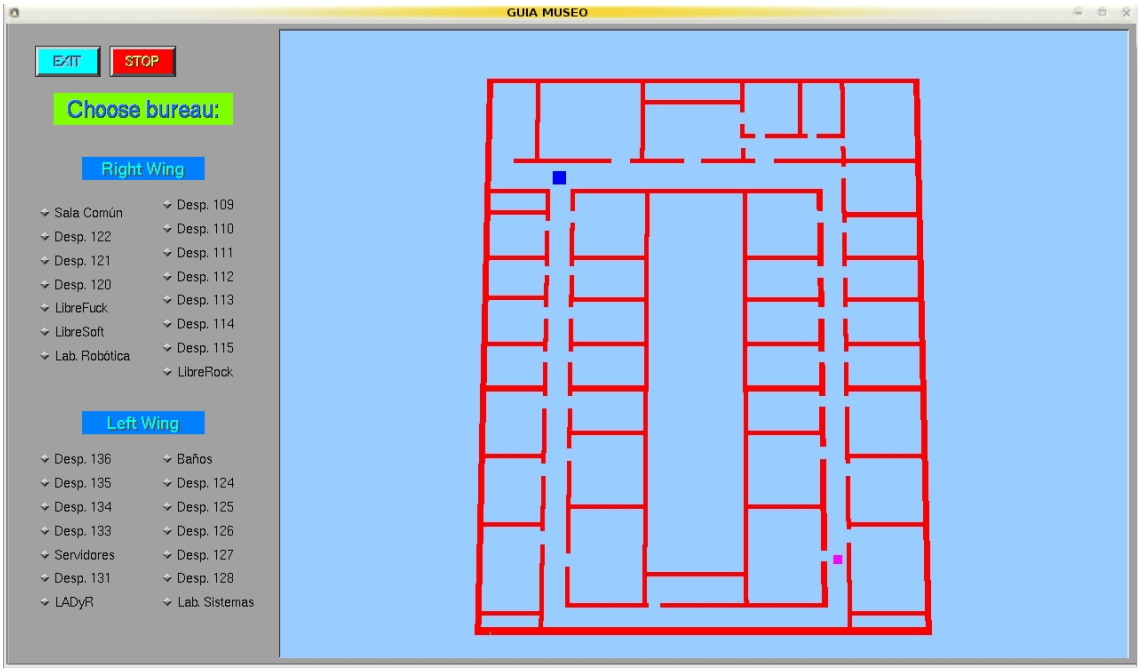


Figura 4.7: Aspecto general del simulador de navegación.

En esta imagen, por ejemplo, está el robot situado a la entrada del departamental (baliza azul) esperando que le soliciten un destino. Por defecto está establecido el Laboratorio de Robótica (baliza rosa).

4.8. Experimentos

En esta sección se detallan los aspectos más destacables de la puesta en marcha del algoritmo anteriormente explicado, haciendo incapié en el entorno visual empleado para observar los resultados.

4.8.1. Ejemplo de ejecución típica

A continuación detallaremos un ejemplo[1] de ejecución típica del esquema de navegación global. Reflejaremos desde el instante inicial en que el usuario selecciona el destino deseado hasta que el robot llega a tal posición, pasando por las distintas fases de propagación del campo de gradiente, así como del de distancias.

Propagación del campo de obstáculos

En la siguiente imagen podemos apreciar cómo una vez hemos señalado el destino (marca rosa) al que irá el robot (marca azul), se empieza a generar el gradiente de obstáculos (zonas rojas, y sus alrededores oscuros).

[1]Vídeo disponible en http://jde.gsyc.es/index.php/jmvega_guide_robot

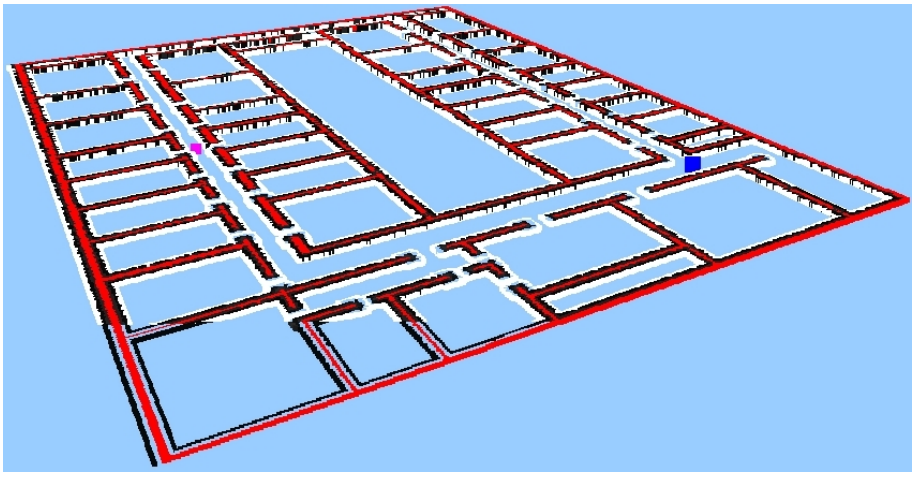


Figura 4.8: Fase inicial del algoritmo GPP. Generación obstáculos.

Tal frente de obstáculos empieza con un valor de campo no excesivamente elevado, porque a posteriori habrá que invertirlo, como ya explicamos en su momento (4.5).

Inicio del gradiente de distancias

Podemos apreciar el cambio de tonalidad en los colores circundantes a los obstáculos entre las figuras 4.8 y la que vemos a continuación. Esto es debido a la inversión del frente de obstáculos explicada en 4.5.

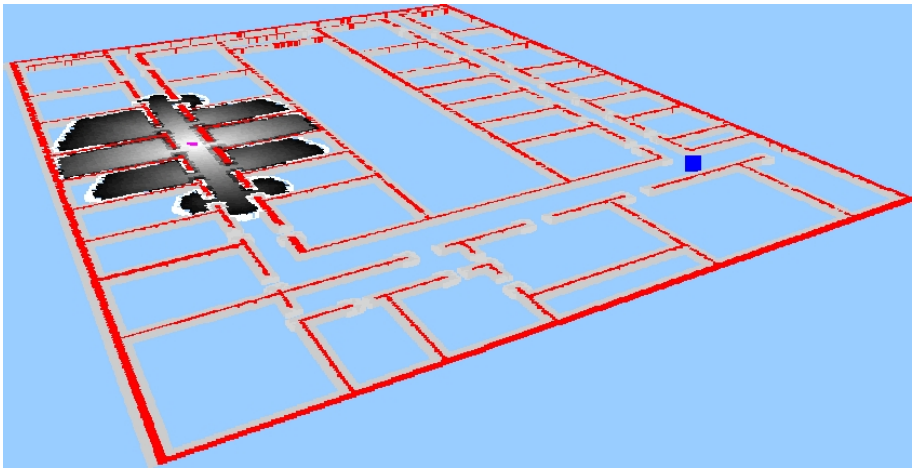


Figura 4.9: Fase inicial de propagación del gradiente de distancias.

Asimismo apreciamos el inicio de propagación de distancias desde el destino establecido (el Despacho 113). En este punto el frente es totalmente blanco, y se va oscureciendo paulatinamente.

Intermedio del gradiente de distancias

En esta segunda toma de la propagación del frente de distancias, apreciamos cómo se ha *fusionado* con el frente de obstáculos, de ahí que el valor del gradiente final de distancias sea paulatinamente más oscuro, tomando como referencia el destino (color blanco). Y además en las posiciones correspondientes al frente de obstáculos el campo se oscurece aún más (efecto de repulsión ejercido por los obstáculos).

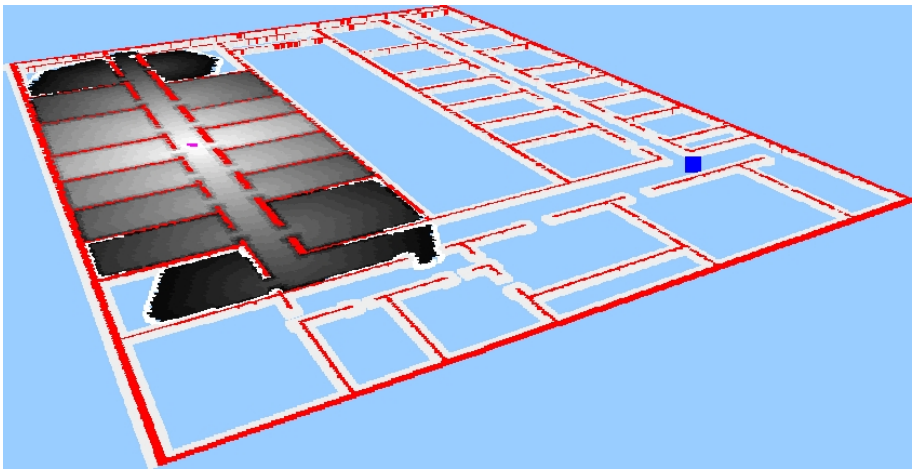


Figura 4.10: Fase intermedia de propagación del gradiente de distancias.

Final de propagación del gradiente de distancias

Ahora podremos apreciar el gradiente en su totalidad, y cómo cuando el frente llega a la posición de origen del robot, éste comienza automáticamente a seguir el gradiente generado y la propagación finaliza.

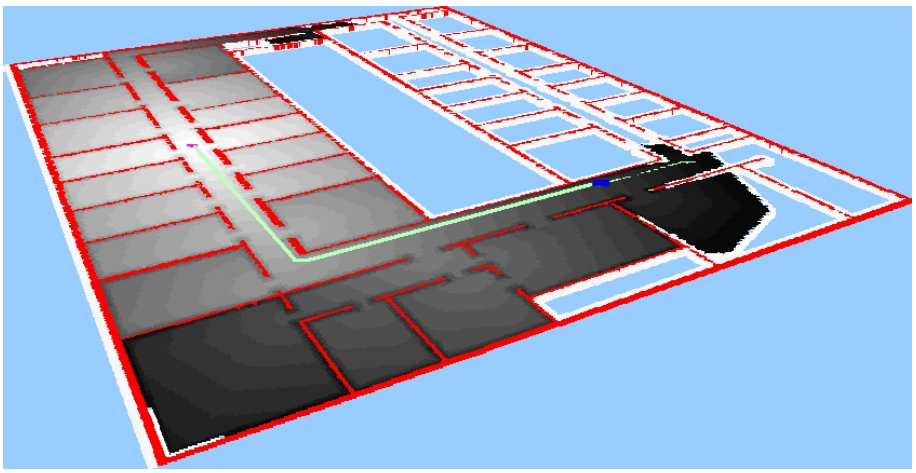


Figura 4.11: Fase final de propagación del gradiente de distancias.

El rastro verdoso que vemos en la parte inferior de la imagen es la ruta que seguirá el robot en búsqueda de su destino. Las imagen 4.12 refleja las distintas etapas

correspondientes al movimiento descrito por el robot hasta que finalmente se para cuando llega al objetivo.

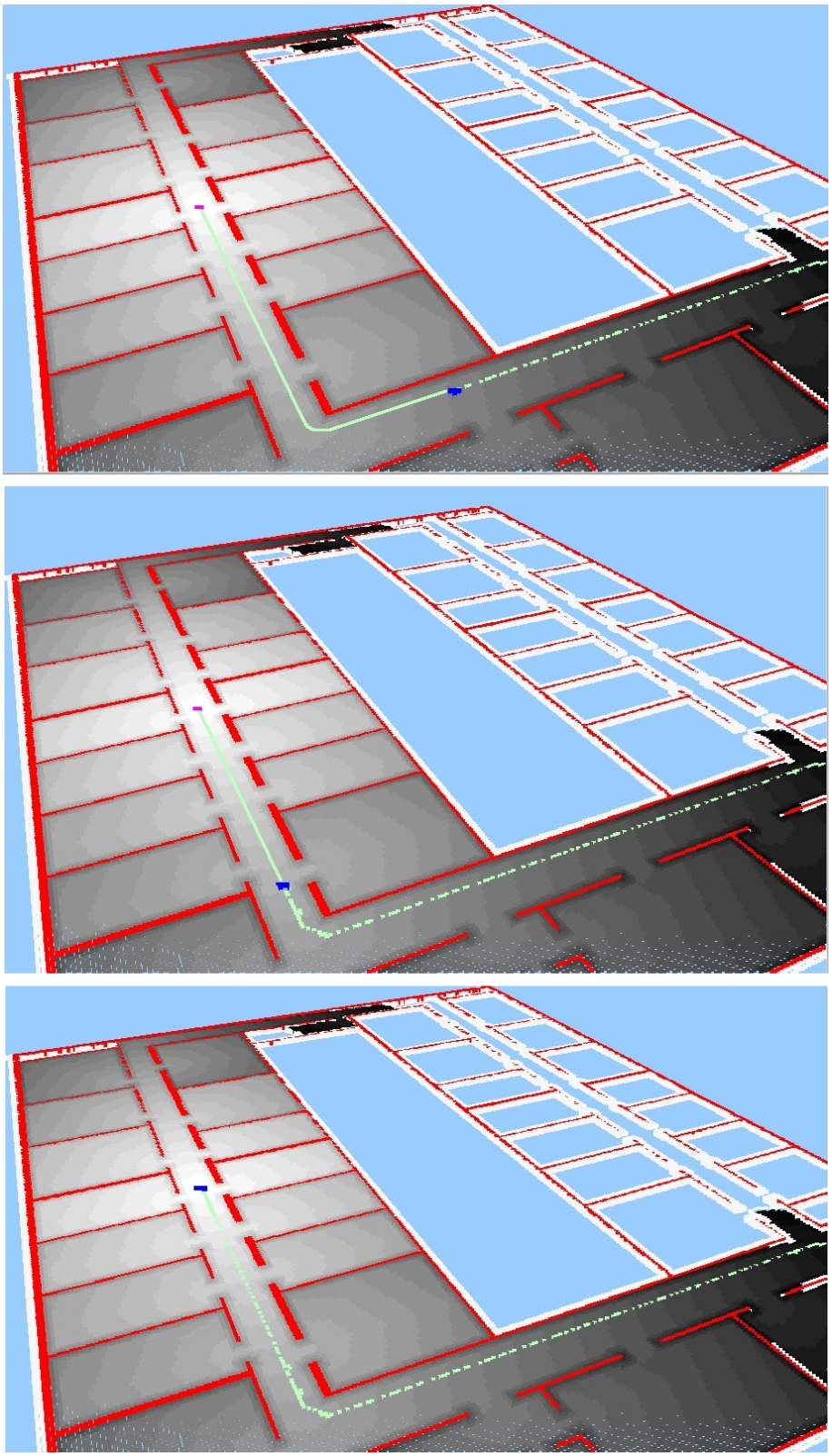


Figura 4.12: Avance del robot por la ruta calculada.

Tiempos

El algoritmo tiene un coste proporcional al número de puntos del espacio. Tendremos que calcular dos gradientes: uno para el coste de obstáculos (ver apartado anterior 4.5), y otro para la función de navegación (ver 4.3). La mayor parte del tiempo se emplea en generar el gradiente correspondiente a los obstáculos.

En un PC normal (Intel Centrino, 1.5 GHz). Implementación en C, para un espacio de 40m. x 40m. y con celdas de 5cm. de ancho y alto, la función de navegación es calculada en 3 segundos, al igual que el frente de obstáculos termina de generarse a los 2 segundos.

Capítulo 5
Localización

La imagen que un solo hombre puede formar es la que no toca a ninguno. Infinitas cosas hay en la tierra; cualquiera puede equipararse a cualquiera. Equiparar estrellas con hojas no es menos arbitrario que equipararlas con peces o con pájaros. En cambio, nadie no sintió alguna vez que el destino es fuerte y es torpe, que es inocente y es también inhumano.

J. L. Borges, *El Aleph*

La localización es uno de los problemas clásicos dentro del área de la robótica móvil. El problema consiste en que un robot estime su posición automáticamente dentro de un determinado mundo usando para ello sus sensores y la información de su entorno, que en general se ofrece en forma de un mapa. Actualmente hay técnicas avanzadas de localización y creación de mapas simultáneamente, *SLAM* (*Simultaneous Localization and Mapping*).

5.1. Técnicas de localización

Dentro de la localización, podemos distinguir dos problemas distintos. El primero y más sencillo de abordar es el de partir de una posición inicial conocida. En este caso, para determinar la posición final del robot habrá que estimar los errores acumulados por los odómetros (en el caso de robots con ruedas). El segundo problema, y el más difícil, es la localización global. En este caso se desconoce la posición inicial del robot y es justo el que abordamos en este proyecto.

5.1.1. Sensores para localización

La recogida de información de los sensores en los robots es un elemento indispensable en la resolución de la localización. Los principales sensores utilizados en localización para robots con ruedas son los *encoders* (figura 5.1-a), que devuelven una estimación de la posición del robot basada en contar los giros de sus ruedas. Estos sensores presentan errores por desplazamiento, holguras, falta de precisión, etc. Los errores son acumulativos y por ello si no se corrigen podemos llegar a obtener una localización demasiado imprecisa. Existen dos tipos de errores en la odometría:

- *Errores sistemáticos*. Dependen de las características del robot y los sensores. Son errores que vienen de fábrica: diferentes diámetros de ruedas, falta de alineamiento de las mismas, resolución limitada de los encoders y velocidad limitada de muestreo en éstos. Estos errores se pueden corregir mediante calibración, ruedas auxiliares o encoders adicionales.

- *Errores no sistemáticos*. Son impredecibles y son debidos a agentes externos al robot, como puede puede ser la naturaleza del suelo, que por ejemplo cause el patinaje de las ruedas.

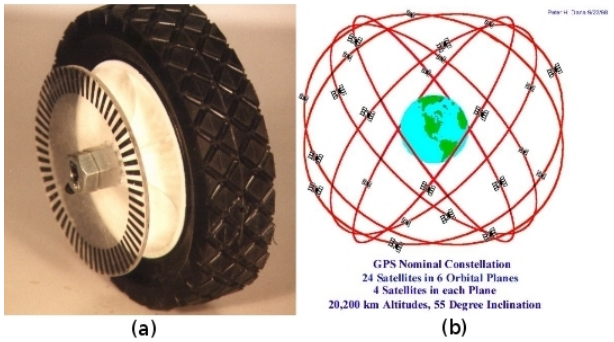


Figura 5.1: Sensores de estimación de posición: (a) *encoders* y (b) sistema *GPS*.

Existen también sensores como las cámaras, los láseres o los sónares que por sí solos no dan información específica de posición y que para utilizarlos en la autolocalización es necesario añadirles la información del mapa. En nuestro caso hemos empleado la cámara junto con un mapa visual del techo del entorno por donde se mueve el robot.

Otro tipo de sensor utilizado para resolver el problema de la localización es el receptor *GPS* (*Global Positioning System*, figura 5.1-b). Este sistema está basado en el envío de coordenadas de posición y tiempo a receptores en la tierra a través de satélites. Es un sistema muy útil para la localización en exteriores. Sin embargo en entornos interiores, como pueden ser los edificios, la señal proveniente de los satélites no entra con la suficiente fuerza. Por tanto, este mecanismo es utilizado primordialmente para la localización y planificación de rutas en vehículos.

5.1.2. Métodos de localización

Existe una gran cantidad de técnicas de estimación de posición que intentan resolver el problema de la localización sin el uso de sensores específicos:

- *Localización con balizas*. Ésta técnica permite localizar al robot en un entorno restringido mediante el emplazamiento específico de un determinado número de balizas en posiciones conocidas. Para estimar la posición se puede utilizar la

triangulación basándose en el ángulo con que se ven las balizas y la trilateración basándose en la distancia a las balizas.

- *Los filtros de Kalman.* Es otra técnica que trata de estimar recursiva y periódicamente la posición de mínima varianza fusionando información parcial e indirecta sobre localización. Su principal limitación es que es una técnica unimodal y exclusivamente gaussiana ([Isard y Blake, 1998]). Este método no soporta bien el ruido de lecturas sensoriales ni entornos dinámicos y no es capaz de manejar múltiples hipótesis.

- *Scan matching.* Aquí se emplean mapas locales para compararlos con lecturas sensoriales, alineando estas lecturas con los diferentes mapas en posiciones cercanas a la que creemos que está el robot. Para ello se necesita una estimación de la posición inicial del robot representada como una distribución gaussiana que se va actualizando con las lecturas sensoriales.

- *Localización probabilística ([Thrun, 2000]).* Es adecuada para interiores ya que incorpora incertidumbre de acciones y de observaciones que se acoplan a la incertidumbre que muestran los sensores. Este tipo de localización consiste en determinar la probabilidad de que el robot se encuentre en una determinada posición a través de sus lecturas sensoriales y movimientos a lo largo del tiempo. A cada posible posición se le asocia una probabilidad reflejando la verosimilitud de ser la posición actual del robot. Esta probabilidad se va actualizando con la incorporación de nuevas lecturas y movimientos del robot. Estas técnicas nos permiten localizar al robot aun desconociendo su posición inicial, permitiendo representar situaciones ambiguas, cuya ambigüedad irá desapareciendo posteriormente. La eficiencia computacional de estas técnicas generalmente depende del tamaño del mapa.

La intuición de la localización probabilística se puede apreciar en la figura 5.2. En un primer paso, se asume un espacio unidimensional en el que el robot sólo se puede desplazar horizontalmente y desconoce su posición inicial. El estado de incertidumbre se representa como una distribución uniforme sobre todas las posibles posiciones.

En el siguiente paso, suponemos que el robot detecta mediante sus sensores que se encuentra enfrente de una puerta. Esto se refleja en el aumento de la verosimilitud en las zonas donde están las puertas y la disminución en zonas donde no hay puertas. La información disponible hasta el momento actual es insuficiente para poder determinar inequívocamente la posición del robot. Si el robot se desplaza, la distribución se desplaza de forma análoga, como se refleja en el tercer paso.

En el cuarto y último paso el robot ha detectado, de nuevo a través de sus sensores, que se encuentra frente a una puerta por ello aumenta la verosimilitud en esa posición que sumada a la acumulación del tercer paso, nos lleva a la conclusión de que es muy probable que el robot se encuentre en la segunda puerta.

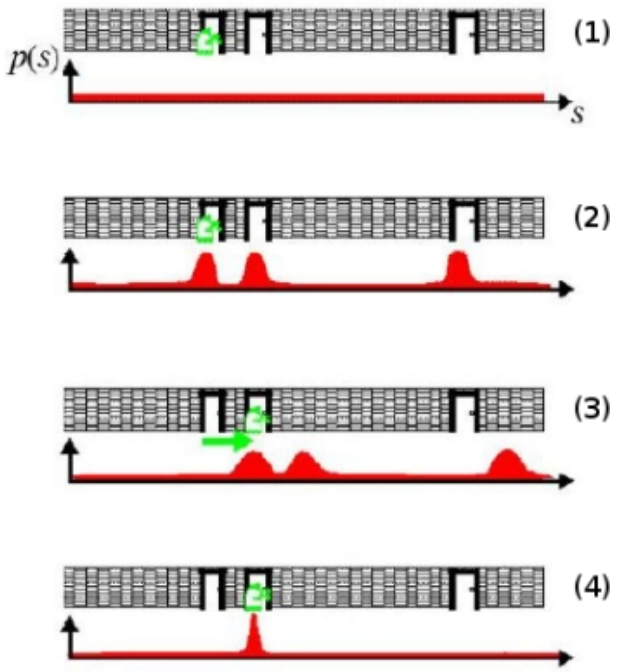


Figura 5.2: Localización probabilística.

Dentro de la localización probabilística, dos de las técnicas más importantes y que se utilizan para resolver la localización son: *(i)* la probabilística sin muestro, que implica mucho tiempo de cómputo debido a que almacena y actualiza la distribución de probabilidades para *todas* las posibles posiciones; y *(ii)* con muestreo, que se utiliza para agilizar el tiempo de cómputo y hacer de la localización un proceso escalable a grandes entornos. En este proyecto hemos empleado este último mecanismo, ya que nuestro objetivo es presentar una solución enfocada hacia un robot guía que operará en un entorno relativamente amplio, como es el Departamental II de la universidad. Para ello nos hemos basado en el *Método de MonteCarlo*, en concreto, los filtros de partículas.

5.2. Localización con filtro de partículas

A continuación se introducen los fundamentos teóricos en los que se basa el Método de MonteCarlo, para posteriormente detallar cómo se ha diseñado y programado la implementación del algoritmo en este proyecto.

5.2.1. Fundamentos teóricos

La idea de los filtros de partículas es la de mantener una representación muestreada de la distribución de probabilidad, de manera no uniforme. La probabilidad no se evalúa en todo el espacio de estados; sólo en un conjunto de muestras llamadas partículas, dispersas a lo largo de él. La estimación probabilística independiente de cada partícula viene determinada como la verosimilitud de ese estado a raíz de las observaciones

sensoriales.

Una *muestra* es una partícula, siendo ésta un punto *(x,y,theta)* del espacio para entornos planos como el nuestro, con un determinado peso en función de su probabilidad (p_i). El conjunto muestral representa las posibles posiciones del robot en el marco de referencia ([Fox *et al.*, 1999], [Mackay, 1999] y [Barrera *et al.*, 2005]).

Cada partícula tiene asociada una imagen teórica que, comparada con la imagen real, determinada la probabilidad asociada a tal partícula. La imagen teórica es aquella imagen que teóricamente vería el robot si estuviera en la posición y orientación de la partícula. Por otro lado, la imagen real es la que ve el robot a través de su cámara.

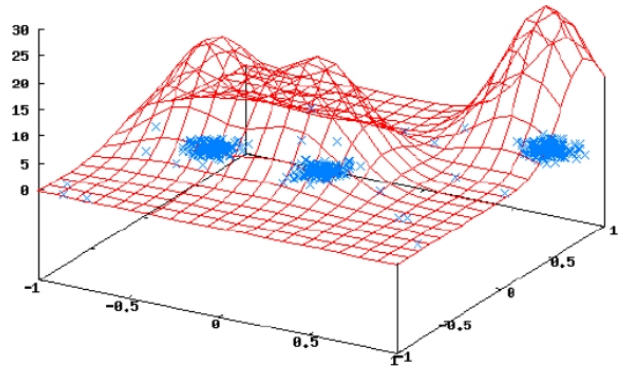


Figura 5.3: Muestreo de la densidad de probabilidad.

Resumidamente, se tiene una población de N muestras, cada una con un peso, que representan las posibles posiciones del robot. Se van generando nuevas poblaciones a partir de la población anterior junto con las imágenes que obtenemos por la cámara (modelo de observación visual) y los movimientos en *(x,y,theta)* producidos por el robot (modelo de movimiento).

Inicialmente el conjunto de muestras se distribuye de manera aleatoria por todo el espacio tridimensional abarcando las posibles posiciones del robot. Posteriormente los pesos y las posiciones de las partículas van variando a medida que se incorporan las observaciones visuales y odométricas. Las observaciones visuales suben el peso a las partículas compatibles con la observación (imagen real de la cámara). Las lecturas odométricas, que reflejan el movimiento realizado por el robot, trasladan las partículas siguiendo el desplazamiento y giro medido por los *encoders*.

Gracias al paso de remuestreo, donde se utiliza el *Algoritmo de la Ruleta*, las partículas con más peso en la antigua población generan más hijos para la siguiente y eso lleva a un desplazamiento neto de las partículas hacia posiciones compatibles con las observaciones sensoriales.

El algoritmo realiza iteraciones continuas, cada una de ellas con tres pasos fundamentales en los que se basa el filtro de partículas (figura 5.5):

- *Modelo de movimiento*. Se trasladan las muestras según el desplazamiento y giro ejercido llevado a cabo por el robot. La posición de las nuevas partículas vendrán determinadas por este desplazamiento y un cierto ruido incorporado.
- *Modelo de observación*. Se calculan las nuevas probabilidades *a posteriori* de las muestras a partir de la última imagen captada.
- *Remuestreo*. Se genera un nuevo conjunto muestral a partir de las verosimilitudes anteriores.

Con éste método no se guarda explícitamente la historia de las verosimilitudes del conjunto muestral en instantes anteriores, sino que va vinculada a las nuevas distribuciones de las muestras, las cuales se van concentrando alrededor de las posiciones más probables.

5.2.2. Esquema de localización

Este algoritmo se ha diseñado como un esquema *Jde*, como se indica en la figura 5.4. Se trata de un esquema perceptivo denominado *visualLocalization*. Visto como *caja negra*, el esquema admite como entradas la odometría *(x, y, theta)* dada por los *encoders*, las imágenes proporcionadas por la cámara y el mapa visual del techo, que hemos implementado con una rejilla bidimensional para albergar la información del escenario.

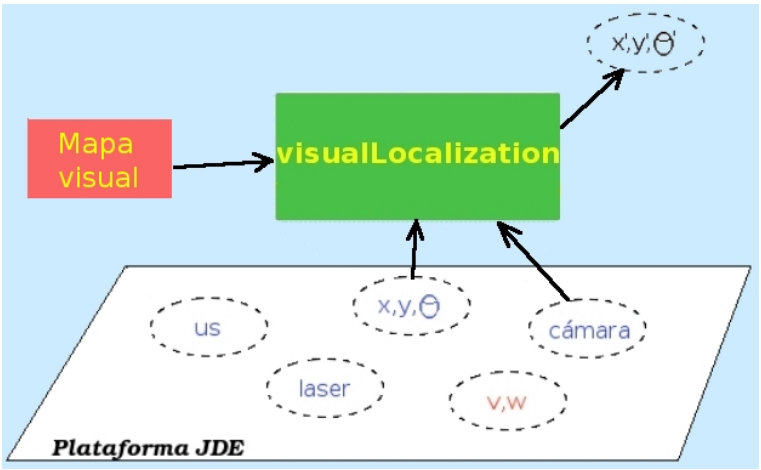


Figura 5.4: Diagrama de E/S del esquema de localización.

Como salida, el esquema nos vierte la posición *(x,y,theta)* estimada de dónde se encuentra el robot. Siempre daremos como posición estimada la posición de la partícula con mayor probabilidad en cada iteración o una media de las partículas másprobables. Al principio de los tiempos habrá muchas posiciones posibles pero a medida que el algoritmo avanza se van descartando posiciones y va convergiendo en una única

posición.

El esquema *visualLocalization* se encarga de realizar las funciones del esquema global de la figura 5.5, que conforman el algoritmo de localización a través del filtro de partículas. Estas funciones son:

1. Captura del mapa visual desde fichero.
2. Localización iterativa con filtro de partículas.
   a) Modelo de observación.
   b) Modelo de movimiento.
   c) Remuestreo.

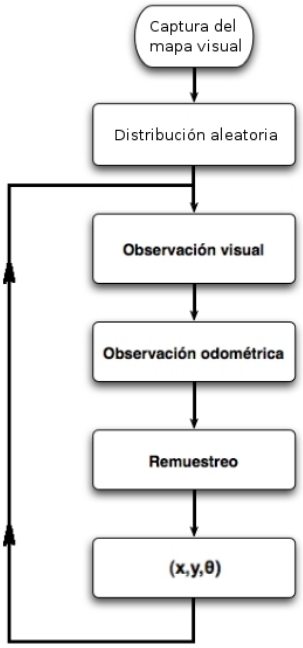


Figura 5.5: Pseudocódigo de localización con filtro de partículas.

5.3. Mapa visual

El mapa visual ejerce de guía sobre el robot. Este mapa dado al robot contiene información del mundo por el que se moverá éste. La información a capturar es la del techo del Laboratorio de Robótica de la universidad, el cual cuenta con balizas visuales reales. Como puede verse en figura 5.6, las balizas que utilizamos son marcas compuestas por la combinación de 4 colores distintos (rojo, cyan, verde y magenta). El esquema recibe la información del mapa a través de una imagen *.PPM* directamente, para posteriormente someterla a filtrado. De este modo nos quedaremos únicamente

con los colores que nos interesan, ignorando el resto.

Basándonos en la misma idea empleada en la solución del robot guía *Minerva* (capítulo 1), la imagen del techo es una *foto-composición* formada a partir de varias fotografías tomadas por el robot real desde distintas posiciones. Ése es el mapa que hemos usado en el modelo de observación (ver sección 5.4).

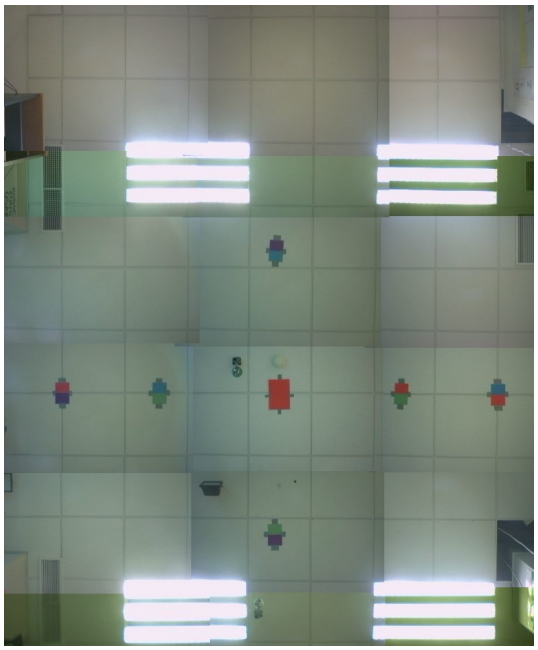


Figura 5.6: Mapa visual. *Foto-composición* del techo real del Laboratorio de Robótica.

Para la captura de la información del mapa se utiliza un analizador sintáctico que va leyendo línea a línea dos archivos, almacenando la información necesaria:

- Fichero de configuración del mundo. Este archivo contiene la ruta donde se encuentra la imagen del techo y la resolución que tiene cada píxel dentro de esa imagen. Adicionalmente, este fichero también contiene el puerto donde el simulador o robot real nos vierte datos a nuestra aplicación, así como la posición inicial *(x,y,theta)* del robot en ese mundo.

- Imagen PPM a color del techo (figura 5.6). Este fichero contiene en las primeras cinco líneas, entre otras cosas, el ancho y el alto que ocupa la imagen. En las siguientes líneas contiene los *bytes* que componen la imagen.

Toda la información del mundo es almacenada en la siguiente estructura de datos:

```
typedef struct {
  int ancho;
  int alto;
  int dimension;
  float resolucion;
  float x_robot;
  float y_robot;
  float theta_robot;
} Tmundo;
```

Figura 5.7: Estructura de datos para almacenar información del mundo.

Con la información de la imagen del mapa y el fichero del mundo hacemos uso de la biblioteca *Gridslib* ya comentada en el capítulo 3, para que el esquema se construya un mapa interno en forma de rejilla bidimensional, almacenando por cada píxel de la imagen el color que le corresponde dentro de la rejilla. Quedándonos con los colores representativos únicamente. Veámos:

```
if ((int) casillas[celda]==54) // ROJO
  (*grid).map[k].estado = -6;
else if ((int) casillas[celda]==46) // MAGENTA
  (*grid).map[k].estado = -7;
else if ((int) casillas[celda]==-74) // VERDE
  (*grid).map[k].estado = -8;
else if ((int) casillas[celda]==-55) // CYAN
  (*grid).map[k].estado = -9;
else
  (*grid).map[k].estado = 0; // COLOR IRRELEVANTE
```

Figura 5.8: Codigo relativo al filtrado de colores del techo.

Para tener el mapa correctamente almacenado es necesario anclar la imagen a posiciones espaciales concretas. Esto se hace a través de los datos del fichero del mundo realizando conversiones entre tamaños de rejilla y el de los píxeles.

5.3.1. Generación de imagen sintética

Como ya se comentó en el apartado 5.2.1, la idea de los filtros de partículas es la de mantener una representación muestreada de la distribución de probabilidad. Cada muestra es una partícula, con posición *(x, y, theta)* y una determinada probabilidad asociada. Esta probabilidad está directamente relacionada con la información que nos aporta esa partícula, y que calculamos consiguiendo la imagen que teóricamente vería el robot si ocupara la posición de tal partícula (imagen teórica) y comparándola con la imagen real de la cámara.

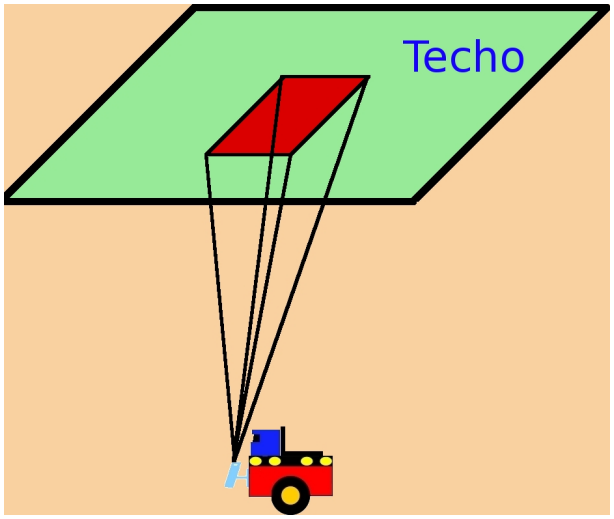


Figura 5.9: Imagen sintética de 320x240 vista desde una determinada posición.

El algoritmo se encarga de calcular para cada partícula una imagen teórica con un cono de visión determinado por el ángulo de visión de la cámara. Sabiendo el tamaño real que cubre del techo la cámara en una imagen de 320x240 píxeles, podemos determinar la parte del techo que se aprecia en cada uno de los píxeles de esa imagen teórica. Obviamente suponiendo que la cámara estará situada siempre a la misma distancia del techo. La función que realiza esta conversión es la siguiente:

```
/* Calculamos la coordenada (x, y) correspondiente en un sistema de referencia
 local solidario al robot, dándonos un píxel de esa imagen (entre 320 y 240 en
 u y v respectivamente). */
void pixel2coordenadaLocal (int u, int v, int* x, int* y) {
  int variacionPixelX = anchoVentanaReal / 320; // milímetros / píxeles
  int variacionPixelY = altoVentanaReal / 240; // milímetros / píxeles

  *x = -(anchoVentanaReal/2) + (u*variacionPixelX);
  *y = (altoVentanaReal/2) - (v*variacionPixelY);
}
```

Figura 5.10: Conversión de un píxel del techo a coordenadas relativas al robot.

Repitiendo esto para cada píxel, viendo el color que el mapa visual tiene para esa parte del techo sabemos el color de cada píxel de la imagen teórica, el resultado es una imagen de 320x240 píxeles correspondiente a la parte del techo visualizada desde la posición de la partícula. El proceso completo es realizado por el siguiente código:

```
void filtroTeorica (float posParticleX, float posParticleY, float thetaParticle) {
  int i,j,c,row,offset,R,G,B;
  float X, Y;
  int x, y;
  Tcolor myColor;

  for (i=0; i<(SIFNTSC_COLUMNS*SIFNTSC_ROWS)/100;i++) {
    c=i%(SIFNTSC_COLUMNS/FACTOR_MUESTREO);
    row=i/(SIFNTSC_COLUMNS/FACTOR_MUESTREO);
    j=(row*(SIFNTSC_COLUMNS)+c)*FACTOR_MUESTREO;

    pixel2coordenadaLocal (c*FACTOR_MUESTREO, row*FACTOR_MUESTREO, &x, &y); // paso a coordenadas relativas al robot
    local2absoluto (posParticleX, posParticleY, thetaParticle, x, y, &X, &Y); // paso a coordenadas absolutas en el mundo
    myColor = devolverColor (X, Y);

    myHSV2 = (struct HSV*) RGB2HSV_getHSV ((int)myColor.R,(int)myColor.G,(int)myColor.B); // pasamos de RGB -> HSV

    if ((((myHSV2->H*DEGTORAD >= 0) && (myHSV2->H*DEGTORAD <= HmaxRojo)) ||
        ((myHSV2->H*DEGTORAD >= HminRojo) && (myHSV2->H*DEGTORAD <= 6.28))) &&
        ((myHSV2->S >= SminRojo) && (myHSV2->S <= SmaxRojo)))
      B = 0; G = 0; R = 255; // Filtro para el rojo
    else if (((myHSV2->H*DEGTORAD >= HminVerde) && (myHSV2->H*DEGTORAD <= HmaxVerde) &&
      (myHSV2->S >= SminVerde) && (myHSV2->S <= SmaxVerde)))
      B = 0; G = 255; R = 0; // Filtro para el verde
    else if (((myHSV2->H*DEGTORAD >= HminCyan) && (myHSV2->H*DEGTORAD <= HmaxCyan) &&
      (myHSV2->S >= SminCyan) && (myHSV2->S <= SmaxCyan)))
      B = 255; G = 255; R = 0; // Filtro para el cyan
    else if (((myHSV2->H*DEGTORAD >= HminMorado) && (myHSV2->H*DEGTORAD <= HmaxMorado) &&
      (myHSV2->S >= SminMorado) && (myHSV2->S <= SmaxMorado)))
      B = 255; G = 0; R = 128; // Filtro para el morado
    else
      B = 128; G = 128; R = 128; // Si no se ha filtrado ningun color

    obs_teorica_filtrada[i*4+0]=(unsigned char)(int)B;
    obs_teorica_filtrada[i*4+1]=(unsigned char)(int)G;
    obs_teorica_filtrada[i*4+2]=(unsigned char)(int)R;
  }
}
```

Figura 5.11: Obtención de una imagen teórica filtrada de 80x60 píxeles.

En los códigos de las figuras 5.10 y 5.11 se puede apreciar que en primer lugar se pasa cada uno de los píxeles de los 320x240 a coordenadas relativas a la posición de la partícula que estamos tratando. Una vez hecho este paso, ya se puede pasar a coordenadas absolutas en el mundo, y podemos ver qué píxel del techo corresponde con tal posición *(X, Y)* absoluta. Así, repetiremos este proceso hasta completar la imagen sintética completa (320x240 píxeles) correspondiente a una partícula.

5.4. Modelo de observación

Este modelo es el que captura toda la información de posición que tiene la imagen de la cámara dentro del marco probabilístico. Calculamos las probabilidades de estar en cada posición a la luz de la imagen observada. Este cálculo consiste en obtener la distancia entre la imagen real resumida y la imagen teórica resumida calculada para cada posición. Esta distancia es básicamente el número de píxeles diferentes entre las imágenes, por lo tanto, si la distancia es baja la probabilidad de ser iguales será alta.

$$P_{obs}(x, y, theta/img_{real}) = f(img_{real}, img_{teorica}) \tag{5.1}$$

Se realizan varios pasos fundamentales hasta llegar al cálculo de las probabilidades: filtrado y resumen de imagen observada, cálculo de imagen teórica para cada posición

posible y cálculo de la distancia entre imágenes.

5.4.1. Resumen de imágenes

Debemos capturar información de posición desde la imagen que se obtiene de la cámara del robot. Contamos con una imagen cruda (320x240 píxeles) en formato *RGB*. Convirtiendo la imagen a un espacio de colores *HSV*, filtraremos la imagen para obtener los colores de los objetos relevantes de nuestro mundo, que son las balizas visuales del entorno. El motivo de cambiar al espacio *HSV* es porque resulta más robusto que *RGB* ante variaciones de luminosidad.

Para hacer este proceso adecuado para trabajar en tiempo real, con el robot físico, recorreremos esta imagen de forma muestreada; esto es, considerando tan sólo 80x60 píxeles. Es un proceso de muestreo puro: tomamos un píxel de cada cuatro columnas y una fila de cada cuatro.

El siguiente paso consiste en resumir en 4x3 píxeles la imagen filtrada. Este proceso consiste en recorrer la imagen filtrada por *zonas* (12 zonas con 4 columnas y 3 filas) contando los píxeles que son de colores relevantes. Si el número de píxeles en la imagen de 80x60 supera un cierto umbral, lo incorporamos a la imagen resumida de 4x3 píxeles. Una ilustración de un filtrado simple podemos verla en la figura 5.12.

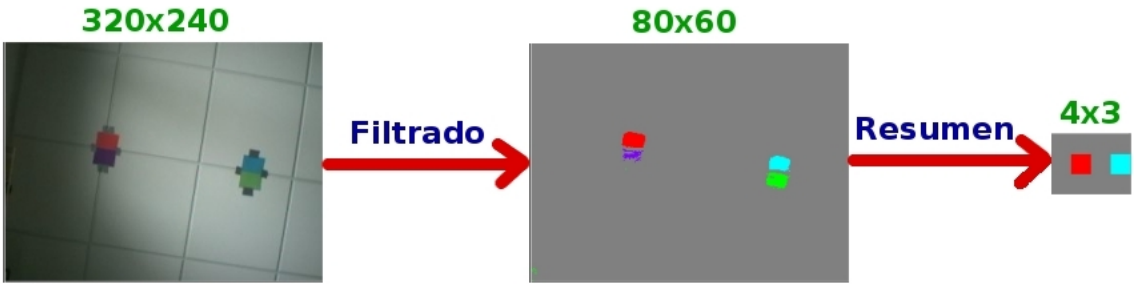


Figura 5.12: Filtrado de una parte del techo tomada por el robot real.

En el caso de que en una misma zona de la imagen se encuentren varios colores relevantes, se establecen las prioridades de color para determinar qué color es el *dominante* de esa zona. En la figura 5.12 podemos ver cómo se almacena en la imagen resumida el color rojo y cyan, respectivamente. El rojo, porque es el color que proporciona más información de la zona que comparte con el magenta; asimismo, el cyan proporciona más información de su zona que el verde. Los colores no dominantes de las respectivas zonas se nombrarán de aquí en adelante como colores *subdominantes*, ya que tienen también su importancia en nuestro algoritmo, aunque no sean prioritarios.

Una vez realizado el resumen de la imagen real que obtenemos de la cámara, es necesario poder relacionar esta imagen simplificada con las posiciones. Para ello usamos el mapa extrayendo por cada posición una imagen teórica resumida también de 4x3

píxeles. Esta imagen teórica se compara con la que hemos obtenido como resumen de la imagen de la cámara real para determinar el parecido que tienen, conformando así la probabilidad de ser o no posiciones válidas según la imagen real percibida. Aquellas posiciones en las que la imagen real y su imagen teórica sean muy parecidas tendrán alta probabilidad; las que no sean parecidas tendrán baja probabilidad.

Al igual que hicimos con la imagen real, ahora también generamos la imagen teórica de una forma muestreada (80x60 píxeles), de ahí que aparezca en el código (5.11) el factor de muestreo, determinado experimentalmente. Por último, esa imagen teórica de 80x60 píxeles es resumida en 4x3 zonas, tal como hicimos en el proceso de imagen real. En primer lugar contamos los píxeles de cada color por zona:

```
void resumirImagenTeorica() {
  int i,j;
  int zona = 0; // zona en la que estamos
  int miZona = 0;
  int pos;

  for (i=0; i<NUM_COLUMNAS*NUM_FILAS; i++) { // inicialización de las zonas
    colores_encontrados_teorica[i].num_pixel_azul=0;
    colores_encontrados_teorica[i].num_pixel_morado=0;
    colores_encontrados_teorica[i].num_pixel_rojo=0;
    colores_encontrados_teorica[i].num_pixel_verde=0;
  }

  for (i=0;i<SIFNTSC_COLUMNS/FACTOR_MUESTREO;i++) {
    if ((i!=0) && ((i%(SIFNTSC_COLUMNS/(NUM_COLUMNAS*FACTOR_MUESTREO))) == 0)) { // salto en esa fila!
      zona ++; // zona siguiente
      miZona = zona;
    } else zona = miZona;

    for(j=0;j<SIFNTSC_ROWS/FACTOR_MUESTREO;j++) {
      if ((j!=0) && ((j%(SIFNTSC_ROWS/(NUM_FILAS*FACTOR_MUESTREO))) == 0)) { // salto en esa columna!
        zona ++; // zona siguiente
      }
      pos = (j*(SIFNTSC_COLUMNS/FACTOR_MUESTREO)+i)*4;

      if (((unsigned char)obs_teorica_filtrada[pos+0]==0)
        &&((unsigned char)obs_teorica_filtrada[pos+1]==0)
        &&((unsigned char)obs_teorica_filtrada[pos+2]==255)) { // rojo
          colores_encontrados_teorica[zona].num_pixel_rojo ++;
      } else if (((unsigned char)obs_teorica_filtrada[pos+0]==0)
        &&((unsigned char)obs_teorica_filtrada[pos+1]==255)
        &&((unsigned char)obs_teorica_filtrada[pos+2]==0)) { // verde
          colores_encontrados_teorica[zona].num_pixel_verde ++;
      } else if (((unsigned char)obs_teorica_filtrada[pos+0]==255)
        &&((unsigned char)obs_teorica_filtrada[pos+1]==255)
        &&((unsigned char)obs_teorica_filtrada[pos+2]==0)) { // azul
          colores_encontrados_teorica[zona].num_pixel_azul ++;
      } else if (((unsigned char)obs_teorica_filtrada[pos+0]==255)
        &&((unsigned char)obs_teorica_filtrada[pos+1]==0)
        &&((unsigned char)obs_teorica_filtrada[pos+2]==128)) { // morado
          colores_encontrados_teorica[zona].num_pixel_morado ++;
      }
    }
  }
```

Figura 5.13: Obtención de una imagen teórica resumida de 4x3 píxeles. Cuenta de píxeles de colores.

Una vez sabemos los píxeles de cada color en cada zona, podemos determinar el color dominante y los subdominantes de cada una de las 12 zonas. Esta diferenciación entre dominante y subdominante nos servirá de ayuda en el proceso de hallar la distancia (o parecido) entre una imagen real resumida y una imagen teórica resumida.

```
for (zona = 0; zona < NUM_COLUMNAS*NUM_FILAS; zona ++) { // establecemos el color dominante
  if ((colores_encontrados_teorica[zona].num_pixel_rojo >= umbralColorZona)
    && (colores_encontrados_teorica[zona].num_pixel_rojo >= colores_encontrados_teorica[zona].num_pixel_azul)
    && (colores_encontrados_teorica[zona].num_pixel_rojo >= colores_encontrados_teorica[zona].num_pixel_verde)
    && (colores_encontrados_teorica[zona].num_pixel_rojo >= colores_encontrados_teorica[zona].num_pixel_morado))
      colores_encontrados_teorica[zona].color_dominante = 0; // domina el Rojo
  else if ((colores_encontrados_teorica[zona].num_pixel_verde >= umbralColorZona)
    && (colores_encontrados_teorica[zona].num_pixel_verde >= colores_encontrados_teorica[zona].num_pixel_azul)
    && (colores_encontrados_teorica[zona].num_pixel_verde >= colores_encontrados_teorica[zona].num_pixel_rojo)
    && (colores_encontrados_teorica[zona].num_pixel_verde >= colores_encontrados_teorica[zona].num_pixel_morado))
      colores_encontrados_teorica[zona].color_dominante = 1; // domina el Verde
  else if ((colores_encontrados_teorica[zona].num_pixel_azul >= umbralColorZona)
    && (colores_encontrados_teorica[zona].num_pixel_azul >= colores_encontrados_teorica[zona].num_pixel_rojo)
    && (colores_encontrados_teorica[zona].num_pixel_azul >= colores_encontrados_teorica[zona].num_pixel_verde)
    && (colores_encontrados_teorica[zona].num_pixel_azul >= colores_encontrados_teorica[zona].num_pixel_morado))
      colores_encontrados_teorica[zona].color_dominante = 2; // domina el Azul
  else if ((colores_encontrados_teorica[zona].num_pixel_morado >= umbralColorZona)
    && (colores_encontrados_teorica[zona].num_pixel_morado >= colores_encontrados_teorica[zona].num_pixel_azul)
    && (colores_encontrados_teorica[zona].num_pixel_morado >= colores_encontrados_teorica[zona].num_pixel_verde)
    && (colores_encontrados_teorica[zona].num_pixel_morado >= colores_encontrados_teorica[zona].num_pixel_rojo))
      colores_encontrados_teorica[zona].color_dominante = 3; // domina el Morado
  else colores_encontrados_teorica[zona].color_dominante=-1; // no hay nadie "dominante"
 }
}
```

Figura 5.14: Obtención de una imagen teórica resumida de 4x3 píxeles. Determinación de color dominante.

5.4.2. Medida de distancia. Cálculo de probabilidad

Una vez obtenida la imagen teórica resumida por cada posición, se compara con la imagen real resumida obtenida anteriormente para obtener la probabilidad en cada posición. Esta comparación se realiza píxel a píxel (de los 4x3). Se establece así una relación de distancia entre las imágenes y se calcula la probabilidad a través de la siguiente función:

$$puntuacion_{dominante} = \frac{\sum_{i=0}^{pixeles_{resumida}} zonasiguales}{12} \tag{5.2}$$

$$puntuacion_{subdominante} = \frac{\sum_{i=0}^{pixeles_{resumida}} zonasiguales}{36} \tag{5.3}$$

$$probabilidad = \frac{(k\, puntuacion_{dominante}) + ((1-k)\, puntuacion_{subdominante})}{2} \tag{5.4}$$

Donde la ecuación 5.2 refleja el cálculo de la puntuación aportada por los colores dominantes de cada zona a la distancia entre imágenes. Se trata de una suma de zonas iguales, considerando como zonas iguales aquéllas que tienen el mismo color dominante tanto en la imagen teórica como en la real. Como máximo son 12 zonas las iguales, de ahí que normalicemos dividiendo entre 12; el resultado será un valor entre 0 y 1.

En la segunda ecuación (5.3), se aplica el mismo criterio, pero considerando como zonas iguales, aquéllas que tienen el mismo color subdominante tanto en la imagen teórica como en la real, con un determinado porcentaje de similitud. Cada zona podrá aportar 3 puntos, siempre que en esa zona los tres colores subdominantes superen un cierto umbral en ambas imágenes. Por tanto como máximo el resultado podrá ser 36, o 3 subdominantes por cada zona; de ahí que para normalizar el resultado de esta

ecuación se divida por 36. Nuevamente el resultado será un valor entre 0 y 1.

Observando la ecuación 5.4, la probabilidad se calcula teniendo en cuenta la aportación de los colores dominantes y los subdominantes, modulando un factor de ponderancia (k). Experimentalmente se ha deducido este valor a 0.3; dando por tanto más valor a la aportación de los colores subdominantes. Esto es así puesto que es muy difícil que el color dominante de una imagen sea tan representativo como para anular la información de los otros colores; normalmente un resumen que tenga en cuenta todos los colores aporta una información más concisa de la imagen que se está resumiendo. Finalmente, el resultado se divide entre 2 para normalizar el resultado entre 0 y 1, ya que estamos tratando con probabilidades.

5.5. Modelo de movimiento

Este modelo es el que captura la información de posición que proporcionan los sensores de odometría del robot y que se usan para medir desplazamientos y giros relativos. En la práctica, esto se consigue desplazando la nube de partículas con un movimiento homólogo al medido para el robot por los sensores de odometría.

Generalmente, en los algoritmos de localización se realiza un planteamiento clásico del modelo de movimiento donde la probabilidad viene determinada por el estado anterior y las órdenes que se mandan a los actuadores del robot. Sin embargo, nuestro modelo de movimiento no ordena a los actuadores si no que toma observaciones de los *encoders* del robot en cada momento para poder estimar los desplazamientos que el robot realiza en cada iteración del algoritmo. Por este motivo, nuestro localizador es capaz de funcionar en paralelo con cualquier programa que sea capaz de mover el robot.

La incorporación del movimiento consiste en calcular el desplazamiento que ha realizado el robot desde la última incorporación y aplicar ese desplazamiento a las partículas. La incorporación del desplazamiento es directa: las partículas se desplazan y rotan de la misma manera que lo hace el robot. Éste es un modelo aleatorio, al cual aplicamos también un ruido gaussiano, evitando así que una partícula repetida genere partículas en la siguiente población exactamente en la misma posición.

El desplazamiento del robot se calcula a través de las siguientes ecuaciones:

$$\Delta_r = Mov_{obs} + Ruido_{lineal} \tag{5.5}$$

$$\Delta_\theta = Giro_{obs} + Ruido_{angular} \tag{5.6}$$

$$Mov_{obs} = \sqrt{(x_t - x_{t-1})^2 + (y_t - y_{t-1})^2} \tag{5.7}$$

$$Giro_{obs} = \theta_t - \theta_{t-1} \tag{5.8}$$

$$Ruido_{lineal} = gaussiana(0, \sigma_r) \tag{5.9}$$

$$Ruido_{angular} = gaussiana(0, \sigma_\theta) \tag{5.10}$$

Y a cada partícula se le aplica el desplazamiento en pequeños incrementos para ganar suavidad en el movimiento de la población. Esto se consigue añadiendo independencia entre observaciones. Es muy importante recalcar que cada partícula se desplaza en la dirección que indica su orientación ya que cada partícula representa la posición y la orientación de dónde el robot puede estar.

La secuencia es muy importante: se calcula el desplazamiento lineal y se aplica para después incorporar el desplazamiento angular. En la figura 5.15 tenemos en rojo dos partículas en posiciones $(x_1, y_1, theta_1)$ y $(x_2, y_2, theta_2)$ respectivamente en $(t\text{-}1)$. Una vez calculado el desplazamiento del robot, en primer lugar la partícula se desplaza r en la orientación que tenía en $(t\text{-}1)$. En segundo lugar se realiza el desplazamiento angular sumando *theta* a la orientación en $(t\text{-}1)$. Las nuevas posiciones en (t) serán $(x_1, y_1, theta_1)$ y $(x_2, y_2, theta_2)$, que se marcan en color azul.

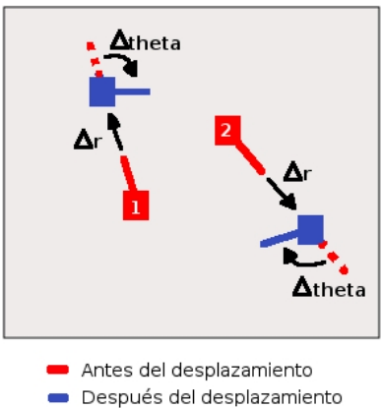


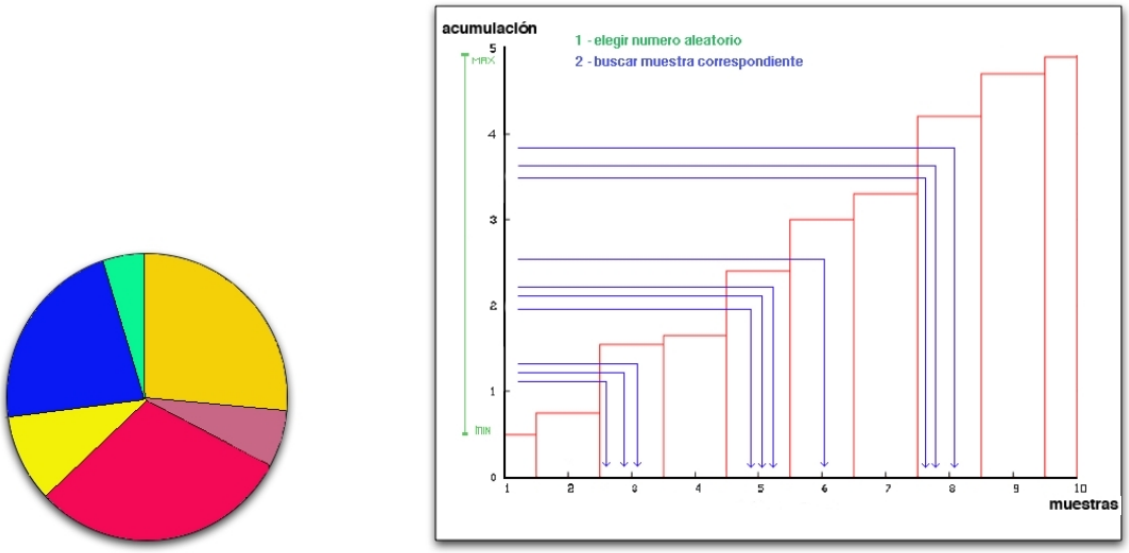
Figura 5.15: Ejemplo de desplazamiento de partículas.

5.6. Remuestreo

Para que la localización sea correcta con el filtro de partículas es necesario que la nube de partículas vaya convergiendo a las posiciones más probables. Para conseguir esto es necesario, a parte del modelo de observación y el de movimiento, el paso de *remuestreo*. Este paso es el que aporta la inteligencia a nuestro algoritmo generando poblaciones de partículas cada vez más concentradas alrededor de las posiciones compatibles.

La idea del remuestreo es la de generar nuevas poblaciones a partir de poblaciones anteriores tal que las partículas que acumulan mayor probabilidad tengan más opciones de propagarse a la siguiente población. Esto se implementa con el *Algoritmo de la Ruleta*. La implementación se ha realizado mediante el muestreo aleatorio y uniforme en el eje Y de la acumulación de probabilidad de las partículas, seleccionando como partícula para la siguiente generación la correspondiente en el eje X (figura 5.16). Una misma partícula puede generar muchos hijos iguales en la población siguiente. En este

caso, es la aplicación del modelo de movimiento con ruido gaussiano lo que hace que esos hijos se separen un poco unos de otros y exploren más zonas del espacio.



(a) Ruleta (b) Acumulación de probabilidad

Figura 5.16: Proceso de remuestreo siguiendo el Algoritmo de la Ruleta.

A nivel de código se tienen dos estructuras de tipo array que almacenan la población actual de partículas y la acumulación de esa población respectivamente. Este último array (ordenado) de acumulación almacena en cada posición, junto con las partículas que componen la población actual, la suma de todas las probabilidades de las posiciones anteriores (eje Y de la figura 5.16).

En el paso de remuestreo, al realizar el muestreo aleatorio sobre el eje de acumulación, se realiza una búsqueda binaria (ver figura 5.17) en el array ordenado de acumulación para ver qué partícula debe propagarse a la siguiente población. Estas partículas se almacenan de nuevo en el array de partículas actual representante de la nueva población.

```
int binary_search (float number) {
  int li,ls,middle,found;

  li=0;
  ls=NUM_PARTICLES;
  middle=(li+ls)/2;
  found=0;

  while (!found) {
    if ((number<=accumulation[middle].accumulation) &&
    (number>=accumulation[middle].low)) found=1;
    else if (number>accumulation[middle].accumulation) li=middle;
    else ls=middle;
    middle=(ls+li)/2;
  }

  return middle;
}
```

Figura 5.17: Código que implementa la búsqueda binaria del array de partículas.

Inicialmente la población de partículas se distribuye de manera aleatoria uniforme dentro del espacio (figura 5.23), asignando equiprobabilidad a todas las partículas. A medida que el algoritmo va iterando, se van realizando los pasos de observación y remuestreo. Las observaciones influyen en la probabilidad de las partículas. En cada paso de remuestreo (figura 5.24) se va generando una nueva población de partículas cada vez más concentrada alrededor de las posiciones que son más compatibles con la imagen.

En el caso de que la distribución de probabilidad de toda la población sea muy baja, se dice que la población ha degenerado. En tal caso se volvería, como inicialmente, a distribuir aleatoriamente una población de partículas sobre el espacio.

Se tiene continuamente una estimación de posición que mejora con cada iteración. Esta estimación se realiza o bien a través de la media aritmética de las posiciones de las partículas, o bien a través de la posición de la partícula con probabilidad más alta. Para matizar la validez de esta estimación utilizamos la desviación típica que presenta la población de partículas.

5.7. Interfaz gráfico

Aquí expondremos los resultados a nivel gráfico, bajo el visualizador desarrollado con la librería *OpenGL*. Recreamos el entorno en el que se han realizado los experimentos, el Laboratorio de Robótica. Para poder navegar en el entorno virtual, tenemos una cámara virtual que se controla con el ratón.

La parte *GUI* dentro del esquema *visualLocalization* (*visualLocalization_gui*) nos ha servido para depurar y observar la evolución de las partículas, a partir de la visualización de sus estructuras internas y la modificación de los parámetros de funcionamiento.

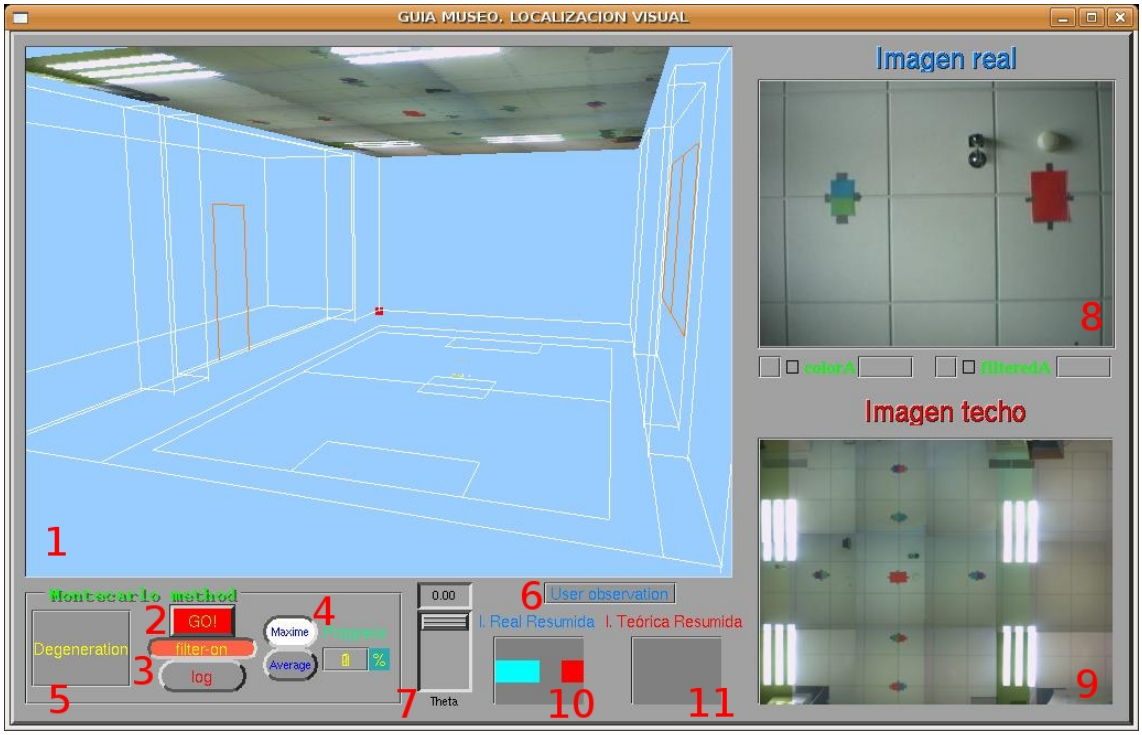


Figura 5.18: Aspecto del interfaz gráfico de la aplicación de localización.

Entre las partes más importantes de este interfaz gráfico se encuentran las siguientes:

1. Escenario virtual, donde podemos interactuar con el entorno a nuestra voluntad, con el ratón. Aquí visualizaremos todo el comportamiento de las partículas.

2. Botón *GO!* que activa el pintado de la población de partículas.

3. Botón *Filter on* que pone en marcha el algoritmo de filtro de partículas.

4. Botón *average* y *maxime* para expresar la localización como la media aritmética de la posición de las partículas y también como la posición de la partícula con mayor probabilidad.

5. Botón informativo *Degeneration*, situado en la parte inferior izquierda, indica que el sistema ha detectado degeneración en la población de partículas.

6. Botón *User observation*, para habilitar la posibilidad de que el usuario pinche en el entorno visual con el botón derecho y comprobar así qué se vería desde esa posición. Permite generar la imagen sintética y su resumen en la posición *(x, y)* que indique el usuario humano con el ratón.

7. Dial *Theta*, está ligado con el objeto gráfico anterior. Permite al usuario rotar el punto de observación desde la posición que ha seleccionado con el ratón. Fija el *theta* para completar el punto *(x, y, theta)* desde el que el usuario humano quiere ver lo que vería la cámara.

8. *Ventana Imagen Real*, nos ofrece las imágenes reales que está captando el robot en todo momento.

9. *Ventana Imagen Techo*, donde pondemos observar la *foto-composición* del techo, bajo el cual está navegando el robot.

10. *Ventana I. Real Resumida*, donde obtenemos la imagen real filtrada y resumida.

11. *Ventana I. Teórica Resumida*, donde obtenemos la imagen teórica filtrada y resumida.

5.8. Experimentos

Se han realizado varios experimentos para validar, caracterizar e intentar sacar el mayor rendimiento y eficacia del algoritmo desarrollado, así como para comprender mejor el funcionamiento del mismo. En nuestro caso hemos estudiado, entre otras cosas, el número de partículas de la población, el ruido gaussiano aplicado al movimiento de las partículas o la función de distancia.

En la figura 5.19 apreciamos el techo real, y dónde están situadas las balizas que analizaremos en los experimentos.

Figura 5.19: Techo real usado en los experimentos.

Las pruebas realizadas a este algoritmo han sido muy variopintas. Inicialmente se han establecido 1000 partículas para la población. Este número se ha ido variando a lo largo de los experimentos para establecer un número de partículas adecuado para el correcto funcionamiento del algoritmo. Los valores de la desviación típica que utilizamos

para aplicar el ruido gaussiano en el modelo de movimiento también se han ido variando a lo largo de los experimentos. Es necesario establecer una relación entre el número de partículas y el ruido gaussiano para obtener buenos resultados.

5.8.1. Estudio del modelo de observación

En un primer experimento hemos analizado el comportamiento de nuestro modelo de observación para ver si el cálculo de las probabilidades es razonable. Así lanzamos aleatoriamente 500 partículas, con su correspondiente posición *(x, y)*; y para cada una de ellas probaremos 36 ángulos diferentes, partiendo de 0 a incrementos de 10. De este modo, exploramos sistemáticamente todo el espacio de búsqueda *(x, y, theta)*.

Posteriormente sacamos gráficas de forma que en primer lugar sacaremos las gráficas empleando para cada partícula una *theta* buena (0 grados), una *theta* aproximada (10 grados) y una *theta* no aproximada (20 grados); finalmente, para cada posición calcularemos la mejor de las 36 la *thetas*, esto es, la que tenga mayor probabilidad de cada posición *(X, Y)*.

Figura 5.20: Situación del robot y observación dada.

Ahora vemos las gráficas vertidas por la aplicación *GnuPlot*, según lo explicado anteriormente.

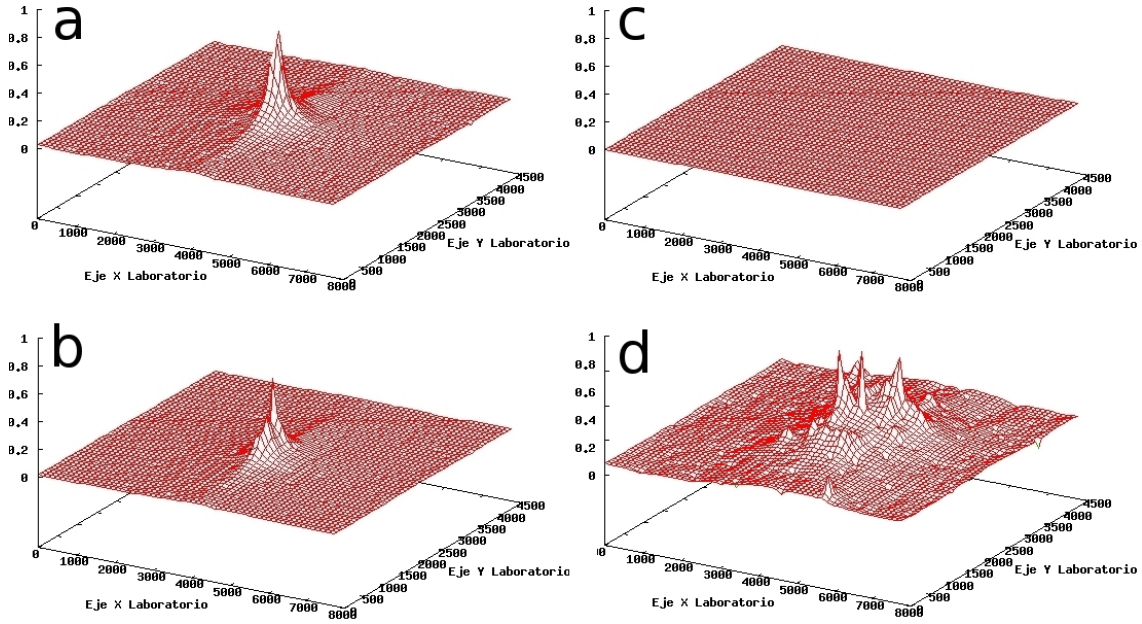


Figura 5.21: Probabilidad de las partículas según su *theta*. (a) *Theta* 0; (b) *Theta* 10; (c) *Theta* 20; y (d) Partículas con la probabilidad más alta de entre 36 *thetas* diferentes.

El robot real está fijado en la posición *(X = 4120mm, Y = 1750mm, theta = 0º)*. En la gráfica 5.21-a vemos cómo quedándonos con aquellas partículas con *theta* = 0, la probabilidad en la zona estimada es elevada mientras que en el resto del espacio la probabilidad es nula.

En la gráfica 5.21-b nos hemos quedado con las partículas que tienen un *theta* aproximado al real (*theta* = 10 grados). Así, la probabilidad de las partículas que están en la zona estimada es sensiblemente menor.

Por último, considerando las partículas con un *theta* alejado del real (*theta* = 20 grados), obtenemos una gráfica (figura 5.21-c) donde no hay ninguna zona estimada. Esto significa que no existe ninguna partícula que tenga una cierta probabilidad en ningún punto del espacio de muestreo, con ese valor de *theta*.

Con todo lo anterior, podemos concluir que nuestro modelo de observación funciona razonablemente bien: *(i)* ofreciendo una alta probabilidad en la zona estimada, considerando una orientación igual al real; y *(ii)* a medida que esa orientación de las partículas se va distanciando de la que tiene realmente el robot, la zona estimada va desapareciendo.

Para ver resumidamente la probabilidad en todo el espacio *(x, y, theta)*, no sólo en *(x, y)* hemos realizado un experimento donde probamos para cada posición 36 orientaciones diferentes. El resultado es el mostrado en la figura 5.21-d, donde vemos que aparecen en este caso ciertos picos de ruido alrededor de la zona estimada. Esto picos se pueden deber a simetrías. Podemos concluir diciendo que la orientación de las partículas sí resulta decisivo en la probabilidad de una misma partícula, como era de suponer.

5.8.2. Ejecución típica con robot parado

Una vez visto que el modelo de observación es correcto, veremos si efectivamente las partículas convergen siguiendo nuestro algoritmo en la zona donde está el robot. En la figura 5.22 vemos dónde estaba situado el robot *(X = 4400mm, Y = 3800mm, theta = 180º)* cuando tomó la instantánea que vamos a analizar en este experimento. Podremos así comparar al final de este apartado si la posición estimada por el algoritmo se puede considerar válida.

Figura 5.22: Posición real del robot en el experimento actual.

Repartimos 1000 partículas de forma aleatoria, y con el valor de *theta* también aleatorio. La situación inicial se refleja en la figura 5.23.

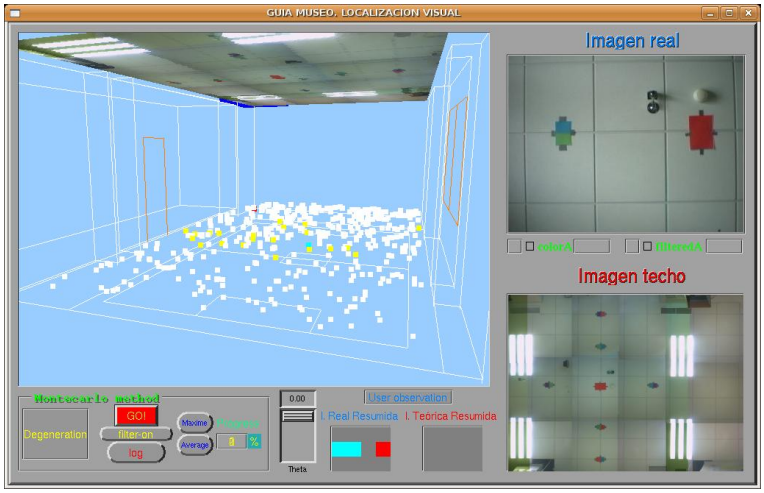


Figura 5.23: Ejecución del filtro de partículas. Distribución inicial aleatoria.

Tras filtrar las partículas, en un paso intermedio, las partículas se acumulan en tres zonas probables.

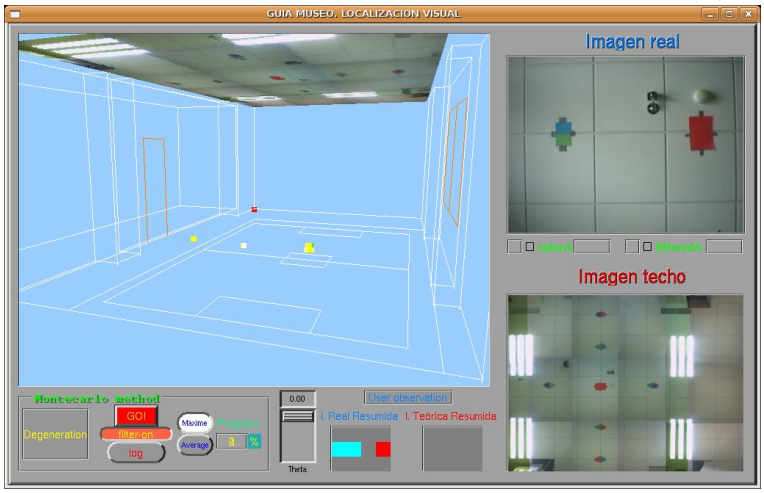


Figura 5.24: Ejecución del filtro de partículas. Acumulación en zonas probables.

Finalmente el algoritmo acaba convergiendo en la zona con mayor probabilidad *(X = 4350mm, Y = 3850mm, theta = 180º)* que es muy cercana a la posición que realmente es.

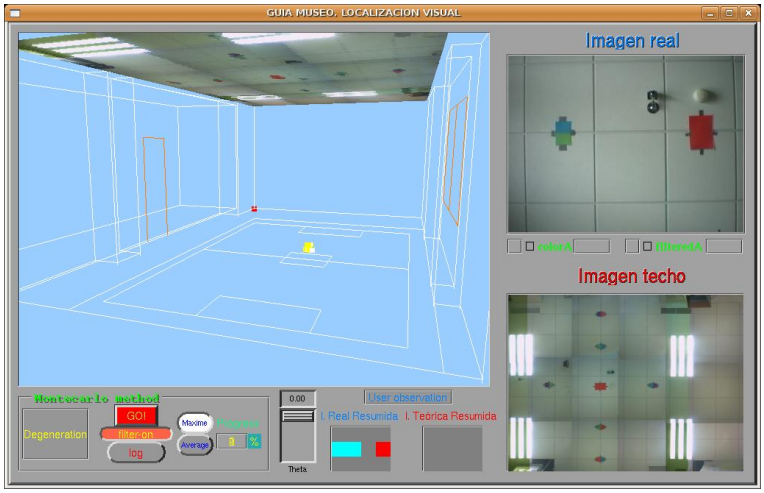


Figura 5.25: Ejecución del filtro de partículas. Localización final.

5.8.3. Precisión

En total hemos probado con 9 posiciones distintas dentro del Laboratorio de Robótica (figura 5.26). A continuación mostramos una tabla (5.27) con tales posiciones reales, así como las estimadas por el algoritmo. Podremos así sacar conclusiones sobre la precisión de nuestro algoritmo.

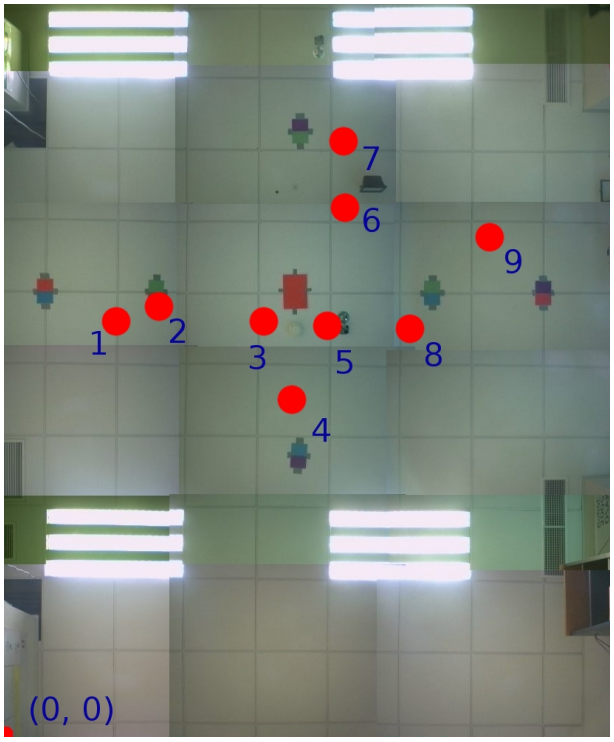


Figura 5.26: Mapa visual del Laboratorio de Robótica. Posiciones reales experimentadas.

Punto	Posición real (X,Y)	Posición estimada (X,Y)
1	[3950, 550]	[3900, 555]
2	[4200, 650]	[4100, 620]
3	[4120, 1750]	[4100, 1730]
4	[3250, 2250]	[3100, 2200]
5	[4000, 2270]	[4120, 2290]
6	[4720, 2420]	[4550, 2500]
7	[5500, 2450]	[5430, 2350]
8	[3950, 2900]	[4010, 2970]
9	[4400, 3800]	[4350, 3850]

Figura 5.27: Posiciones reales frente a las posiciones estimadas.

En vistas a la tabla mostrada en la figura 5.27 podemos concluir diciendo que nuestro algoritmo tiene una precisión centimétrica, con un error de ±2cm en el peor de los casos. Las pruebas se han realizado fijando el valor de *theta* a 0º para el robot real. Las respectivas posiciones arrojadas por nuestro algoritmo están comprendidas en un margen de error de ±5º. Este margen de error experimentado, tanto en posición como en orientación, podemos considerarlo admisible para la autolocalización del robot.

5.8.4. Ajustes del sistema. Ruido gaussiano

El algoritmo tiene una serie de parámetros configurables. Así, por ejemplo, en este apartado veremos el efecto de variar el radio de dispersión de ruido gaussiano $(\sigma_r, \sigma_\theta)$ habiendo modificado asimismo el número de partículas que conforman la población.

El hecho de aplicar un ruido gaussiano en el modelo de movimiento nos servía para dar holgura a las posiciones de las partículas. En este experimento hemos partido de la posición y observación *(X = 3950mm, Y = 55mm, theta = 0º)* que ilustramos en la figura 5.28. Hemos fijado el ruido en *theta* a 0.5º, y probaremos modificando el ruido en el radio.

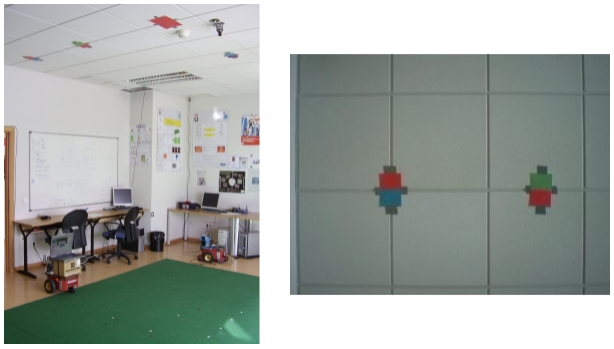


Figura 5.28: Situación del robot y observación dada.

Lanzamos las partículas de forma aleatoria y filtramos, siguiendo el procedimiento anterior. En este caso emplearemos la mitad de partículas, 500 en total. El resultado se refleja en los fotogramas siguientes. Hemos probado inicialmente un radio de disperción de ruido de 30 mmm alrededor de cada partícula, lo cual es algo razonable. Mostramos diferentes etapas:

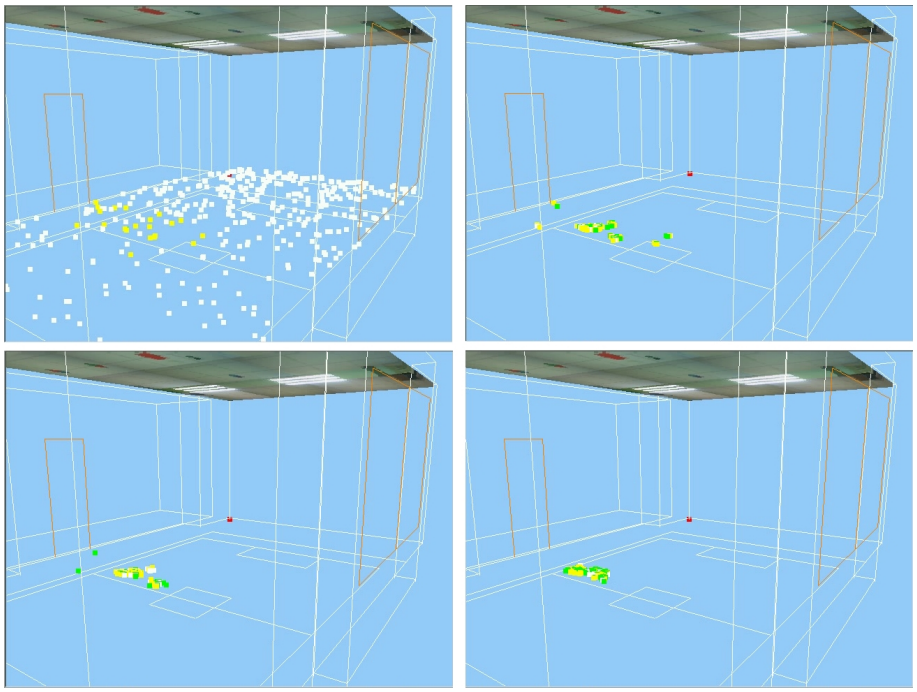


Figura 5.29: Evolución de las partículas con ruido gaussiano de 30 mm.

Ahora, repetimos el experimento, pero con un ruido gaussiano insignificante. Con tan sólo 5 mm alrededor de cada partícula, lo cual no es un valor muy apropiado para este tipo de algoritmos. En cualquier caso veámos qué ocurre y después razonamos los distintos comportamientos.

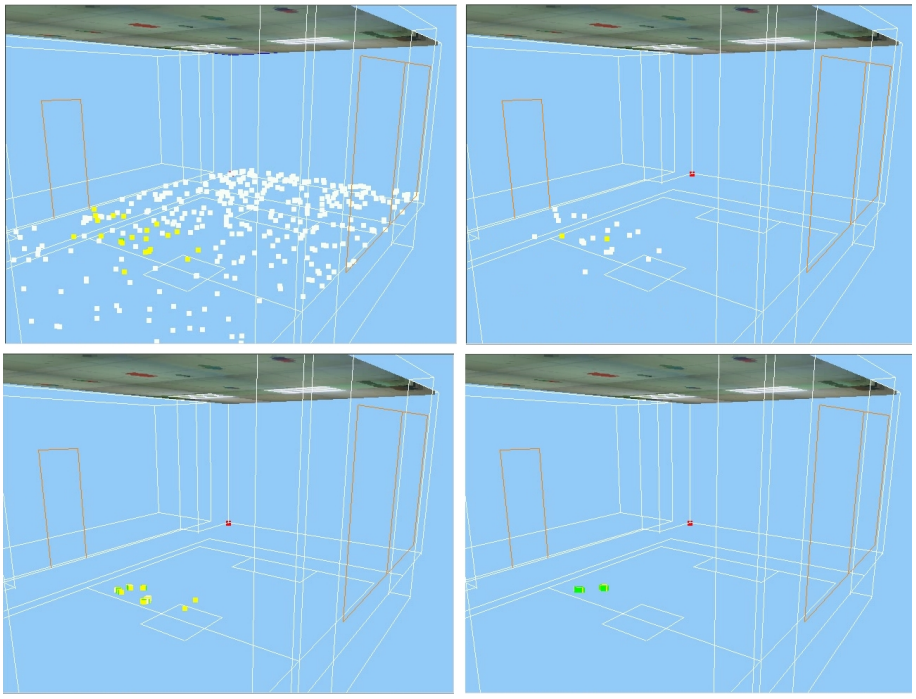


Figura 5.30: Evolución de las partículas con ruido gaussiano de 5 mm.

Mientras que en el primer experimento la convergencia (posición estimada: $X = 3900mm$, $Y = 555mm$, $theta = 2^o$) tardó apenas 8 segundos, en el segundo caso fueron más de 30, y finalmente no converge a una única nube de puntos, sino que se queda *titubeando* entre dos. Por otro lado, hemos visto que con 500 partículas el algoritmo funciona perfectamente, en el entorno que estamos trabajando. Si extrapolamos la solución a un entorno más amplio, el número de partículas debería ser incrementado para cubrir correctamente todas las zonas posibles.

Como conclusión final de este experimento, vemos que el ruido gaussiano es importante para la convergencia de las partículas. No obstante, hay que modular convenientemente este valor para no producir resultados no deseables; por ejemplo, un valor extremadamente elevado supondría zonas muy extensas de convergencia, con lo que la posición de mayor probabilidad dentro de esa zona estimada podría no ser la correcta. En nuestro caso, un valor de 30mm se considera razonable para la autolocalización del robot en el entorno experimentado.

Capítulo 6
Navegación local

Estoy en movimiento y no moviéndome del todo. Yo soy como la Luna encima de las olas que siempre rueda y se balancea. No es un «yo estoy haciendo esto», sino más bien una comprensión interna de que «esto me esta sucediendo a través de mí» o «está haciendo esto para mí». La conciencia de sí mismo es el mayor obstáculo para la ejecución correcta de toda acción física.

Bruce Lee, *El Tao del Jeet Kune Do*

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* (vista en el capítulo 4) en que no necesita información exacta de la posición global, y 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 éstos no aparecen en el mapa dado como entrada para el algoritmo de navegación global.

Tanto la navegación local necesita de la navegación global, como ésta de la primera. La navegación local necesita que le vayan actualizando el destino local, para que no pierda el rumbo; idealmente no necesita saber su posición, aunque como veremos en la práctica con el robot real, esto no es rigurosamente exacto. Por su parte, la navegación global vemos que necesita de la navegación local para evitar chocar con obstáculos imprevistos.

A lo largo de la historia de la Robótica se han ido desarrollando varios métodos destinados a evitar colisionar con obstáculos, desde los primeros sencillos algoritmos que se basaban en detener al robot cuando se encontrara cerca de un obstáculo, hasta otros más sofisticados que nos permiten sortearlos. A parte de evitar los obstáculos, conviene que la navegación local mantenga la dirección hacia el objetivo, y también que sea rápida, que se mueva ágilmente y de forma fluida.

6.1. Esquema de navegación local

El esquema *localNavigation* se encarga de implementar el algoritmo de navegación local, que pilotará al robot de manera que evite obstáculos cercanos. Hemos implementado, analizado y experimentado tres técnicas distintas de navegación local. En un primer lugar empleamos el algoritmo de *Gradient Path Planning*, el mismo utilizado en su forma original para solucionar la navegación global (ver capítulo 4). Posteriormente probamos con otra técnica conocida como *VFF* ([Borenstein, 1989]); y, por último, incorporamos una mejora propia sobre este método, incluyendo una *ventana de seguridad* circundante al robot.

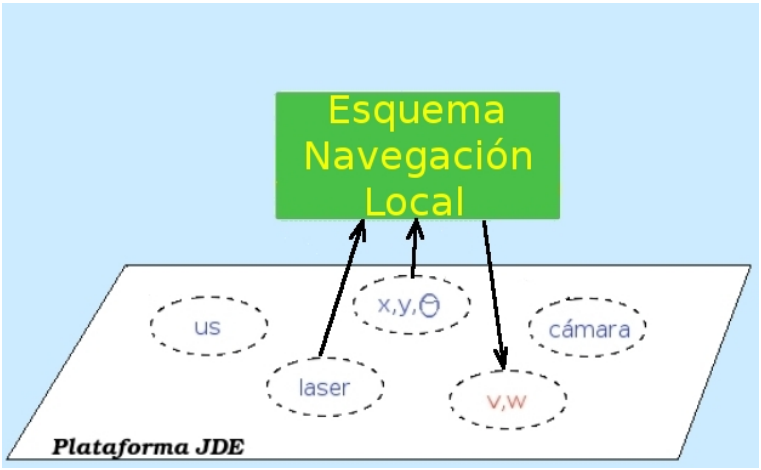


Figura 6.1: Diagrama de E/S del esquema de navegación local.

Todos estos algoritmos recogen información del entorno desde el sensor láser. Utilizando una rejilla local de ocupación en la que vuelcan las lecturas del láser, para almacenar posición de obstáculos y espacios libres para la navegación. Asimismo los algoritmos necesitan la odometría para saber si el robot ha alcanzado el destino y para situar en 2D la información del láser.

La salida del algoritmo son órdenes de velocidad lineal y angular, gobernando el movimiento *aquí y ahora* del robot.

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. Esta distancia se ha establecido empíricamente en 1 metro. Si no hay obstáculos en ese radio alrededor del robot, éste sería guiado mediante el mecanismo de navegación global.

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.

6.2. Información del espacio en rejilla local

Ya vimos cómo se producía la expansión del gradiente (4.4). En este caso las dimensiones de nuestro espacio de configuración serán bastante más reducidas, en tanto en cuanto estamos trabajando en un espacio *local*. Así, en nuestro caso, consideraremos unas dimensiones de 4m. x 4m., lo que será suficiente para tener un conocimiento apropiado del entorno inmediato que rodea al robot.

Para que un robot se mueva de forma coherente, la información que tiene del medio que le rodea debe ser lo más fidedigna posible. Podremos hacer distinción entre objetos estructurales (ventanas, puertas), objetos estáticos (sillas, mesas, estanterías), y objetos dinámicos (personas, carros, cajas).

Los sensores láser de un robot pueden dar información de los obstáculos que tiene inmediatamente delante de él hasta una distancia de 8 m., pero no puede ver objetos que estén siendo tapados por otros. Por tanto, tendremos que tener siempre presente que el conocimiento que un robot pueda tener de su entorno es limitado.

Mejor que tener *láser (t)*, podemos crearnos un mundo a partir de la información que nos proporcionan los sensores en cada instante. Así, tenemos la información que alcanza a darnos el láser, y además la memoria en la que vamos acumulando estos datos. Si los sensores de odometría son ideales y no introducen *ruido* sobre la posición del robot en el mundo; y si además suponemos un mundo estático, esta solución resulta perfecta. Sin embargo, en la realidad no ocurren ninguna de las dos condiciones.

Para nuestro propósito, nos hemos creado una rejilla de información de 4m. de ancho y alto (centrada en el robot), donde cada celda cuadrada tiene unas dimensiones de 5cm. de lado. Cada lectura láser será incorporada en su celda correspondiente, que llevará incorporado un error relativo a las dimensiones de las celdas empleadas. Para actualizar tal rejilla seguiremos estos pasos:

- Añadir las nuevas lecturas a la rejilla, según la posición del robot en *t*.
- Si el punto de la rejilla en *t* está en la misma trayectoria que alguno de los puntos de la rejilla en *t-1*, y además está más cercano al robot, no borramos el anterior.

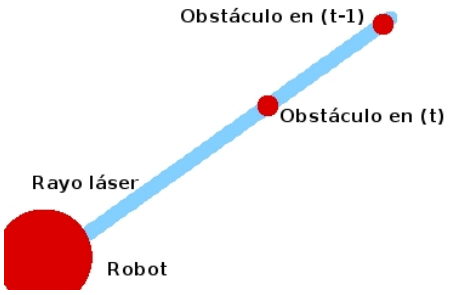


Figura 6.2: No borramos los puntos antiguos más lejanos. Acumulación de información.

- Si el punto de la rejilla en *t* está en la misma trayectoria que alguno de los puntos de la rejilla en *t-1*, pero está más alejado respecto al robot, borramos el anterior.

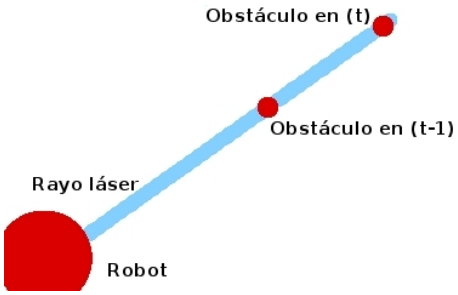


Figura 6.3: Borramos los puntos antiguos más cercanos. Evitar estelas *fantasmas*.

Dado que se trata de una rejilla solidaria al robot es, por tanto, una rejilla móvil. En principio debería moverse a medida que se mueve con el robot, pero esto es muy costoso computacionalmente, ya que hay que refrescarla totalmente en cada iteración. Tampoco podemos refrescarla cada mucho tiempo porque el robot podría no percatarse de algunos obstáculos. Un valor de compromiso es establecer una ventana central solidaria a la rejilla, cuya dimensión es de 1mx1m; cuando el robot salga de tal ventana, actualizamos.

La actualización se realiza de forma que la zona solapada entre la rejilla en *(t-1)* y la rejilla en *(t)* se copia de una a otra, mientras que las nuevas celdillas se inicializan a vacío, hasta que el robot incorpore nueva información.

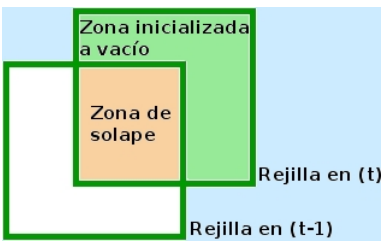


Figura 6.4: Solape de la rejilla móvil.

En la figura 6.5 vemos una secuencia de cómo se produce la actualización de la rejilla cuando aparecen y desaparecen obstáculos.

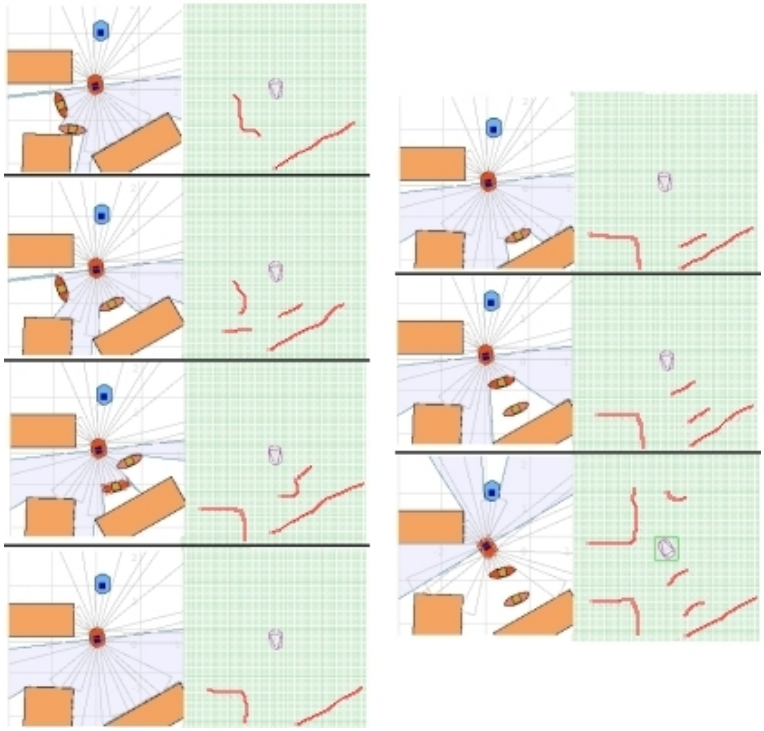


Figura 6.5: Actualización de la rejilla de información local.

Como podemos ver en el experimento en que el robot intenta salir por una puerta (6.3.3), el proceso de actualización de la rejilla tiene algunas disfunciones, puesto que puede quedar *desfasado* de la realidad, sobretodo en aquellas zonas que hace tiempo que no se observan con el láser.

Para paliar estas desavenencias, introducimos algunas mejoras como el *olvido temporal de memoria*. De esta forma, los puntos almacenados en memoria durante un tiempo superior a un determinado intervalo, serán eliminados de ésta para que no generen falsos comportamientos. Estamos suponiendo un ambiente dinámico, por lo que es muy probable que la situación de aquéllos puntos recordados desde hace mucho tiempo haya cambiado.

El código que implementa esta funcionalidad es el siguiente:

```
void updateGridTiming () {
  int j;

  for(j=0;j< ((occupancyGrid->size)*(occupancyGrid->size)); j++) {
    /* Recorremos la rejilla de información */
    if ((dameTiempo() - occupancyGrid->map[j].ultima_actualizacion) > MAX_TIME_ON_GRID) {
      /* Se trata de un punto antiguo */
      occupancyGrid->map[j].estado = EMPTY; /* lo marcamos a vacio para no me haga de frontera */
      occupancyGrid->map[j].ultima_actualizacion = dameTiempo ();
    }
  }
}
```

Figura 6.6: Refresco de puntos antiguos en la rejilla de información.

En la figura 6.7 se muestra el diagrama de flujo en ejecución de la construcción de memoria.

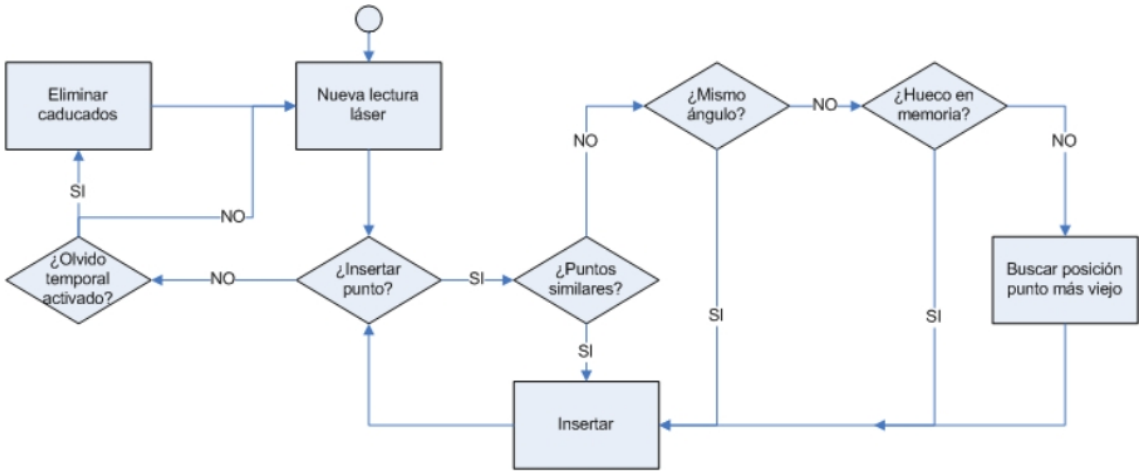


Figura 6.7: Diagrama de flujo de la construcción de la memoria de puntos.

6.3. Método *Gradient Path Planning*

Hemos optimizado la implementación realizada para la navegación global, introduciendo ciertas diferencias. Las diferencias fundamentales son: *(i)* el cálculo del gradiente se lanza no sobre el mapa global, como en el capítulo 4; *(ii)* ahora hay que tener en cuenta los obstáculos dinámicos; *(iii)* contamos con una rejilla de ocupación que se mueve siguiendo el movimiento del robot, y hay que recalcular el campo virtual repetidamente a medida que el robot se desplaza; y *(iv)* el destino es local y puede no caer dentro de la rejilla de ocupación. Asimismo hay que resaltar que, en esta ocasión, se consume una gran cantidad de tiempo tiempo en regenerar el gradiente, donde se suele emplear entre 3 y 15 seg. de media, dependiendo si hay obstáculos o no en la vecindad del robot, los cuales generarán (como ya vimos en su momento) otro gradiente por separado.

Es obvio el suponer que este método trabaja de forma más eficaz en ambientes estáticos. No obstante, se ha desarrollado una mejora para ir incorporando información proveniente de los láser de forma continua. Esto, añadido al propósito del *GPP*, garantiza una navegación segura, robusta e *inteligente*.

6.3.1. Destino intermedio

Una de las primeras diferencias con respecto al escenario global es que es posible que el destino se *escape* fuera de la rejilla local de información. En este caso optamos por establecer nosotros un destino intermedio entre el robot y el destino final, tal como demuestra la siguiente figura:

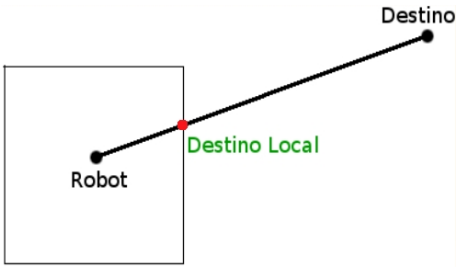


Figura 6.8: Cálculo del destino intermedio.

Hemos interpolado la imaginaria línea recta que une el robot con el destino final, y nos hemos quedado con la posición última que entra dentro de la rejilla de información. El código que realiza esta funcionalidad es el siguiente:

```
deltaX = myDestination->x - posRobot.x;
deltaY = myDestination->y - posRobot.y;

celdilla = xy2celdaRejilla (gradientGrid, posRobot);

while (celdilla != -1) { /* Recorremos el rayo hasta llegar al límite de nuestra rejilla */
  actual.x = posRobot.x + i*incremento*deltaX;
  actual.y = posRobot.y + i*incremento*deltaY;
  celdilla = xy2celdaRejilla (gradientGrid, actual);

  i ++;
} /* Cuando salgamos será porque celdilla ya pertenece al exterior de la rejilla */

/* Retornamos a una posicion anterior, que será la últimísima casilla dentro de rejilla */
actual.x -= incremento*deltaX;
actual.y -= incremento*deltaY;

myDestination->x = actual.x;
myDestination->y = actual.y;
```

Figura 6.9: Código relativo al cálculo del destino intermedio.

6.3.2. Novedades de ocupación

Otro problema es establecer un criterio de cada cuánto tiempo hemos de regenerar el campo virtual. Ya comentamos en el capítulo anterior (4.8.1) que esta operación es costosa computacionalmente, de ahí que haya que plantearse esta cuestión.

Lógicamente no podemos regenerar uno nuevo cada iteración de nuestro esquema de navegación local, ya que sería demasiado lento. Otra opción sería regenerarlo cuando lleguemos al *destino intermedio* comentado previamente; pero esta opción sería poco realista, puesto que en ese largo intervalo de tiempo podrían aparecer nuevos objetos dinámicos que no estaríamos considerando. Así llegamos a concebir un nuevo término al que llamaremos *novedades*, y que representará el número de puntos que corresponden a nuevos obstáculos detectados y a los nuevos huecos desde la última vez que se calculó el gradiente.

La consideración de puntos novedad será más oportuna en ambientes dinámicos, ya que las nuevas mediciones serán muy convenientes para generar un nuevo gradiente que implique nuevos caminos posibles hacia nuestro destino. Esta regeneración por novedades conlleva un cierto retardo en la navegación del robot, pero en cambio será mucho más segura. Un caso típico que refleja este problema lo podemos ver en la figura 6.10, donde vemos una situación en que el robot está mirando hacia el sur (1) y genera su gradiente para ir hacia donde le han indicado el destino, que en este caso es hacia el norte. Cuando el robot se da la vuelta (2), estamos ante un *nuevo mundo* que desconocíamos en el momento de generar el gradiente. Así, éste es inconsistente con la situación actual. Por lo que sería muy conveniente tener en cuenta las *novedades* producidas y generar un nuevo gradiente (3) y ya navegar con la información recopilada del entorno, que ahora sí resulta coherente (4).

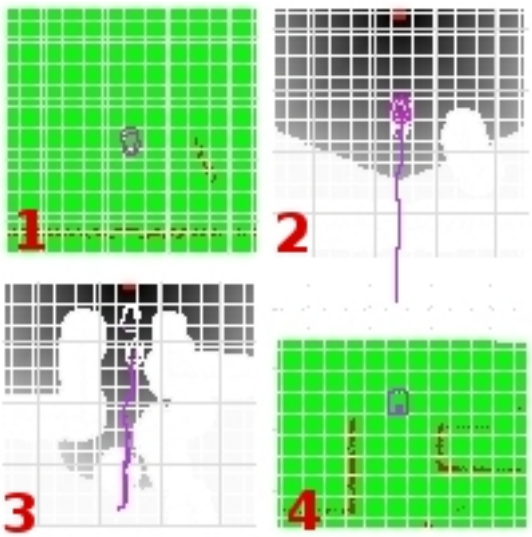


Figura 6.10: Sucesivas fases en la navegación con algoritmo GPP. Incorporación de novedades y generación de nuevo gradiente.

Si por el contrario nuestro entorno es estático, la navegación del robot será mucho más liviana y no será tan importante tener en cuenta esos puntos *novedad*, puesto que ya estaban todos *previstos* de antemano. Asimismo, el efecto de *recuerdo* de puntos muy antiguos será positivo en este caso, ya que nos ofrecerán información veraz.

Así, cada 100 ms. se añadirán nuevas lecturas procedentes de los lásers, aunque el gradiente no será calculado hasta que haya suficientes *novedades* como para considerarlas influyentes. De hecho, la calidad de nuestro algoritmo dependerá de cómo tratemos este factor. Dependiendo de la circunstancia en la que nos encontremos, tendremos necesidad de regenerar el gradiente con más o menos novedades; por ejemplo, si estamos en un ambiente estático, nos bastaría generar un único gradiente, mientras que si estamos en una sala con diversos objetos dinámicos, el gradiente deberá regenerarse con mayor frecuencia.

6.3.3. Experimentos

A continuación veremos una serie de experimentos que hemos realizado para validar los algoritmos, afinar algunos de sus parámetros y depurarlos bien. Los hemos llevado a cabo tanto en simulador como posteriormente sobre el robot real. Recreamos situaciones típicas y usuales que pueden darse en un entorno de interiores.

Entrar en una habitación

Podemos apreciar en las siguientes figuras el algoritmo en sus diferentes fases. El color de las celdas de nuestra rejilla con el campo virtual es proporcional al coste de los puntos en el espacio; a medida que el color es más oscuro, el coste es menor. Los puntos que pertenecen a obstáculos o sus alrededores más inmediatos tienen un coste muy elevado, por lo que su color es de un blanco intenso. Este *acolchonamiento* de los obstáculos nos garantiza en mayor medida que el robot no colisionará en ningún momento con los mismos.

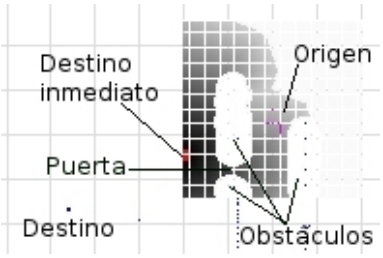


Figura 6.11: Situación inicial. Primer gradiente hasta destino inmediato.

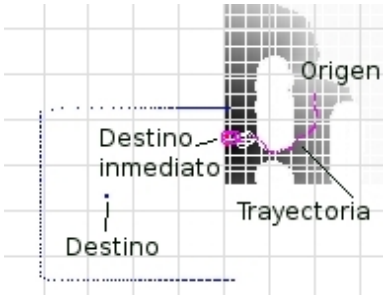


Figura 6.12: Situación intermedia. Llegada al destino inmediato.

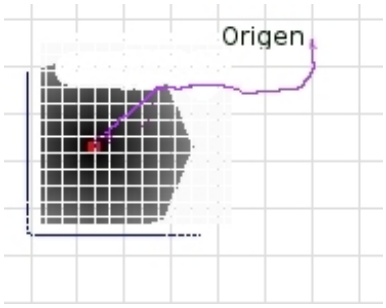


Figura 6.13: Cálculo de nuevo campo virtual y gradiente al alcanzar el primer destino inmediato. Llegada al destino final.

Comienza por la posición inicial del robot (figura 6.11), junto con las distintas posiciones seguidas a lo largo del gradiente. A la llegada al destino inmediato se genera un nuevo gradiente, y con ello las nuevas posiciones adoptadas por el robot a lo largo del nuevo gradiente (figura 6.12). La unión de los distintos gradientes generados se realiza por unión en línea recta entre origen y destino, estableciendo así el destino inmediato al que hacemos alusión. De este modo, podremos recorrer cualquier distancia, sin importar las dimensiones de nuestro gradiente. Pero como ya comentamos en el comienzo de este apartado, mientras más gradientes tengamos que generar, mayor será el coste de computación y, por tanto, mayor tiempo de navegación. En la imagen 6.13 podemos observar la trayectoria seguida por el robot; es el camino de mínimo coste.

Salir por una puerta

Hemos realizado una serie de experimentos para optimizar el método explicado en este capítulo. Y los resultados experimentales nos demuestran que el comportamiento del robot ante ambientes desconocidos es *torpe* y/o lento.

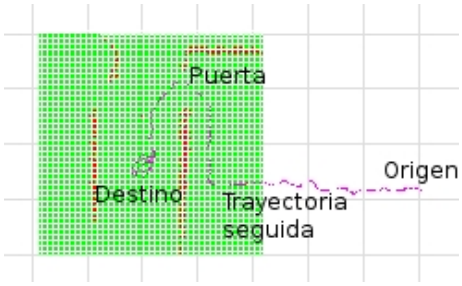


Figura 6.14: Navegación GPP. Actuación del robot al salir por una puerta.

Vemos cómo el campo de gradiente guía correctamente al robot, que en vez de ir por el camino recto hacia el destino (camino que minimiza la distancia euclídea), se dirige acertadamente hacia la puerta.

Experimentos con el robot real Pioneer

A continuación mostramos cinco situaciones distintas[1] en las que recreamos las situaciones comentadas previamente sobre simulador. El robot *Pioneer* parte de una posición inicial y, tras atravesar por el único hueco posible, llega al destino señalizado con una baliza.

Figura 6.15: Experimento 1.

[1]Vídeo disponible en http://jde.gsyc.es/index.php/jmvega_guide_robot

Figura 6.16: Experimento 2.

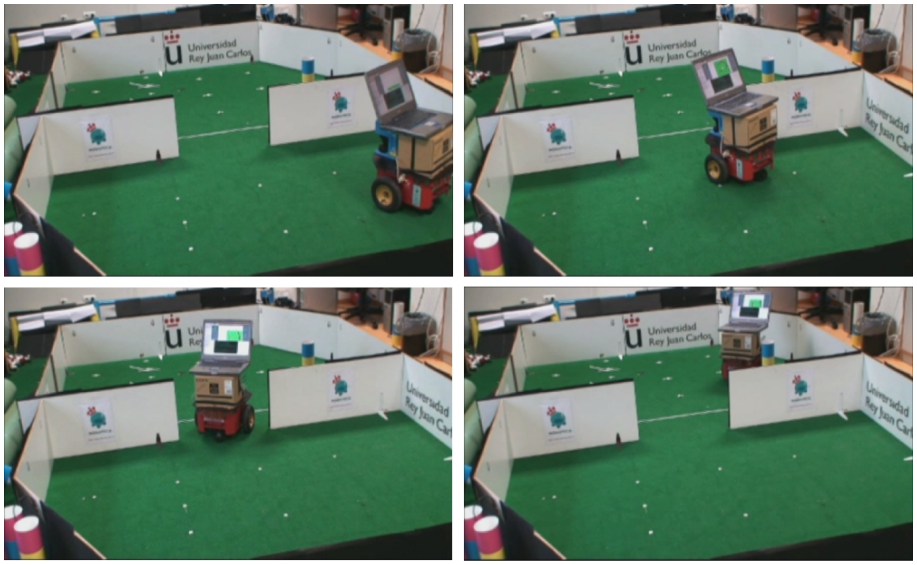


Figura 6.17: Experimento 3.

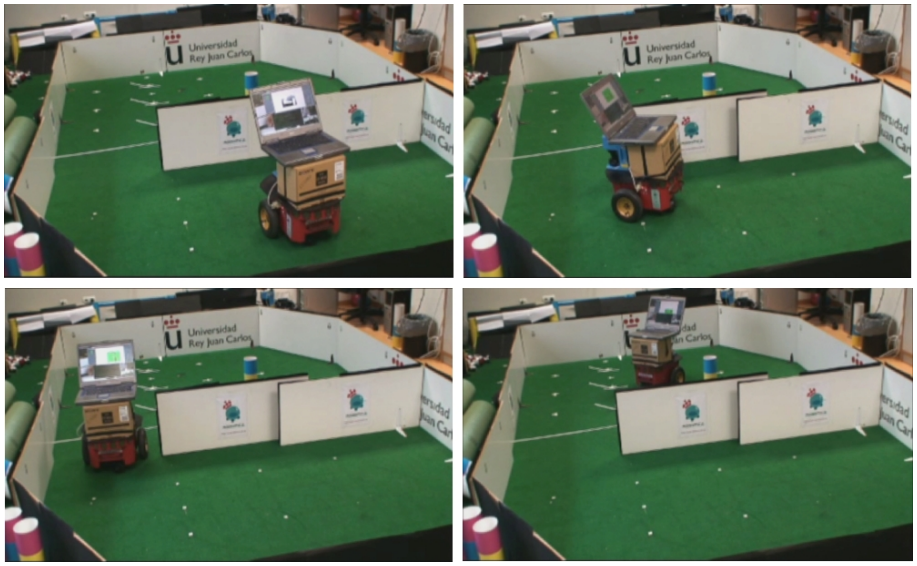


Figura 6.18: Experimento 4.

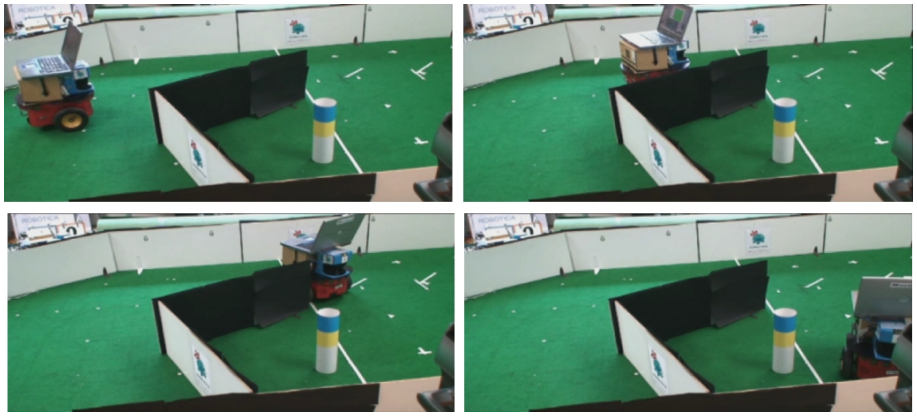


Figura 6.19: Experimento 5.

En todos los experimentos vemos que el robot llega satisfactoriamente hacia el destino cercano sorteando los obstáculos.

Un aspecto positivo de este algoritmo es que nunca se queda estancado el robot en ninguna posición, por la propia definición del algoritmo empleado; y es que cada punto del espacio virtual tiene un valor de campo asociado, que forma parte de un gradiente a través del cual navega el robot. Por contra, podemos decir que la *reactividad* que exige la navegación local no está bien resuelta por este mecanismo; en tiempo real es inabordable esta solución, porque el robot ha de pararse demasiado tiempo en calcular la *ruta* local a seguir.

Veremos a continuación otra técnica que soluciona este problema que es de primordial interés en la técnica de navegación local que elegiremos para aplicar a la

solución final de robot guía.

6.4. Método *Virtual Forces Field*

La segunda técnica utilizada para la navegación local es la de los campos de fuerzas virtuales *VFF* o *Virtual Forces Field*. En esta técnica el robot es atraído por la posición del objetivo mientras que los obstáculos ejercen una fuerza repulsiva que hace que se aleje de ellos.

Para desarrollar el algoritmo de navegación local nos basamos en el propuesto por Borenstein ([Borenstein, 1989]). Según éste, tenemos en todo momento un campo de fuerzas virtuales rodeando al robot, compuesto por fuerzas repulsivas (generadas por los obstáculos) y fuerzas atractivas (generadas por los destinos de navegación); resultando así una fuerza total que será la que empuje al robot hacia su objetivo.

La idea de tener obstáculos ejerciendo fuerza sobre un robot fue dada por Khatib [Khatib, 1984]. Después, tal concepto fue llevado más allá por otros investigadores como Krogh [Krogh, 1985] o Thorpe [Thorpe, 1985], quien aplicó un campo de potencial exterior al camino planificado. La combinación de tal método junto con una rejilla de ocupación para almacenar y representar los obstáculos es el denominado método *Virtual Forced Field*; de ahora en adelante, *VFF*.

6.4.1. Cálculo de fuerzas

Cada celda marcada como ocupada generará una fuerza *repulsiva* sobre el robot, empujándolo en sentido contrario a la dirección *robot-celda*. La magnitud o módulo de esta fuerza será inversamente proporcional al cuadrado de la distancia a la que se halle el robot de tal celda. Vemos si está demasiado cerca o no; en cuyos casos aplicaremos diferentes constantes de la siguiente forma (figura 6.20):

$$|\vec{F_{rep}}| = \frac{K}{d^2} \tag{6.1}$$

```
if ((distancia*distancia) < alcancePeligroso)
  fuerza = K2/(distancia*distancia); /* generamos MAS fuerza repulsiva */
else
  fuerza = K/(distancia*distancia); /* generamos MENOS fuerza repulsiva */
```

Figura 6.20: Cálculo de la fuerza repulsiva ejercida por un punto-obstáculo.

De este modo, aquellas celdas que estén extremadamente cerca del robot ejercen una fuerza de repulsión sobre éste mucho mayor. La fuerza repulsiva total se calcula como la suma vectorial de todas las fuerzas individuales ejercidas por todas las celdas:

$$\vec{F_{rep\,Tot}} = \sum_{i=0}^{n} \vec{F_{rep}}(i) \tag{6.2}$$

Figura 6.21: Concepto de VFF. Proximidad de obstáculo (pared) y acción de las fuerzas ejercidas por las celdas ocupadas.

Además de esta fuerza de repulsión, durante toda la navegación existe otra fuerza de *atracción* entre el robot y el objetivo (cuyas coordenadas -de ambos- son conocidas en todo momento), donde el módulo es constante y sus componentes vectoriales se calculan como sigue:

$$\vec{F_{at}} = K_3 \left(\frac{x_d - x_r}{d_{rob,dest}}, \frac{y_d - y_r}{d_{rob,dest}} \right) \tag{6.3}$$

Pues bien, la suma vectorial de todas las fuerzas; las repulsivas ejercidas por las celdas ocupadas sobre el robot, y la atractiva (entre el robot y el destino), produce una fuerza resultante cuya dirección será la que siga el robot en su movimiento y cuyo módulo marcará la velocidad desarrollada por el mismo (ver sección 6.4.2).

$$\vec{F_t} = a\vec{F_{rep}} + b\vec{F_{atr}} \tag{6.4}$$

De forma esquemática, podemos ver el cálculo de la fuerza resultante en el siguiente trozo de código extraído del esquema de navegacionLocal, donde a partir de los puntos obstáculos que tenemos en memoria calculamos una fuerza resultante final que dirigirá el movimiento del robot:

```
for (i=0; i<tamMemoria; i++) { /* Para cada punto-obstáculo de nuestra memoria */
  distancia=  (((m[i].x-myencoders[0])*(m[i].x-myencoders[0]))
            + ((m[i].y-myencoders[1])*(m[i].y-myencoders[1])));
  radianes=atan2((m[i].y-myencoders[1]),(m[i].x-myencoders[0]));
  angulo=(int) (radianes*RADTODEG);

  /* SUTILEZA PARA QUE EL ROBOT TENGA QUE GIRAR LO MENOS POSIBLE */
  if(ang_robot>=180) ang_robot=-360+ang_robot; /* =-(360-ang_robot)-->pasa a negativo */
  diff_ang=abs(angulo-ang_robot); /* calculamos el total que debería girar el robot */
  if (diff_ang>=180) diff_ang=360-diff_ang; /* mejor si gira por el camino más corto */

  /* Vemos si el punto está dentro del alcance en el que los puntos generan fuerza de repulsión */
  if (((diff_ang>45)&&(diff_ang<135)&&(distancia<alcanceLateral))||
    (((diff_ang<=45)||(diff_ang>=135))&&(distancia<alcance))) {
      if (((diff_ang>45)&&(diff_ang<135)&&(distancia<seguridadLateral))||
      ((diff_ang<=45)&&(distancia<seguridad)))
        (*res).avanzaEnGiro=FALSE; /* avanzamos sin girar */

      radianes=radianes+(180.0*DEGTORAD); /* sumo 180º ya que queremos el angulo opuesto */

      modulo=calculoFuerza(distancia,diff_ang); /* MODULO FUERZA REPULSIVA DEL PUNTO ACTUAL */

      /* Añadimos las componentes a la suma de x y de y */
      sumaX+=cos(radianes)*modulo;
      sumaY+=sin(radianes)*modulo;
  }
}

(*res).direccionFuerzaRepulsiva[0]=sumaX; /* FUERZA REPULSIVA TOTAL */
(*res).direccionFuerzaRepulsiva[1]=sumaY;

/* Calculo la DIRECCIÓN DE FUERZA RESULTANTE atribuyendo factores de proporción a la fuerza atractiva y repulsiva */
(*res).direccionFuerzaResultante[0]=  (alpha *(*res).direccionFuerzaAtractiva[0])
                                    + (beta*(*res).direccionFuerzaRepulsiva[0]);
(*res).direccionFuerzaResultante[1]=  (alpha *(*res).direccionFuerzaAtractiva[1])
                                    + (beta*(*res).direccionFuerzaRepulsiva[1]);

sumaX=(*res).direccionFuerzaResultante[0]; // variables temporales
sumaY=(*res).direccionFuerzaResultante[1];

(*res).moduloFuerzaResultante=sqrt((sumaX*sumaX)+(sumaY*sumaY)); /* MODULO FUERZA RESULTANTE */
(*res).anguloFuerzaResultante=atan2(sumaY,sumaX); /* ANGULO FUERZA RESULTANTE */
```

Figura 6.22: Cálculo de la fuerza resultante, en función de todos los puntos-obstáculo.

6.4.2. Control de velocidad

Los motores de un robot no entienden términos de fuerza, de ahí que haya que traducir esos valores calculados de fuerza a órdenes de velocidad lineal (v) y velocidad angular (w). Lo ideal sería que cuando encontraramos un obstáculo en nuestro camino redujéramos paulatinamente la velocidad, en función de lo cerca que estuviera éste, y se mantuviera esta *velocidad de emergencia* hasta que hayamos rodeado el obstáculo y tengamos nuevamente el rumbo hacia el objetivo. Sin embargo, en la realidad no ocurre así, ya que hemos de considerar factores como la inercia que lleva el robot. Además, unido a esta inercia sucede que cuando el robot se percata de la presencia de un obstáculo cercano, es fuertemente *expulsado* hacia el lado contrario; cuando ya está relativamente alejado, las fuerzas de atracción le llevan nuevamente a irse *contra* el obstáculo. Así, el resultado es un movimiento oscilatorio, como vemos en la figura 6.23, fruto de un experimento que hemos llevado a cabo en el simulador.

Figura 6.23: Oscilaciones con VFF ante la proximidad de un obstáculo.

Hemos de imprimir una gran fuerza repulsiva cuando el robot está cerca de un obstáculo, y disminuir ésta mientras lo está sorteando. En nuestro caso, matemáticamente lo que hacemos es aplicar unos factores de proporción diferentes a las fuerzas atractivas ($a = 0.7$) y repulsivas ($b = 0.4$), a la hora de calcular la fuerza resultante. Este equilibrio entre repulsión y atracción modula lo precavido u osado que es el robot, pero no resuelve las oscilaciones.

Intuitivamente, la velocidad que llevaría el robot con el método del *VFF* sería proporcional al módulo de la fuerza resultante. Sin embargo, el mejor modo de dar solidez y eficacia a la navegación es proporcionar los comandos de velocidad (tracción y rotación, o v y w) según la diferencia entre el ángulo actual del robot y la orientación del objetivo respecto a este último. Asimismo, es primordial comprobar si hay algún obstáculo imprevisto que nos fuerce girar rápidamente. De modo que resultan una serie de reglas ad-hoc que citamos a continuación.

```
difAngle= (resultante.anguloFuerzaResultante) - (resultante.anguloQueLlevo);

if(peligro) { /* LO MAS URGENTE, SI HAY PELIGRO INMINENTE, GIRAMOS LENTAMENTE */
  if(difAngle<0) {*myv=0.0; *myw=wNegMax/2;}
  else {*myv=0.0; *myw=wPosMax/2;}
} else if (abs(difAngle)<limiteTrayectoriaSimilar) {*myv=vMax; *myw=0.0;} /* VIENTO EN POPA */
else if (abs(difAngle)>limiteTrayectoriaDesigual) { /* GIRAMOS RAPIDAMENTE */
  if (difAngle<0) {*myv=0.0; *myw=wNegMax;}
  else {*myv=0.0; *myw=wPosMax;}
} else if (difAngle<0) {*myv=vMax/2; *myw=wNegMax/2;} /* MITAD, MITAD */
else {*myv=vMax/2; *myw=wPosMax/2;}
```

Figura 6.24: Reglas para comandar la velocidad del robot, según la situación.

Dicho de forma esquemática, para un instante t:

$$(v, w)(t) = (0, w_{max}) \quad \text{si} \quad peligro\ o\ trayectoria\ diferente\ a\ la\ actual \tag{6.5}$$

$$(v, w)(t) = (v_{max}, 0) \quad \text{si} \quad trayectoria\ similar\ a\ la\ actual \tag{6.6}$$

$$(v, w)(t) = (v_{max}/2, w_{max}/2) \quad \text{si} \quad otro\ caso \tag{6.7}$$

De este modo, si no hay obstáculos en nuestro camino, el robot irá a su máxima velocidad. Pero si no podemos continuar en línea recta, disminuiremos la velocidad de tracción y aumentaremos la de rotación según la situación.

6.4.3. Experimentos

En este apartado resaltamos las ventajas y los inconvenientes de este método respecto al anterior visto previamente. Además plantearemos los problemas que no quedan resueltos con ninguna de las dos técnicas.

Ventajas sobre métodos convencionales

En cuanto a detección de bordes, la precisión milimétrica que ofrece el láser permite al robot generar un contorno de objetos muy coherente con el real; lo que permite obtener información muy precisa del entorno que rodea al mismo. Evitando así posibles lecturas erróneas que nos podrían generar *obstáculos fantasmas*, típicos de otros sensores como los ultrasonidos, que complicarían el cálculo de la fuerza adecuada.

Este método, además, no precisa de mucho tiempo computacional para reaccionar. Así, no es necesario detener al robot para procesar y evaluar la información dada por los sensores, por lo que el movimiento de sortear obstáculos es fluido y rápido.

Problemas

Un problema propio de VFF es el hecho de que el robot pueda caer en situaciones de las que no pueda salir. Típicamente se produce cuando el robot se encuentra en el interior de obstáculos que tienen forma de U. También es posible que se quede *atrapado* cuando debe salir por la puerta de una habitación, ya que no será consciente de ello y se quedará permanentemente *cabeceando* de cara a la pared más próxima en línea recta hacia el objetivo. Tal situación se conoce como el problema de *mínimos locales* y lo podemos apreciar en la figura 6.25.

Figura 6.25: Mínimo local. Cabeceo con VFF ante una esquina.

Del mismo modo, y a consecuencia de las fuerzas repulsivas, el hecho de salir por una puerta no resulta tan fácil como podríamos pensar a priori. Dadas las dimensiones del Pioneer y la anchura de una puerta corriente, disponemos de una holgura de unos 15 cm. aprox. a cada lado del robot. Esto no es suficiente empleando el método del VFF, ya que la fuerza repulsiva ejercida por ambos extremos del marco de la puerta será extremadamente superior a la atractiva, lo que implicará un nuevo cabeceo permanente en el frontal de la puerta por parte del robot.

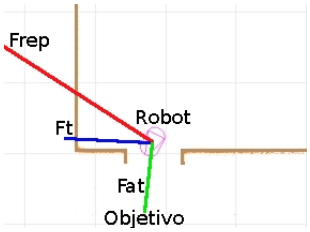


Figura 6.26: Cabeceo con VFF al salir de una puerta.

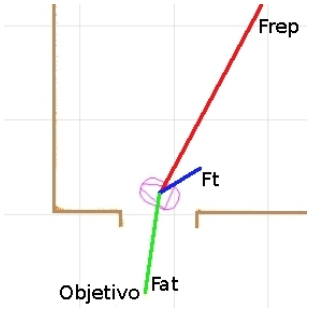


Figura 6.27: Retroceso o repulsión al salir de una puerta.

Modificando los parámetros de proporción aplicados a la fuerza repulsiva y atractiva, explicados en la sección anterior (6.4.2), esta situación ya estaría solucionada. No obstante volveríamos a caer en el problema que citábamos en la misma sección, y es que ante obstáculos cercanos es conveniente otorgar de forma exhaustiva prioridad a la fuerza repulsiva; para contrarrestar los efectos de la inercia que pueda llevar el robot en un momento determinado.

En conclusión, podríamos decir que este método nos ha resuelto el problema de reactividad en tiempo real que teníamos con el de *Gradient Path Planning* visto en la sección anterior (6.3). Así, este algoritmo resulta muy fiable en cuanto a sortear obstáculos que puedan aparecer de modo imprevisto, ya que el robot tiene una gran

vivacidad y rapidez de respuesta.

Por otro lado nos hemos encontrado con problemas que teníamos resuelto con el método anterior, como el de los *mínimos locales*. Sin embargo, este problema no es importante, puesto que está resuelto por el mecanismo de navegación global que vimos en el capítulo 4. La principal dificultad que tenemos con la técnica expuesta aquí es el mero hecho de salir por sitios relativamente estrechos, como son las puertas. En segundo lugar, las oscilaciones del movimiento resultante al esquivar un obstáculo no son tampoco deseables.

Con motivo de satisfacer estos dos problemas inherentes a la técnica del *VFF* hemos hecho uso de una técnica que va más allá e incorpora algunas mejoras sobre éste. A continuación presentamos el mecanismo que se ha seguido.

6.5. VFF con Ventana de seguridad

A lo largo de la sección anterior hemos visto algunas deficiencias del método *VFF*. Por una parte veíamos cómo se producían oscilaciones al sortear un obstáculo, que daban como resultado un movimiento poco uniforme. En otro caso hemos apreciado ciertas carencias en situaciones tan usuales como salir por una puerta.

Nuevamente haremos uso del método *VFF* implementado tal y como hemos comentado en la sección anterior (6.4). Por tanto, haremos el mismo uso de la rejilla de ocupación. Sin embargo, desactivaremos el efecto de las fuerzas virtuales cuando *notemos* que estamos en un *sitio estrecho*; consecuentemente, la forma de obtener los comandos de velocidad varía notablemente.

Así, hemos introducido una notable mejora incorporando una ventana de seguridad solidaria al robot, que nos irá advirtiendo en todo momento del entorno más próximo que rodea al robot. Diferenciaremos cuatro partes en dicha ventana (como muestra la figura 6.28).

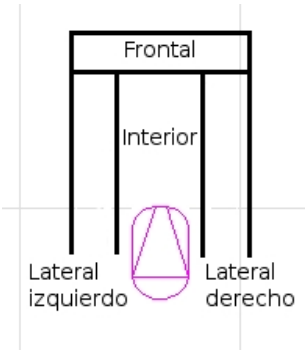


Figura 6.28: Descripción de la ventana de seguridad.

6.5.1. Control basado en casos

El comportamiento del robot cambiará según la información que le proporcione esta ventana de seguridad. En tanto que si se cumplen las condiciones:

1. No hay ningún obstáculo en el *interior*.
2. No hay obstáculo delante, en el *frontal*.
3. Hay obstáculo en alguno de los *laterales*, o en los dos.

entonces los comandos de velocidad que enviaremos al robot serán:

$$v(t) = v_{max} \tag{6.8}$$

$$w(t) = 0 \tag{6.9}$$

Si no estamos en este caso entonces seguimos los mismos cálculos que el *VFF* descrito en la sección 6.4.

```
if (sitioEstrecho ()) { // COMPORTAMIENTO VENTANA
  *myv = vMax;
  *myw = 0.0; /* Seguir recto por el sitio estrecho */
} else { // COMPORTAMIENTO VFF
  // Reglas AD-HOC vistas para VFF
}
```

Figura 6.29: Comandos de velocidad en situación de sitio estrecho.

6.5.2. Experimentos

A continuación veremos cómo los problemas que nos surgían con el método anteriormente experimentado se resuelven con la mejora de la ventana de seguridad incorporada.

Movimiento no oscilatorio

Aplicando el método descrito, podremos detectar, por ejemplo, cuándo estamos sorteando un obstáculo. Recordemos las oscilaciones de la figura 6.23. La misma situación, incorporando esta ventana de seguridad, daría como resultado el movimiento que vemos en la siguiente figura. Es evidente la mejoría en el movimiento, ahora mucho más dinámico y preciso.

Figura 6.30: Comportamiento con ventana de seguridad ante la proximidad de un obstáculo.

Salir por una puerta

Otro problema que nos habíamos encontrado era el salir de una puerta. Mientras que con *VFF* las fuerzas de repulsión nos obligaban a retroceder, con la ventana de seguridad detectaremos que nos encontramos ante un hueco estrecho y tornaremos a ejecutar un movimiento rectilíneo.

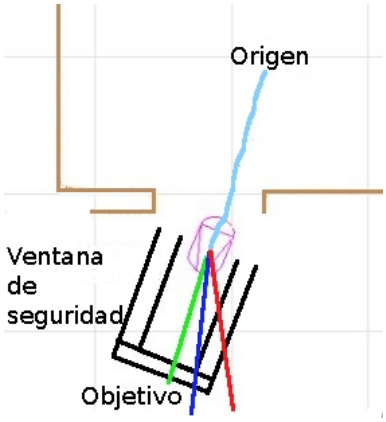


Figura 6.31: Comportamiento con ventana de seguridad ante una puerta.

Experimentos con el robot real Pioneer

A continuación mostramos cuatro situaciones distintas[2] en las que recreamos las situaciones comentadas previamente sobre simulador. El robot *Pioneer* parte de una posición inicial y, tras sortear varios obstáculos, llega al destino señalizado con una baliza.

[2]Vídeo disponible en http://jde.gsyc.es/index.php/jmvega_guide_robot

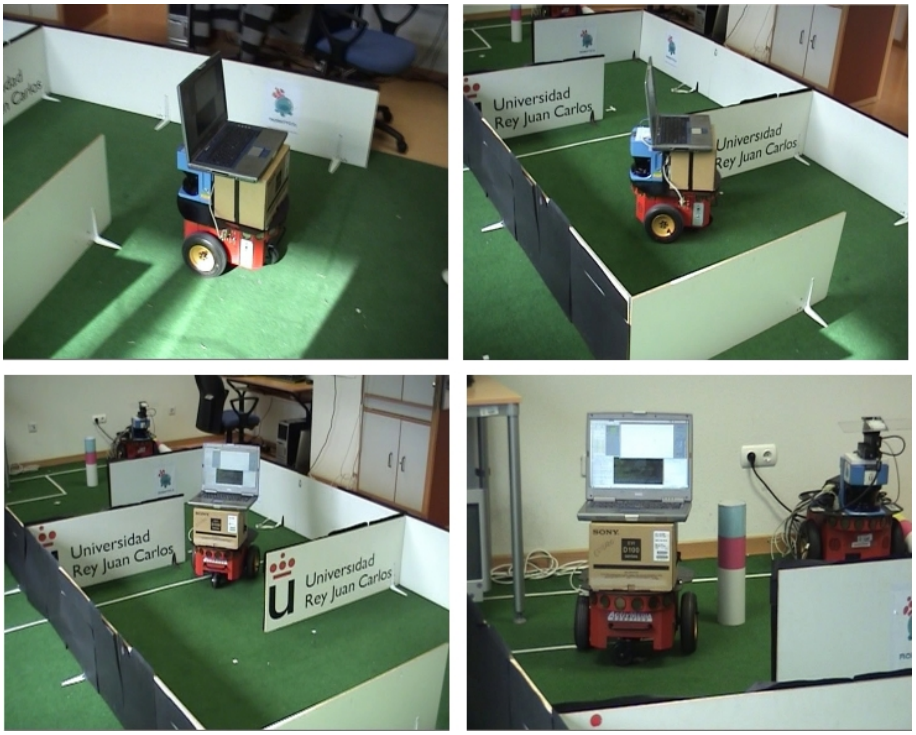


Figura 6.32: Experimento 1.

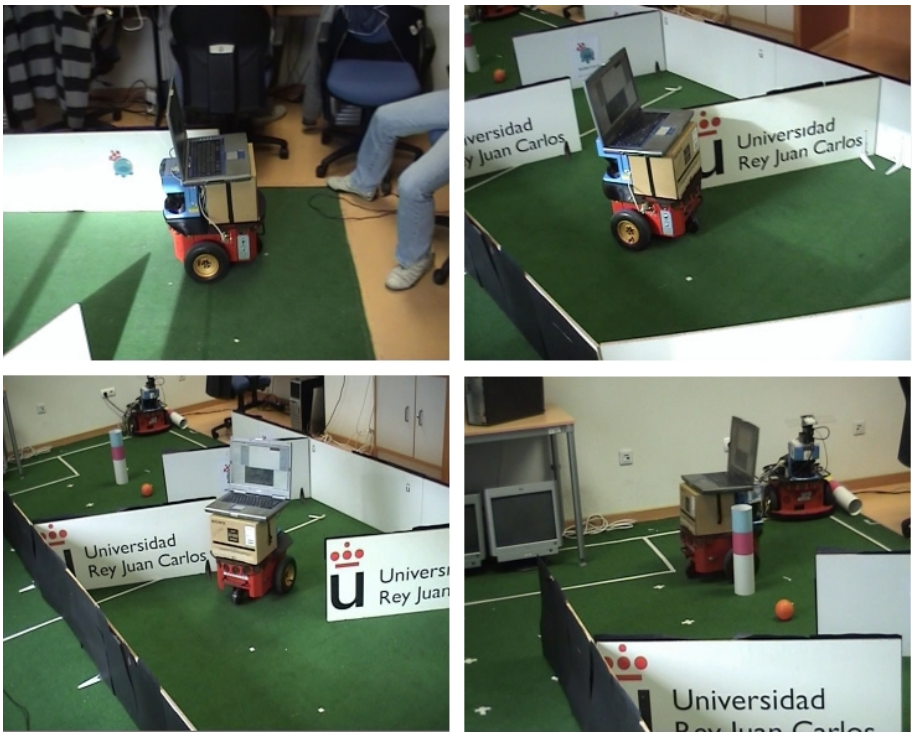


Figura 6.33: Experimento 2.

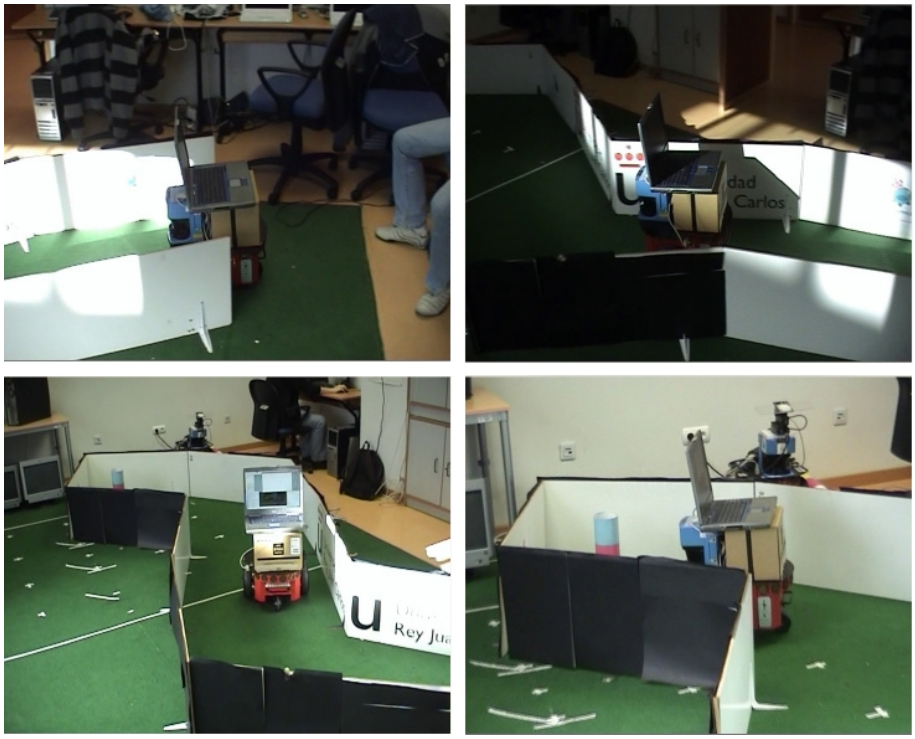


Figura 6.34: Experimento 3.

Figura 6.35: Experimento 4.

Este método es idóneo para una navegación local reactiva en tiempo real. Incorpora lo mejor del método *VFF*; mientras que en los casos en que éste no ofrecía solución, obtenemos movimientos muy fluidos, naturales (sin oscilaciones) y rápidos.

En este capítulo hemos visto cómo en un intento por experimentar el método de *Gradient Path Planning* en navegación local, no es aceptable en tiempo real, dado su elevado coste computacional y, por tanto, su gran retardo temporal. Así, pasamos a implementar y probar con el método *VFF*, que vimos que nos resolvía satisfactoriamente esta problemática, aunque nos presentaba problemas como las *oscilaciones* y sobretodo la dificultad de navegar por sitios *estrechos*. Tales problemas han sido resueltos con la ventana de seguridad; por lo que finalmente podemos concluir que hemos llegado a una solución factible validada sobre el robot real.

Capítulo 7
Conclusiones

¡Figúrate que en la Tierra sólo viviesen plantas y animales! Entonces no habría habido nadie capaz de distinguir entre «gatos» y «perros», «lirios» y «frambuesas».

Jostein Gaarder, *El mundo de Sofía*

En los capítulos anteriores hemos descrito los problemas abordados en este proyecto y las distintas soluciones propuestas junto con un serie de experimentos que las validan. En este capítulo expondremos las conclusiones obtenidas y las posibles líneas de trabajo en el futuro, cerrando así la presente memoria.

7.1. Conclusiones

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

- *Desarrollar un algoritmo de navegación global*. Para ello hemos implementado el algoritmo de navegación usando el método de *Gradient Path Planning* ([Konolige, 2000]). Como se comentó en el capítulo 4 de esta memoria, se ha programado un esquema que expande un campo virtual desde el punto objetivo deseado hasta la posición del robot y hace navegar a éste siguiendo el gradiente de este campo escalar, por el camino más corto hasta el destino y manteniéndose alejado de los obstáculos.

- *Desarrollar un algoritmo de localización*. Se ha diseñado e implementado un algoritmo de localización basado en filtro de partículas, denominado *visualLocalization*. Este algoritmo utiliza las imágenes de la cámara que el robot lleva con un modelo de observación novedoso que compara la imagen real con la imagen teórica desde cada una de las partículas. Igualmente se han realizado varios experimentos con el robot real de tal algoritmo, a través de la interfaz gráfica diseñada de forma que los resultados sean vistosos e informativos. Tal implementación se ha tratado en el capítulo 5, donde profundizábamos en el filtro de partículas con el Método de MonteCarlo.

- *Desarrollar un algoritmo de navegación local*. Para ello se han implementado, analizado y experimentado tres técnicas distintas, detalladas en el capítulo 6.

Desde un primer algoritmo basándonos en una versión a nivel local del algorimo de *GPP*, hasta un último método que construímos optimizando la técnica de fuerzas virtuales o *VFF* ([Borenstein, 1989]). El conjunto de las tres técnicas implementadas resulta en un esquema general de navegación local que se encarga de llevar al robot lejos de los obstáculos a la par que avanzar hacia el destino cercano. Las tres técnicas se han probado y ajustado sobre simulador y sobre el robot real.

Estos objetivos tenían implícitos una serie de requisitos que condicionaban la solución que debíamos obtener. Veamos cómo se han satisfecho estos requisitos y en qué medida:

- *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 íntegramente en *C* y es una colección de esquemas, que se ejecutan bajo *JDE*.

- *La solución final ha de funcionar sobre el robot real.* Era uno de los requisitos fundamentales, ya que supone un avance importante respecto a otros proyectos anteriores realizados en el Grupo de Robótica. Los algoritmos han tenido que ser continuamente mejorados y optimizados de cara a funcionar en condiciones reales, más exigentes que las de los simuladores, donde todo resulta perfecto. En concreto, se han probado de forma robusta sobre robot real los algoritmos concernientes a la navegación local, tanto con el método de *GPP* como el de *VFF*, y el algoritmo de localización.

  La navegación global no se ha probado sobre robot real ya que la localización no resulta del todo robusta con el robot en movimiento, mientras que con el robot parado, este último mecanismo funciona satisfactoriamente.

- *El comportamiento del robot ha de ser vivaz.* Ya que el funcionamiento del robot se produce en un entorno real, resulta imprescindible que éste responda en tiempo real a las situaciones que se le puedan plantear; ha de ser rápido en sus movimientos. Asimismo, aparte de moverse rápidamente, los movimientos del robot han de ser fluidos; deben ir concatenados unos con otros, evitando así la sensación de discontinuidad. Este requisito puede darse por cumplido, después de ver los vídeos del robot real *Pioneer*, así como los fotogramas mostrados en el capítulo de navegación local (6).

- *Navegación segura.* Por último, otro requisito no menos importante que los anteriores, es la seguridad que ofrece el robot durante su navegación. Dado que el entorno donde se mueve el robot es un edificio de oficinas, es usual que se cruce en su camino con personas, así como con el numeroso mobiliario instalado en él. Obviamente no es concebible que el robot pueda ir chocando con nada de su entorno. Por este motivo hemos puesto especial énfasis en la seguridad mostrada por el robot en todo momento y en diferentes situaciones, haciendo que

su comportamiento sea más cercano a la *timidez* que al *atrevimiento* a acercarse a los obstáculos.

Haciendo un balance global, se han programado los componentes básicos necesarios para una navegación autónoma total del robot real *Pioneer*. Por un lado, el robot es capaz de calcular la ruta óptima hasta un objetivo y navegar a través de su entorno evitando obstáculos previstos e imprevistos, combinando algoritmos de navegación global y local. Pero esta navegación no sería completa de no ser por el algoritmo de localización, que permite al robot saber su posición en el entorno por donde se mueve.

Respecto a esto último, añadir que en este proyecto se ha decidido acotar el mundo donde navega el robot por cuestiones de modificación del entorno, ya que para que el mecanismo de localización pueda funcionar es necesario establecer balizas de colores en el techo. Además, nuestro objetivo era conseguir la prueba de concepto del algoritmo sobre el robot real en el laboratorio, más que su funcionamiento robusto a lo largo de todo el edificio.

Respecto a los conocimientos aportados por el proyecto, podemos destacar los conocimientos adquiridos sobre la técnicas robóticas utilizadas actualmente para sistemas de navegación, tanto globales como locales, y las técnicas de autolocalización tanto en exteriores como en interiores; más concretamente en este último campo, un tema candente de investigación. Igualmente, se han adquirido conocimientos de lógica borrosa, necesarios para crear y utilizar controladores borrosos usados en los esquemas de pilotaje. También se han analizado y estudiado algoritmos evolutivos, y en particular hemos profundizado en el filtro de partículas dentro de los Métodos de MonteCarlo.

Dado que éste es un proyecto muy amplio en cuanto a técnicas empleadas y herramientas auxiliares necesarias, hemos aprendido a utilizar un gran abanico de utilidades de software libre: *(i)* la plataforma *JDE* como entorno de desarrollo de comportamientos autónomos sobre el robot *Pioneer*; además de *(ii)* una serie de librerías, aplicaciones y demás herramientas muy importantes hoy día en lo que desarrollo software se refiere. Mencionar por ejemplo *Gnuplot*, *Latex*, OpenGL, etc.

7.2. Trabajos futuros

El fin último del trabajo desarrollado en este proyecto es enfocar todos esos conocimientos, técnicas y algoritmos implementadas hacia la realización de un *robot guía*. Ya hemos mencionado a lo largo de esta memoria que todo lo aquí mostrado es un prototipo de un futuro robot que podría hacer las veces de robot guía del Departamental II de la Universidad Rey Juan Carlos, donde su cometido fuera concretamente guiar a los visitantes al despacho que deseen ir.

Como líneas por las que se puede continuar el trabajo aquí realizado, en primer lugar habría que tratar con un espacio mucho más amplio que el aquí experimentado.

Además, tal trabajo conllevaría un gran esfuerzo de integración de los distintos componentes necesarios para una navegación autónoma robusta: navegación global, navegación local y localización.

Haciéndo especial énfasis en este último aspecto, ya que de él depende que la navegación global sea más o menos robusta. Además, y tratando en un entorno tan simétrico como pueden ser los pasillos de este departamento, la estimación correcta de la posición del robot resulta más difícil.

Por otro lado, lejos de la aplicación de robot guía, se podría decir que con todo el trabajo llevado a cabo en este proyecto se cubren importantes aspectos de la navegación autónoma de un robot móvil, con lo que los propósitos a los que se destine un artefacto con tales características pueden ser variados. Otra rama de investigación podría estar enfocada en conseguir objetivos tales como los presentados en la competición *Urban Challenge* (1).

Bibliografía

[ActivMedia, 2002] Robotics ActivMedia. *Pioneer 2. Operations Manual.* ActivMedia Robotics, 2002.

[Barrera *et al.*, 2005] Pablo Barrera, Jose María Cañas, y Vicente Matellán. Visual object tracking in 3d with color based particle filter. In *Int. Journal of Information Technology*, 2005.

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

[Borenstein, 1991] J. Borenstein. The vector field histogram - fast obstacle avoidance for mobile robots. *IEEE Journal of Robotics and Automation*, 1991.

[Calvo, 2004] Roberto Calvo. Comportamiento sigue personas con visión direccional. *Proyecto fin de carrera Ing. Informática, Universidad Rey Juan Carlos*, 2004.

[Cañas Plaza *et al.*, 2007] José M. Cañas Plaza, Antonio Pineda, Jesús Ruíz-Ayúcar, José A. Santos, y Javier Martín. *Programación de robots con la plataforma jdec.* URJC, 2007.

[Cañas Plaza, 2003] José María Cañas Plaza. *Jerarquía Dinámica de Esquemas para la generación de comportamiento autónomo.* PhD thesis, Universidad Politécnica de Madrid, 2003.

[Cortés, 2007] Ángel Cortés. Localización y construcción de mapas en un robot de interiores. *Proyecto fin de carrera Ing. Informática, Universidad Rey Juan Carlos*, 2007.

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

[Fox *et al.*, 1999] Dieter Fox, Wolfram Burgard, Frank Dellaert, y Sebastian Thrun. Monte carlo localization: efficient position estimation for mobile robots. In *In Proceedings of the 16th AAAI National Conference on Artificial Intelligence*, pages 343–349, 1999.

[Glasius *et al.*, 2004] R. Glasius, A. Komoda, y S. Gielen. Neural networks dynamics for path planning and obstacle avoidance. 2004.

[Hidalgo, 2006] Victor Hidalgo. Comportamiento de persecución de un congénere con el robot pioneer. *Proyecto fin de carrera Ing. Tec. Informática de sistemas, Universidad Rey Juan Carlos*, 2006.

[Isado, 2005] José Raúl Isado. Navegación global por el método del gradiente. *Proyecto fin de carrera Ing. Informática, Universidad Rey Juan Carlos*, 2005.

[Isard y Blake, 1998] M. Isard y A. Blake. Condensation-conditional density propagation for visual tracking. *Int. J. Computer Vision*, 1998.

[Kachach, 2005] Redouane Kachach. Localización del robot pioneer basada en láser. *Proyecto fin de carrera Ing. Tec. Informática de sistemas, Universidad Rey Juan Carlos*, 2005.

[Khatib, 1984] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *IEEE International Conference on Robotics and Automation*, 1984.

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

[Krogh, 1985] B. H. Krogh. A generalized potential field approach to obstacle avoidance control. *International Robotics Research Conference*, 1985.

[Lobato, 2003] David Lobato. Navegación local con ventana dinámica para un robot móvil. *Proyecto fin de carrera Ing. Tec. Informática de gestión, Universidad Rey Juan Carlos*, 2003.

[López, 2005a] Alberto López. Localización de un robot con visión local. *Proyecto fin de carrera Ing. Informática, Universidad Rey Juan Carlos*, 2005.

[López, 2005b] Alejandro López. Navegación global utilizando método del grafo de visibilidad. *Proyecto fin de carrera Ing. Informática, Universidad Rey Juan Carlos*, 2005.

[Mackay, 1999] D.J.C. Mackay. Introduction to monte carlo methods. In *In M. Jordan, editor, Learning in Graphical Models, MIT Press*, pages 175–204, 1999.

[Moreno *et al.*, 2004] L. Moreno, B. L. Boada, y D. Blanco. Symbolic place recognition in voronoi-based maps by using hidden markov models. In *Journal of Intelligent and Robotics Systems.*, page vol. 39, 2004.

[P. Gerkey y Howard, 2003] Brian P. Gerkey y Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the international conference on Advanced Robotics.*, pages 1–36, 2003.

[Thorpe, 1985] C. F. Thorpe. Path relaxation: Path planning for a mobile robot. *Carnegie-Mellon University, The Robotics Institute, Mobile Robots Laboratory, Autonomous Mobile Robots, Annual Report*, 1985.

[Thrun, 2000] S. Thrun. Probabilistic algorithms in robotics. *AI Magazine*, 2000.