UNIVERSIDAD REY JUAN CARLOS

Ingeniería Técnica en Informática de Gestión

Escuela Superior de Ciencias Experimentales y Tecnología

Curso académico 2002-2003

Proyecto Fin de Carrera

Navegación local con ventana dinámica para un robot móvil

Tutor: José María Cañas Plaza

Autor: David Lobato Bravo

Febrero 2003

A Tamara por su apoyo incondicional

Agradecimientos

Especialmente a mi familia por la paciencia y apoyo mostrado. A la dedicación de José María Cañas, sin el cual no hubiera sido posible este proyecto. Y al grupo de robótica de la URJC y a Pedro de las Heras y su portátil.

Índice general

Capítulo 1

Introducción

1.1 Un poco de Historia

Fue Karel Capek, quien utilizó por primera vez el termino *robot* en su obra teatral Rossum's Universal Robots [9]. La obra, estrenada en 1921, narra la historia de Rossum, un científico, que consigue crear *robots* capaces de servir a la clase humana. Tras su perfeccionamiento, los *robots* se rebelan contra sus amos, destruyendo toda vida humana. La palabra robot, proviene de la palabra eslava *robota*, que se refiere al trabajo realizado de manera forzosa. El término, continúa usándose en varios escritos posteriores pertenecientes al género literario de la ciencia-ficción, pero no fue hasta años más tarde, en 1941, cuando Isaac Asimov utilizó por primera vez el termino robótica para referirse a la tecnología de los robots.

Como precursores históricos de los robots actuales, tenemos diversos autómatas, diseñados y construidos por grandes mentes muy avanzadas a su época. Como ejemplos citar el Gallo de la catedral de Estrasburgo de autor desconocido en 1352, el León mecánico de Leonardo Da Vinci en 1499, varios muñecos animados de Jacques Vaucanson en 1738, autor también del primer telar mecánico, o la muñeca dibujante de Henry Maillardet en 1805.

La robótica como hoy la conocemos, surge hacia la década de los 50, con la construcción de las primeras computadoras. Aunque no será hasta la década de los 70 con los primeros microprocesadores cuando comience su vertiginoso avance. Las primeras patentes aparecen en 1946, describiendo primitivos robots para el traslado de maquinaria. Será en 1954, cuando George Devol, diseñe el primer mecanismo programable aplicado a la industria, el *Universal Automation*, más tarde reducido a Unimation y que sería el corazón del primer brazo robótico industrial. Junto con Joseph F. Engelberger,

G. Devol formará la primera compañía comercial dedicada a la robótica, Unimation Inc., dedicada a robots industriales para cadenas de montaje.

Actualmente, el concepto de robótica incluye los sistemas móviles autónomos, capaces de desenvolverse por sí mismos en entornos desconocidos y variantes sin necesidad de supervisión humana. En líneas generales, la historia de la robótica se puede dividir en tres generaciones [5]:

1ª .- El robot es capaz de repetir una secuencia pregrabada de movimientos, sin realimentación sensorial (control en lazo abierto).

2ª .- El robot adquiere información limitada del entorno y actúa en consecuencia (control en lazo cerrado). Puede localizar, clasificar (mediante visión) y detectar esfuerzos, para adaptar sus movimientos.

3ª .- El robot utiliza la inteligencia artificial para resolver el problema planteado, su programación se realiza mediante el empleo de un lenguaje de alto nivel.

Hoy día, la robótica no solo es aplicada a la industria, sino que empiezan a aparecer los primeros robots domésticos [17], generalmente para el entretenimiento, como son los kits robóticos de Lego MindStorm o el robot Aibo de Sony capaces de realizar comportamientos sencillos.

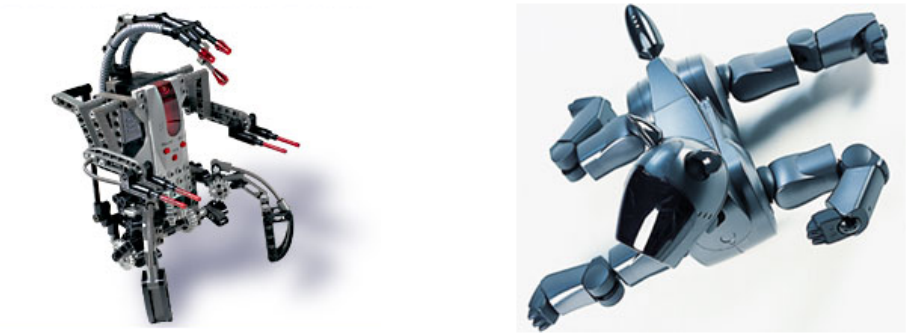

Figura 1.1: Robot Lego MindStorm y Aibo de Sony.

1.2 Robótica en la URJC

Como equipo de trabajo, el grupo de robótica de la URJC se interesa en diversos temas relacionados con la robótica y la inteligencia artificial.

Una de las líneas de trabajo del grupo persigue la consecución de un equipo capaz de competir en la RoboCup. La RoboCup es una competición nacida en 1996 con la intención de incentivar el interés por la ciencia y la tecnología en la que equipos compuestos por robots juegan al fútbol. La competición aporta un escenario de investigación tanto para la robótica, como para la inteligencia artificial. De esta manera, y con el aliciente de la competición y la vistosidad de los encuentros, cada año se celebra este evento, enfrentando a equipos de distintos lugares del mundo en alguna de las diferentes categorías. Dichas categorías van desde los equipos simulados por ordenador, hasta los formados por robots humanoides. La URJC tiene como objetivo en la RoboCup, participar en la categoría de robots móviles pequeños. Para ello se adquirieron los robots EyeBot [1].

Los robots EyeBot están especialmente diseñados para la RoboCup, y constan de varios sensores (una cámara, tres sensores de infrarrojos y un par de encoders) y varios actuadores (motores y servos para el pateador y la cámara). Como elemento de proceso incorpora un microprocesador Motorola 68332 a 35Mhz, que le permite realizar gran cantidad de cálculos, aun dado su reducido tamaño. Además incorpora un sistema operativo propio, RoBios, que facilita la tarea de programación, aportando incluso mecanismos de multiprogramación.

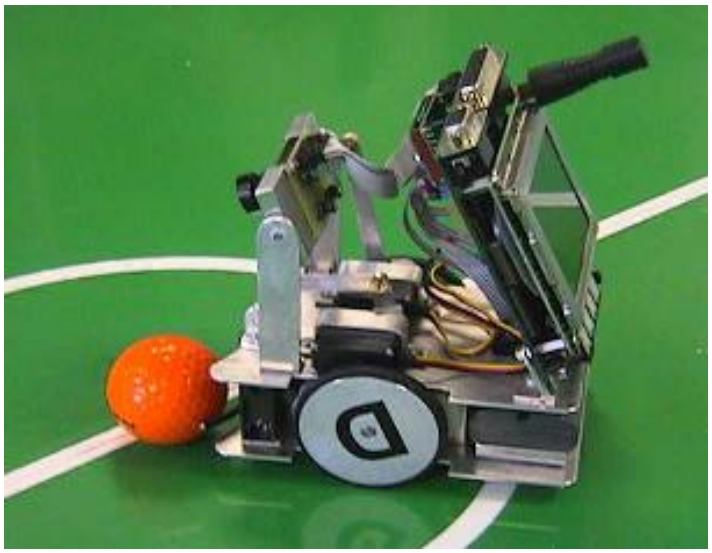

Figura 1.2: Robot EyeBot.

Orientados a la competición, ya se han realizado varios proyectos fin de carrera

[1] www.ee.uwa.edu.au/~braunl/eyebot/

para el EyeBot, como el teleoperador [14], un protocolo de comunicaciones [4], o el comportamiento sigue pelota con visión local [13].

Una segunda línea de trabajo del grupo es el desarrollo de un repertorio de comportamientos autónomos en robots de interiores. En concreto se ha desarrollado una arquitectura de control llamada JDE (**J**erarquía **D**inámica de **E**squemas)[8]. En esencia JDE separa la percepción y el control en diferentes esquemas intercomunicados entre sí. Los esquemas perceptivos se encargan de recibir los estímulos de los sensores, creando las estructuras de datos necesarias para su posterior proceso. Y los esquemas motores (o de control) se encargan de digerir la información sensorizada. En síntesis no es más que la aplicación de *"Divide y Vencerás"* a la generación de comportamientos complejos. Sobre la arquitectura conceptual JDE se hablará en detalle en el capitulo 3. Actualmente se ha desarrollado una implementación de la arquitectura de control JDE para el robot Pioneer, que además incorpora varios esquemas que materializan tanto comportamientos autónomos (`gotopoint`) como percepciones complejas del entorno (detección de puertas, rejillas de ocupación) totalmente reutilizables. Dicha plataforma móvil mostrada en la figura 1.3,es otro de los robots que posee el grupo de robótica de la URJC. El robot Pioneer, es un robot de tamaño medio y con un equipamiento muy superior al que incorpora el EyeBot.

Figura 1.3: Robot Pioneer.

Sus características más destacables son sus sensores sonar, su potencia motora y su capacidad de cálculo proporcional a la máquina que lo controle, ya que incorpora una configuración cliente-servidor que lo hace posible.

En esta línea de trabajo de comportamientos para robots de interiores, se ha realizado el proyecto fin de carrera sigue pared en un robot con visión local para el robot EyeBot[11], y es también directriz en la que se enmarca este trabajo. Este proyecto fin de carrera persigue conseguir un comportamiento que permita al robot viajar a lugares cercanos evitando las colisiones con los obstáculos próximos. Para ello, se utilizará una técnica similar a la desarrollada en [10], que basa su funcionamiento en la búsqueda de un par óptimo (velocidad traslación, velocidad rotación) sobre un subconjunto del *espacio de velocidades*. A dicho subconjunto se le denomina *ventana dinámica*, y sus elementos son velocidades que nuestro robot podría conseguir en el siguiente instante, dadas sus limitaciones dinámicas (aceleración). Así, conseguimos incorporar la dinámica de nuestro robot en la búsqueda del comando motor adecuado.

Una descripción detallada del método se realizará en el capítulo 4. A modo de enfoque sobre las diferentes técnicas de navegación local existentes en la actualidad, la siguiente sección aporta un repaso general sobre el estado del arte.

1.3 Navegación en sistemas móviles

Para lograr sistemas móviles autónomos estos necesitan de mecanismos de navegación fiables, que les permitan moverse por su entorno de manera segura y eficiente. Podemos diferenciar dos grupos de técnicas de navegación, locales y globales. Las primeras, resuelven problemas de navegación a metas cercanas, y basan su comportamiento en los estímulos extraídos de sus sistemas sensoriales. Son métodos con una alta reactividad a los cambios en el entorno. y nos permiten resolver problemas como la evitación de obstáculos o el seguimiento de una pared. Sus estimulos provienen de diferentes sistemas sensoriales, desde simples medidores de distancias (sonares, infrarrojos,...), hasta complejas imágenes tridimensionales del entorno. En el otro grupo, tenemos las técnicas globales, que nos permiten la resolución de problemas de navegación a metas lejanas. Sus métodos se apoyan en la planificación de caminos y generalmente sus datos provienen de mapas del entorno (paredes, pasillos, habitaciones,...). La actuación y reactividad de estos métodos, se delega a técnicas de navegación local. Es decir, nuestro método global planifica las metas locales, que un método local se encargara de alcanzar, guiando así al robot hacia la meta lejana.

Nuestro caso, se centra en los métodos de navegación local o reactivos sobre los que se apoyarán métodos de planificación global. En concreto se desea una navegación

eficiente guiada por una dirección objetivo, y capaz de evitar los obstáculos que se interpongan en su camino. Existen multitud de enfoques para lograr este objetivo, y a continuación de forma breve se exponen algunos de los más destacados en la literatura.

Campos Virtuales de Fuerza (VFF): Este método desarrollado por J. Borenstein en [18] se basa en la idea de fuerzas repulsivas y atractivas ejercidas por los obstáculos y el punto objetivo, respectivamente, sobre el robot. Como representación del entorno utiliza una rejilla de ocupación centrada en el robot, en la que cada celda contiene la probabilidad de que exista un obstáculo en esa posición. Así, cada celda aplica una fuerza repulsiva sobre el robot con una magnitud proporcional a la probabilidad contenida en la celda, e inversamente proporcional al cuadrado de la distancia existente entre la celda y el robot. Por otra parte, existe una fuerza que atrae al robot hacia la dirección objetivo siendo la dirección elegida la resultante entre ésta y las fuerzas repulsivas. Este método no contempla la dinámica del robot.

Método de la Curvatura-Velocidad (CVM): Este método desarrollado por R. Simmons en [16], incorpora como novedad, la consideración de la dinámica del robot en el cálculo de los comandos de velocidad para cada instante. La búsqueda se realiza directamente sobre el *espacio de velocidades* o conjunto de todas aquellas velocidades que en un determinado instante se podrían comandar, y sobre el que nuestra velocidad actual esta representada por un punto. Para la simplificación de dicha búsqueda sólo se contemplan trayectorias circulares definidas por un par (velocidad de traslación , velocidad de rotación), de ahora en adelante (v,w). Así, con esta simplificación y la restricción impuesta por la dinámica del robot, se busca el par (v,w) que maximiza cierta función objetivo, que sopesa determinados criterios. Estos son conseguir la mejor orientación hacia la dirección objetivo, viajar alejado de los obstáculos, y hacerlo a velocidades de traslación altas. Con esto, en cada iteración se obtiene un comando motor que guiará a nuestro robot en el siguiente intervalo.

Método de la ventana dinámica (DWA): Este método, es el tema principal de este proyecto, y es en cierta manera un refinamiento de CVM,

aplicando sus mismas ideas para el cálculo del par (v,w). Sus particularidades se han comentado en la sección anterior, y se detallará en el capítulo 4.

Método de los carriles (LCM): El método de los carriles [15] suple las carencias de CVM y básicamente consiste en guiar al robot por la dirección objetivo yendo por la zona más despejada. Para lograrlo, se divide el entorno en carriles y se escoge el mejor (más ancho y libre), guiando al robot hacia este carril siempre y cuando no suponga un cambio brusco de la dirección. Una vez obtenida la dirección local objetivo, se le pasa al CVM para que cumpla con el trabajo.

Método del haz radial (BCM): Este método al igual que el LCM intenta suplir las carencias del CVM, pero con un enfoque diferente [6]. El uso de trayectorias circulares por parte del CVM, hace que éste obtenga una visión deformada del entorno, que en ciertas situaciones oculta direcciones libres. Para solventar esto, el BCM aprovecha la disposición radial de los sensores del robot. De esta manera se detectan ciertos huecos que con CVM pasaban desapercibidos. Una vez detectado el mejor hueco, BCM obtiene la dirección objetivo local y se la aplica al CVM.

La presente memoria incluye todos los detalles relevantes en la realización del comportamiento de navegación local con la ventana dinámica en el robot Pioneer. El escrito se articula en cinco capítulos. El capítulo 2 describe los objetivos esperados y requisitos de partida para el desarrollo del comportamiento. El capítulo 3 profundiza en la plataforma utilizada describiendo tanto el robot Pioneer, como la ya citada arquitectura JDE, además de otras herramientas software utilizadas. El capítulo 4 incluye una descripción informática del trabajo realizado. Y para finalizar, el capítulo 5, aporta las conclusiones extraídas, así como, las posibles mejoras aplicables al proyecto realizado.

Capítulo 2

Objetivos

Este capítulo, describe los objetivos del presente proyecto, navegación local en un robot Pioneer con ventana dinámica, así como los requisitos que debe cumplir su desarrollo.

El objetivo principal del presente proyecto, es implementar un método de navegación local basado en el enfoque de la ventana dinámica(DWA) [10] sobre el robot Pioneer, aplicando para ello la arquitectura de control JDE [8] desarrollada por el grupo de robótica de la URJC. Para ello, se deberán desarrollar un conjunto de esquemas, que relacionados entre sí, sean capaces de cumplir el comportamiento deseado.

2.1 Objetivos

Los objetivos que se persiguen con el desarrollo de este proyecto van, principalmente, encaminados a la implementación del método de navegación local DWA sobre la plataforma móvil Pioneer, utilizando únicamente sus sensores sonar para la percepción del entorno. Los objetivos intrínsecos al desarrollo de este método, incluyen obtener una alta reactividad en nuestro robot, y permitirle la navegación a velocidades altas de manera segura, manteniendo una orientación objetivo y esquivando los obstáculos imprevistos, tanto estáticos como dinámicos. El objetivo de obtener velocidades de navegación altas, se presenta como uno de los retos del proyecto, ya que la navegación aportada por otro tipo de métodos, solo consigue velocidades moderadas. En la búsqueda de velocidades mas altas ($1\frac{m}{s}$), aparecen dificultades genuinas, como las relacionadas con la velocidad de muestreo de los sensores o la capacidad de frenado del robot, las cuales se afrontarán en este proyecto.

Como objetivos enfocados al diseño, tenemos que el desarrollo seguirá las premisas de la arquitectura de control JDE, y así se dividirá en diferentes esquemas con funciona-

lidades específicas, bien perceptivas, bien de control. Como objetivo derivado del diseño basado en JDE, tenemos que el desarrollo de este proyecto permitirá la reutilización en otros comportamientos de los esquemas creados aquí. Así, un posible comportamiento de navegación global, podrá resolverse con un esquema que comande las direcciones objetivo adecuadas en cada instante, despreocupándose de la esquivación de obstáculos resuelta por nuestro esquema de navegación local.

Y por último, como objetivo enfocado a la implementación, este desarrollo se ajustará a la plataforma móvil Pioneer.Para ello será necesario obtener un conjunto de parámetros y criterios que optimicen el comportamiento sobre dicha plataforma, de manera que el comportamiento desarrollado sea capaz de explotar las capacidades con las que cuenta. En concreto se busca el aprovechamiento de la movilidad holonómica que posee el robot Pioneer, y que supondrá un añadido a la implementación original del enfoque de la ventana dinámica desarrollada en [10].

2.2 Requisitos

El desarrollo del proyecto estará guiado por los objetivos comentados anteriormente, y deberá ajustarse a ciertos requisitos de partida que permitirán su adaptación a la plataforma elegida, el robot Pioneer ,y asegurarán el buen funcionamiento del comportamiento de navegación local descrito.

Estos requisitos son:

Plataforma real :El comportamiento desarrollado, debe ser implementado de manera que funcione en el robot móvil Pioneer2-DXe. Así, se deben tener en cuenta sus características hardware (equipo sensorial y capacidades móviles) y las facilidades software para su programación.

Comportamiento :Dado que el comportamiento a implementar debe desenvolverse en un entorno real, éste debe reaccionar de manera efectiva y en el menor tiempo posible ante los cambios del entorno. Se deberá asegurar la robustez del comportamiento, principalmente para evitar daños al robot, de manera que reaccione lo mejor posible en diferentes entornos. Y por último, se buscará el ajuste adecuado del comportamiento sobre las capacidades móviles del robot Pioneer, que le permitan desplazarse a velocidades generalmente altas sobre el entorno sobre el que se desenvolverá.

Arquitectura de control :El desarrollo del comportamiento, deberá basarse en la arquitectura JDE, arquitectura creada y utilizada por el grupo de robótica de la URJC. La implementación de dicho comportamiento, se apoyará en la última versión disponible (jde1.3), al comienzo del proyecto, de la arquitectura JDE para el robot Pioneer, que aportará los mecanismos y esquemas básicos [8]. De esta manera, el problema se dividirá en diferentes esquemas, unos perceptivos y otros de control, que aportarán la jerarquía de esquemas capaz de describir el comportamiento deseado y facilitará su reutilización en otros comportamientos que así lo requieran.

Capítulo 3

Plataforma de desarrollo y herramientas software

Este capítulo aporta los detalles sobre las plataformas de trabajo utilizadas para el desarrollo del presente proyecto tanto a nivel hardware (robot Pioneer) como a nivel software (saphira) y conceptual (arquitectura JDE) . Además, se presentan, de manera somera, otros detalles acerca de las herramientas software utilizadas en alguna de las partes del proyecto, como son la multiprogramación con Pthreads y la programación de interfaces gráficas con QT.

3.1 El robot Pioneer

El Pioneer2-DXe (Pioneer de ahora en adelante) , es un robot móvil holonómico de tamaño medio para interiores. Sus dimensiones, le permiten participar en la RoboCup, en la liga mediana. Dadas sus buenas características, es un robot bastante popular en la comunidad robótica, siendo plataforma de investigación en bastantes universidades.

Su versatilidad permite la interconexión de numerosos periféricos, sensores o actuadores, suministrados por el fabricante (ActiveMedia) , o por terceros de manera muy sencilla. Algunos de estos periféricos son sensores láser, brazos robóticos, cámaras, unidades *pantilt*, etc . . . En la figura 3.1 se muestran algunos ejemplos.

En concreto, el robot Pioneer de la URJC posee una corona de ultrasonidos, con un total de dieciséis sonares dispuestos radialmente, más un cinturón de sensores de contacto (*bumpers*) delante y detrás (5+5) . También, tiene un par de cuenta vueltas (*encoders*) de 500 tics por vuelta en sendas ruedas. Además, se le ha añadido una *web cam*, que le permite obtener imágenes en color del entorno. Puede verse una imagen en la figura 3.2.

Figura 3.1: Versatilidad del Pioneer2-DXe.

Figura 3.2: Robot Pioneer de la URJC.

Para el movimiento, el robot incorpora dos ruedas motoras, y una rueda loca (para la estabilización). Los dos potentes motores incorporados le permiten desplazarse a velocidades de hasta 1.8 m/s o transportar hasta 23 kg de carga. Como elemento de proceso, incorpora un pequeño microcontrolador Hitachi HS8 encargado del manejo a bajo nivel de los distintos periféricos conectados, y de la comunicación con la unidad de proceso superior. Esta unidad de proceso superior, es en nuestro caso, un portátil colocado sobre la base del robot. Éste será el encargado de ejecutar los servidores que aportarán toda la información sensorial y transmitirán las órdenes de actuación al robot. Dicha configuración, es una de las posibilidades que ofrecen los robots de ActiveMedia, y responde a una configuración cliente-servidor. El resto de las configuraciones se muestran en la figura 3.3.

3.1.1 El sensor sonar

Los sensores sonar incorporados en el robot Pioneer, van a ser la fuente de nuestra representación del entorno. Gracias a ellos, obtendremos datos referentes a la distancia entre el Pioneer y los obstáculos próximos. Por ello, se dedica esta sección a la descripción de sus principios básicos de funcionamiento.

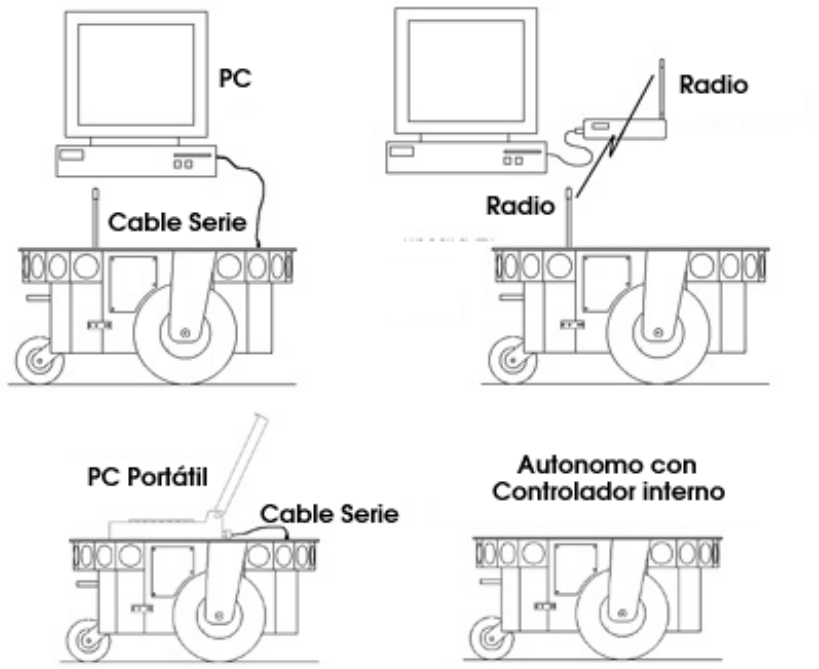

Figura 3.3: Configuraciones de los robots ActiveMedia.

Un sensor sonar, basa su funcionamiento en las propiedades de las ondas ultrasónicas. Dichas ondas, son ondas mecánicas, ya que necesitan de un medio material para su transmisión, que viajan por el espacio a una velocidad próxima a 330 m/s a 0ºC y aire seco. Como toda onda, una onda ultrasónica rebota en las superficies, efecto que conocemos como eco. Pues bien, el funcionamiento básico de un sonar, que se describe en la figura 3.4, utiliza esta propiedad de la siguiente manera:

1. Un transductor emite una onda sonora de frecuencia ultrasónica (por encima de los 44kHz) y a la vez lanza un cronometro.
2. Se espera la vuelta del eco.
3. Cuando se detecta el eco, se calcula el tiempo de ida y vuelta, llamado tiempo de vuelo. Puesto que conocemos la velocidad del ultrasonido, podemos calcular la distancia recorrida por la onda, que será exactamente el doble de la distancia al obstáculo.

Dicho mecanismo, es el utilizado en los sonares del robot Pioneer. Sus sensores de ultrasonidos, desarrollados por Polaroid Co., son capaces de emitir ondas sonoras de 40 a 100 Khz, y pueden medir distancias a obstáculos desde 15cm a 10m.

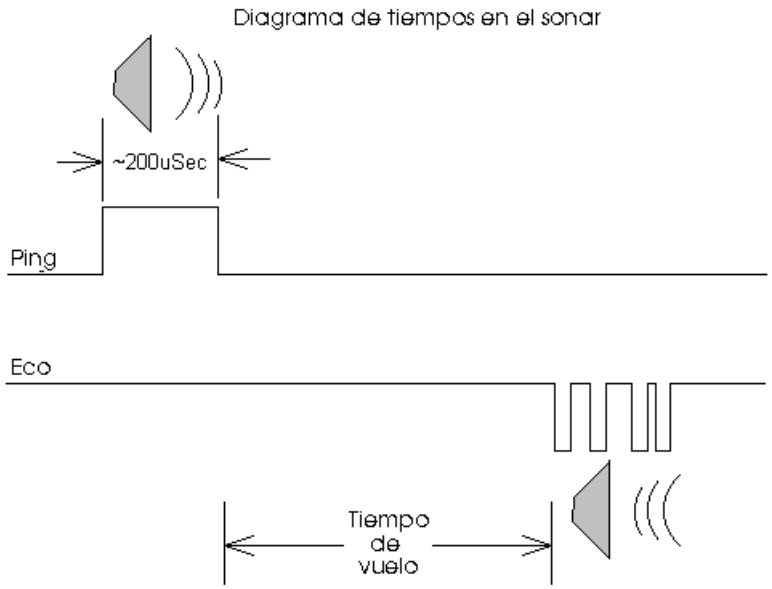

Figura 3.4: Funcionamiento de un sonar.

El espacio sensorizado por nuestros sonares, no se limita a la recta normal al sensor, sino que se expande, aproximadamente, como un cono de ángulo 30º. En concreto, el ángulo de dicho cono depende del cono de propagación de la onda ultrasónica. De manera experimental se obtuvo la figura 3.5 que describe el modelo geométrico sonar del Pioneer.

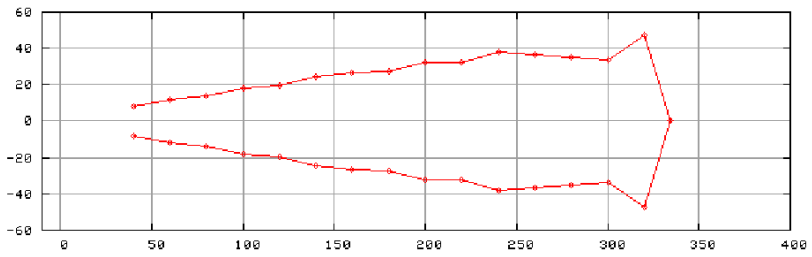

Figura 3.5: Modelo sonar del Pioneer.

Aunque las características de los sonares, en la detección de obstáculos, son bastante buenas con respecto a otros sensores, como los de infrarrojos, también tienen limitaciones. En concreto, este tipo de sensores sufren de ruidos radiales (estimación inexacta de la distancia), reflexión especular sobre ciertas superficies y ángulos de incidencia (múltiples ecos que no permiten detectar el obstáculo real) y de la denominada incertidumbre angular. Este último defecto, se refiere al comentado ángulo α de propa-

gación del sonar. Así, dada una medición d no podemos concretar mas que el obstáculo que produjo la reflexión, se encuentra en algún punto del arco con origen en el sensor, radio d y ángulo α. Si se toma un enfoque conservador para las lecturas de este sensor, se interpretará como ocupado todo el arco comentado.

3.1.2 El encoder

Al igual que los sensores sonar, los encoders, formarán parte de la información sensorial recabada por nuestro comportamiento, a fin de conseguir sus objetivos. En concreto, los encoders nos van a permitir al cálculo de la velocidad a la que se desplaza nuestro robot en cada instante.

Un encoder, no es más que un mecanismo capaz de contar las vueltas de un eje. Los incorporados en el robot Pioneer en sendas ruedas, son capaces de contar 500 tics por vuelta, por lo que nos confieren una precisión de 0.72^o.

3.1.3 Saphira

Saphira es el software incluido con los robots de ActiveMedia. Dicho software incluye diferentes herramientas, entre las que podemos destacar mecanismos para la multiprogramación, navegadores, constructores de mapas, e incluso un potente simulador. Saphira además, nos brinda un API común para el acceso a los recursos del robot (cualquiera de los soportados), incluso si éste está siendo simulado. Así, un programa es capaz de funcionar, bien en el robot, bien en el simulador de manera transparente al programador. Dicha API-Saphira, es local al robot, por lo que para acceder a través de ella, nuestro programa debe ejecutarse sobre la máquina conectada físicamente al robot. Esta limitación será salvada por los mecanismos implementados en los servidores sockets de JDE, que se comentarán más adelante. Como se ha comentado, Saphira incluye un simulador, que nos permitirá simular nuestro robot Pioneer sobre diferentes mundos, e incluso con diferentes parámetros (velocidades máximas, número de sensores, ...). De esta manera, salvaguardaremos la integridad del robot real, mientras depuramos nuestro comportamiento. El simulador de Saphira se muestra en la figura 3.6.

En cuanto a la licencia del software Saphira, decir que es un producto propietario sujeto a licencias comerciales. Paralelamente a Saphira, ha surgido Aria, que es la versión GPL del software para los robots de ActiveMedia. Esta distribuye las herramientas básicas para desarrollo, como el API de acceso a los recursos del robot, y el

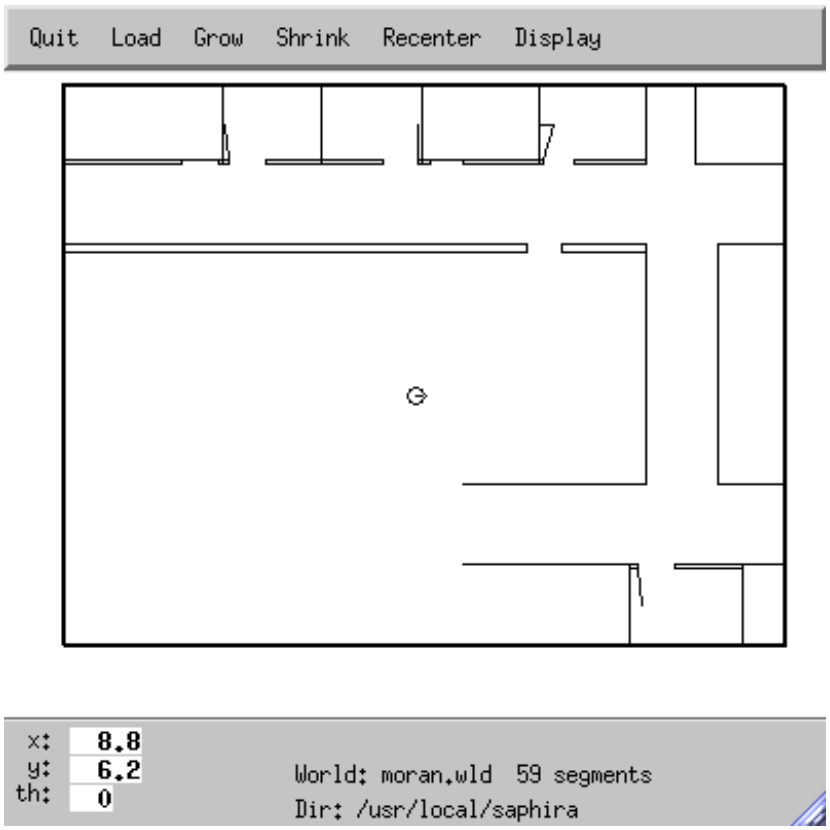

Figura 3.6: Simulador Saphira para el Pioneer.

simulador. Con el fin de usar y fomentar el software libre, el grupo de robótica de la URJC, adoptará el software Aria como plataforma software en un futuro próximo.

3.2 Jerarquía dinámica de esquemas

La construcción de comportamientos autónomos sobre plataformas robóticas requiere, no sólo de buenos dispositivos sensoriales y actuadores que nos permitan obtener mejores y más precisos estímulos y movimientos, sino que requiere de la *combinación inteligente* de éstos.

Para conseguir este objetivo, la construcción de un comportamiento autónomo debe basarse en una organización de la sensorización, actuación y capacidad de cálculo del robot. Dicha organización recibe el nombre de arquitectura de control para un robot, y gracias a su aplicación es posible conseguir comportamientos muy complejos.

JDE (**J**erarquía **D**inámica de **E**squemas), es la arquitectura de control diseñada por el grupo de robótica de la URJC [8], y sobre la cual basa la mayoría de sus trabajos.

Podemos distinguir entre tres tipos de arquitecturas de control:

Arquitecturas deliberativas:

Aplican la inteligencia artificial clásica al desarrollo de un comportamiento. Dicha arquitectura centra sus esfuerzos en el modelado del espacio y la planificación de las actuaciones sobre éste. Su esquema típico de funcionamiento es un bucle infinito con los siguientes pasos:

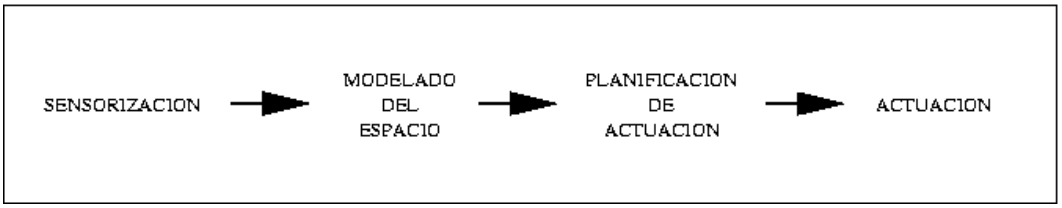

Figura 3.7: Etapas en arquitecturas deliberativas.

Gran parte de la inteligencia del robot yace en la etapa de deducción, donde se rastrea el espacio de posibles estados en busca del camino que lleva a la consecución del objetivo.

Arquitecturas basadas en comportamiento:

La principal característica de este tipo de arquitecturas de control, es que distribuyen el control en diferentes unidades de comportamiento básico, llamadas competencias, niveles, esquemas, agentes, etc...

Cada unidad de comportamiento básico engloba su sensorización y actuación en un rápido bucle, con un objetivo parcial propio. La dificultad ahora, radica en la coordinación de estas unidades para conseguir un objetivo global deseado. Dos paradigmas definen soluciones al problema **Arbitraje** y **Fusión de comandos**. Mediante el arbitraje, las unidades de comportamiento deben competir entre sí para ganar el control, sólo una en cada instante aplica su control. La fusión de comandos, en cambio, produce un control derivado de la combinación de las órdenes generadas por todas la unidades.

Arquitecturas híbridas:

En los últimos años, se han buscado soluciones híbridas, que unan la potencia de las dos arquitecturas comentadas. En concreto, la deducción sobre el entorno de las deliberativas, y la reactividad de las basadas en comportamiento.

3.2.1 Arquitectura cognitiva

La arquitectura de control JDE diseñada por el grupo de robótica de la URJC es principalmente reactiva, aunque puede incorporar deliberación [8]. En concreto, la unidad de comportamiento se denomina esquema. El control esta distribuido en una jerarquía de esquemas. Un esquema es un flujo de ejecución con un objetivo concreto. Además, éste puede ser activado o desactivado y admite una serie de parámetros que modulan su comportamiento. Los esquemas se han implementado como procesos iterativos, cuyo período viene determinado como uno de sus parámetros principales.

Existen dos tipos de esquemas **Esquemas perceptivos** y **Esquemas motores**.Los esquemas perceptivos producen la información que otros esquemas consumen. Por lo general, dicha información es información sensorial más o menos refinada, que es la entrada de los esquemas motores. Por su parte, los esquemas motores producen otras salidas, que son señales de control para esquemas de niveles inferiores. Un esquema de control puede implementarse utilizando diferentes técnicas, como reglas sencillas desde los sensores, controladores de lógica borrosa, máquinas de estados finitos,..., siempre y cuando sean iterativos y suspendibles.

Los esquemas se organizan en jerarquías, que se construyen de manera dinámica. Es decir, si un esquema motor encargado de cierto comportamiento requiere de cierta información, activará el esquema perceptivo que se la facilite, el cual a su vez puede necesitar de otros esquemas, activándose recursivamente todos los esquemas necesarios. Además, cuando se activa un esquema, también se modula su funcionamiento mediante el paso de una serie de parámetros, de manera que un esquema puede funcionar de manera diferente dependiendo de quién lo active y cómo lo module. Así, tenemos una jerarquía de esquemas creada de manera dinámica.

Debemos tener en cuenta, que en cada nivel, sólo puede estar activo un esquema motor, decidiéndose cual de los posibles se activa mediante un proceso de arbitraje. El resto de esquemas, permanecen suspendidos, aunque preparados para su activación en cuanto sean necesarios.

En la figura 3.8 podemos ver una jerarquía de esquemas, que ejecutan cierto comportamiento complejo. Los círculos representan esquemas motores, y los cuadrados esquemas perceptivos. Los esquemas activos aparecen en línea continua, y por nivel, el esquema motor activo aparece relleno.Como se puede ver en la figura 3.8, también es posible que un esquema perceptivo dependa de otros esquemas perceptivos, teniendo así, percepciones compuestas que permiten un mayor grado de abstracción desde los

esquemas inferiores.

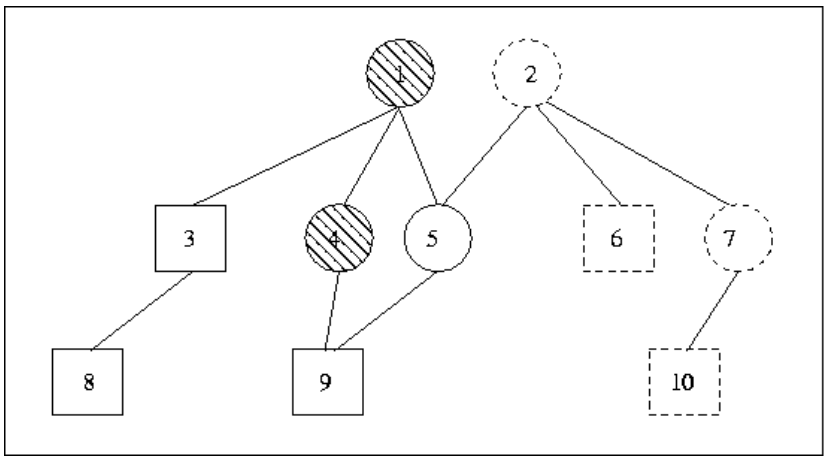

Figura 3.8: Jerarquía de esquemas.

De esta manera, tenemos que un comportamiento robótico determinado es generado por una jerarquía de esquemas única. También, es señalable, que los esquemas no pertenecen a un nivel en particular, sino que pueden ser reutilizados en diferentes niveles para producir diferentes comportamientos.

3.2.2 Arquitectura software

La arquitectura software desarrollada para la implementación de JDE sobre el robot Pioneer, incorpora algunos mecanismos que suplen las carencias de Saphira. Por ello, el software que se describe en esta sección se divide, en la implementación propiamente dicha de JDE tal y como se describió en la sección anterior, más un par de servidores que aportan mecanismos para el acceso remoto a los recursos. Todo se a programado usando Ansi-C.

La arquitectura software completa se presenta en la figura 3.9. En élla, vemos en la parte mas baja y en contacto directo con el robot, los servidores `oculo` y `otos`, encargados de exportar sus recursos (sensores y actuadores), haciendo posible el acceso remoto a los mismos. Por su parte, el servidor `oculo` exporta la visión del robot, obtenida a partir de su *web cam*. Este servidor interactúa con el API `video4linux` para el manejo de la cámara, entregando sus imágenes bajo demanda expresa de los clientes. Dado que el simulador desarrollado por Saphira, no incorpora mecanismos de visión, `oculo` solo puede conectarse al robot real. Su uso no a sido necesario en este proyecto, por lo que no se aportarán mas detalles.

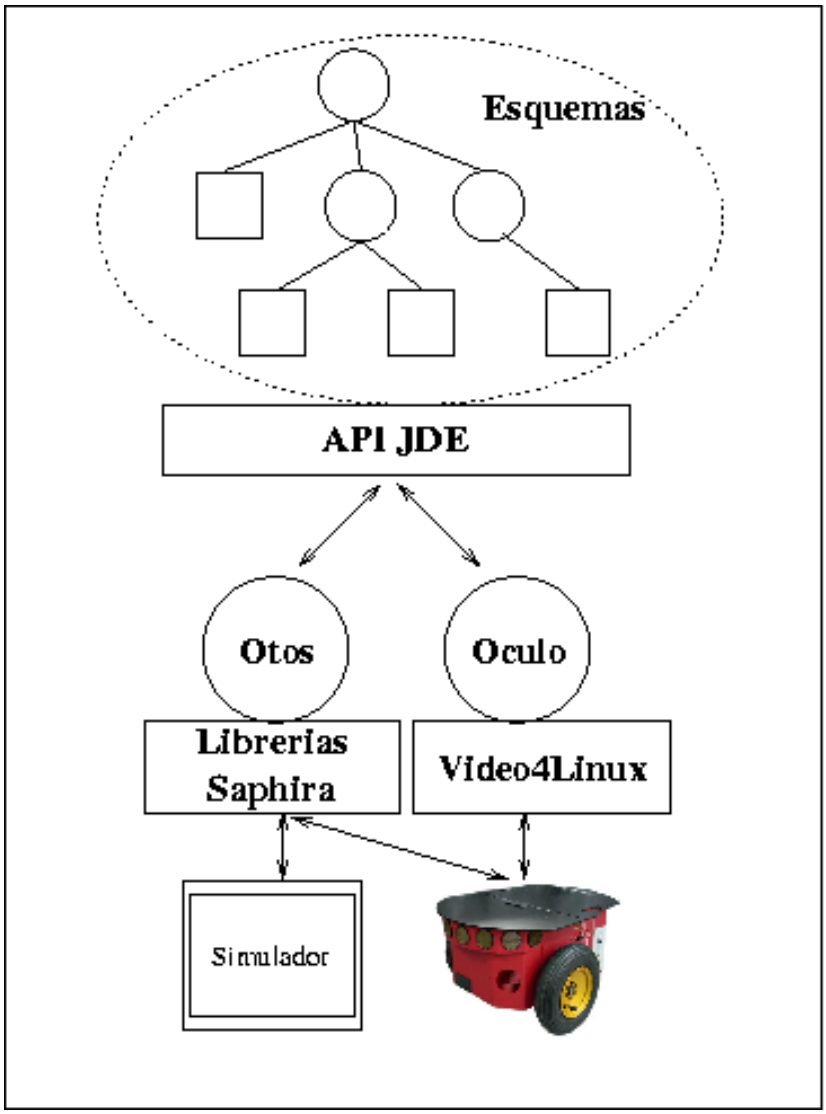

Figura 3.9: Arquitectura software de JDE.

El servidor `otos`, en cambio, interactúa con el API-Saphira, exportando los recursos sensoriales y actuadores del robot Pioneer. Mediante este servidor, podemos monitorizar y teleoperar a nuestro robot de manera remota a través de una conexión TCP. Cabe destacar, que la utilización del API-Saphira para el acceso a los recursos del robot, nos brinda una interfaz transparente para el programador, con respecto al tipo de robot utilizado, bien real, bien simulado.

La comunicación hacia los servidores `oculo` y `otos`, se realiza mediante el API-Jde, que es un protocolo de mensajes codificados como texto. Cada mensaje incluye el código de operación y los parámetros requeridos. Dicho protocolo se apoya en conexiones TCP, que aporta fiabilidad a la comunicación. En concreto, el servidor otos maneja

tres tipos de mensajes con sus clientes (esquemas básicos de JDE), unos gestionan la apertura/cierre de una conexión, otros manejan la parte actuadora (motores), y un último tipo maneja la parte sensorial (encoders,bumpers y sonares).

Por encima, tenemos un conjunto de esquemas básicos, clientes de los servidores `oculo` y `otos`, que hacen accesible los recursos a toda la jerarquía de esquemas. Estos esquemas son `sensationsoculo` y `sensationsotos`.

Y por último, en la parte superior, tenemos los esquemas que conforman la jerarquía. Dichos esquemas, comparten un esqueleto común, que define las operaciones y variables que los convierte en esquemas JDE, y que se han detallado en la sección anterior.

3.3 Herramientas software

En esta sección se introducen las herramientas software de las que depende la implementación de este proyecto. Como herramientas denominamos a aquellas partes software creadas por terceros que juegan un papel importante en alguna de las partes del desarrollo de este proyecto. Por ello, se dedicará un apartado a cada una de ellas, mostrando sus características más generales y su aplicación en nuestro desarrollo.

3.3.1 Paralelismo con Pthreads

La utilidad de la multiprogramación en la robótica, se hace imprescindible dado la gran cantidad de cosas que se deben hacer incluso en el comportamiento más simple (leer sensores, construir representación, refrescar pantalla, decidir actuación, ...). Por ello, cualquier software que acompaña a un robot incluye facilidades para la multiprogramación, véase el caso de `Ray-Scheduler` del robot B21 [7], las `Task` de Saphira [12] o `RoBios` para el robot EyeBot [14]. Así, dada la necesidad de un mecanismo de multiprogramación, se decidió la utilización de una implementación estándar, las Pthreads, en vez de los mecanismos propietarios ofrecidos por Saphira, para facilitar el desarrollo multiplataforma. Este mecanismo, día a día, se consolida en los sistemas operativos modernos, y nos ofrece hebras de kernel eficientes y robustas frente a bloqueos.

Una hebra, técnicamente, puede definirse como un flujo de instrucciones independiente, que puede ser planificado por el sistema operativo [2]. En un sistema UNIX una hebra existe dentro de un proceso, utilizando sus recursos (descriptores de fichero, espacio de memoria, ...). Su flujo de ejecución está separado del de su proceso padre. Un proceso puede contener varias hebras que comparten sus recursos. Dichos flujos

terminan cuando lo hace el proceso padre, o antes.

Las ventajas principales de utilizar multiprogramación basada en hebras en vez de múltiples procesos son:

1. La principal ventaja, es sin duda, el coste añadido a la creación de un nuevo flujo de ejecución, mucho menor que la creación de procesos nuevos. La creación de una hebra requiere de estructuras de datos muy sencillas y pequeñas (pila, registros, propiedades de planificación), mientras que un proceso requiere de una estructura más compleja y costosa de inicializar.
2. Los mecanismos de comunicación entre hebras son mucho más eficientes, al utilizar un espacio de memoria compartido, y fáciles de usar que los mecanismos de comunicación entre procesos (IPC).

En la tabla 3.1 se muestra una comparación de tiempos de las rutinas fork() y pthreads_create(), encargadas de crear procesos y hebras respectivamente. Cada experimento crea 500000 procesos/hebras, el tiempo, en segundos, se ha medido con la utilidad time. El código de este experimento se puede encontrar en [2].

Sistema Linux 2.4	**fork()**			**pthread_create()**		
	real	**user**	**sys**	**real**	**user**	**sys**
Athlon 1800+ 256Mb Memoria	56.24	10.24	45.46	22.35	4.13	18.19
Pentium II 400Mhz 128Mb Memoria	201.31	30.83	170.45	78.67	17.48	61.13
AMD K6-3 400Mhz 128Mb Memoria	239.20	18.13	201.23	95.36	4.24	11.75

Cuadro 3.1: Comparativa de tiempos fork() vs. pthread_create()

Históricamente, cada plataforma implementaba su propia versión de software para multiprogramación. De esta manera, el desarrollo de aplicaciones portables era muy difícil, dadas las diferencias en las implementaciones. Por ello, y probada la eficiencia de la multiprogramación basada en hebras se creo una interfaz de programación estándar. Dicho estándar se especifica en el IEEE POSIX 1003.1c de 1995. Las implementaciones que siguen este estándar, se llaman hebras POSIX, o `Pthreads` (*Posix threads*).

`Pthreads` se definen como un conjunto de tipos y funciones en el lenguaje de programación C. Las funciones incluidas en el API (alrededor de sesenta) permiten la gestión de hebras (creación, eliminación, ...), el manejo de estructuras para exclusión mutua (*mutex*) y funciones para el uso de variables condición. El acceso concurrente a

una variable compartida por varias hebras, puede producir los que se conoce como condiciones de carera (*race conditions*), conduciendo la ejecución a estados impredecibles. Uno de los mecanismos para el control del acceso concurrente, es el clásico semáforo, que permite el acceso en exclusión mutua a una hebra para modificar el valor de una variable compartida, manteniendo bloqueadas las demás (*locked*). Este mecanismo, también permite la sincronización de varias hebras, aunque `Pthreads` incluye un mecanismo más sofisticado, las variable condición, con las que podemos controlar si una hebra pasa a ejecutarse o no, dependiendo del valor de una de estas variables.

El uso de `Pthreads` sobre este proyecto, se plasma en un esqueleto común a todos los esquemas, sobre el que se construye la funcionalidad especifica de cada esquema. Este esqueleto, incluye las funciones básicas para la creación, pausa y reanudación del esquema, más todas las variables y estructuras que gobiernan su estado.

3.3.2 Desarrollo de interfaces gráficas con QT

La utilidad de un mecanismo que nos facilite la depuración del comportamiento que se va a desarrollar, introduce la necesidad de una interfaz gráfica de usuario. Ésta, nos permitirá la interacción con parte del software desarrollado de manera que podremos trazar fácilmente su funcionamiento. Por ello, se necesitará del uso de un toolkit para desarrollo de interfaces gráficas. De entre la diversas herramientas disponibles (X-Forms, Gtk+, Qt,...), se ha elegido Qt, dado que reúne todas las características necesarias (principalmente suspendible a antojo), y por encima de las demás, es la más fácil e intuitiva de utilizar.

Qt es una librería C++ que proporciona un toolkit para el desarrollo de interfaces gráficas multiplataforma [3]. Qt está totalmente orientada a objetos, es fácilmente extensible, y proporciona verdadera programación de componentes mediante el mecanismo **SIGNALS-SLOTS**. Qt es la librería nativa de KDE, el popular entorno de escritorio de Linux.

Una de las principales características de Qt, es sin duda, su orientación multiplataforma. De esta manera, podemos escribir aplicaciones portables a MS/Windows, Unix/X11, Mac y sistemas embebidos, con un mínimo de restricciones inherentes a las diferencias entre sistemas (nombres de ficheros, llamadas al sistema, concurrencia, ...). Qt es un producto de la compañía noruega **trolltech** (`www.trolltech.com`), y se distribuye en diferentes versiones, gratuitas (GPL) o comerciales. El modelo de objetos de C++ nos proporciona un soporte muy eficiente, en tiempo de ejecución, del paradig-

ma de POO[1]. Pero además de la eficiencia, a la hora de desarrollar GUI's[2] requerimos cierta flexibilidad y mecanismos de los que C++ carece.

Por ello, Qt añade ciertos mecanismos, que conllevan a un modelo de objetos más adecuado a nuestras necesidades.

En concreto añade:

- Mecanismo desacoplado para comunicación entre objetos(**SIGNALS-SLOTS**).
- Propiedades de objeto.
- Eventos, y filtros para eventos.
- Facilidades para la traducción de la GUI.
- Temporizadores.
- Jerarquía de objetos accesible en tiempo de ejecución.
- Punteros seguros.

Para conseguir estas nuevas características, Qt se basa en la herencia desde la clase `QObject`, utilizando técnicas estándar de C++, y su compilador de meta-objetos (moc). Así, el modelo de objetos de Qt adquiere características de un modelo basado en componentes (por el bajo acoplamiento entre sus partes), mucho más capacitado para el desarrollo de GUI's.

Para la creación de interfaces, QT incluye una potente herramienta de desarrollo visual, `qtdesigner`. Con esta herramienta, es posible crear la interfaz y programar su comportamiento, bien mediante código C++, bien mediante sus mecanismos visuales.

Sin duda, la principal característica de Qt, es su mecanismo de comunicación entre objetos **SIGNALS-SLOTS**. Este mecanismo, nos permite la interconexión de los eventos generados por un objeto (señales) a alguna funcionalidad de otro objeto (slot). Tanto las señales como los slots pueden ser los predefinidos en los objetos de Qt, o creados por el usuario. De esta manera, podemos generar una reacción simple o compleja de manera fácil. Por ejemplo, dada la interfaz de la figura 3.11, deseamos que el valor mostrado en el recuadro que esta en la parte superior aparezca en el display LCD inferior. Para ello, basta conectar la señal *QSpinBox::valueChanged(int v)* con el slot *QLCDNumber::display(int v)*. Al ejecutar dicha interfaz, cada vez que el valor

[1]Programación Orientada a Objetos
[2]Graphic User Interface

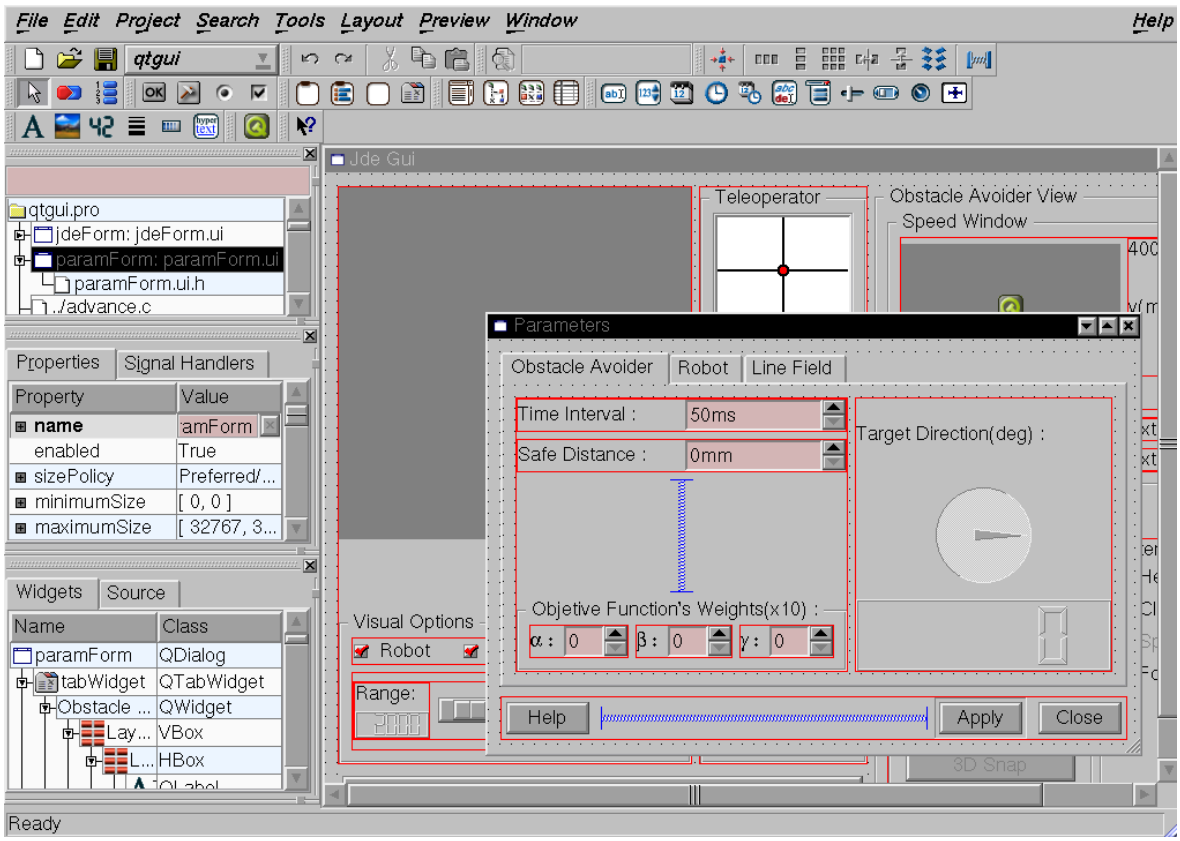

Figura 3.10: QtDesigner.

del recuadro de edición varíe, éste emitirá la señal valueChanged(int) con el valor modificado como parámetro, y el LCD la recibirá en el slot display(int) que pintará el valor pasado en el LCD.

Figura 3.11: Ejemplo Qt

Esta característica, nos permitirá construir interfaces con un nivel de acoplamiento entre sus partes muy bajo, ya que un objeto no conoce quién recibirá sus señales, ni quién se las envía a él.

Aunque Qt ofrece una gran facilidad para la creación de nuevos *widgets* (elementos gráficos *window gadget*), el amplio repertorio incluido en la distribución, permite la

construcción de interfaces muy ricas funcional y gráficamente hablando. Incluso es posible añadir funcionalidad mediante *plugins* de terceros, como los creados para la interacción con el servidor de bases de datos MySql.

Capítulo 4

Descripción Informática

Este capítulo aporta todos los detalles referentes a la implementación del método de navegación local con la ventana dinámica (DWA) sobre el robot Pioneer. En las siguientes secciones se expondrá la estructura del software realizado.

La primera sección, realiza una descripción detallada del método de la ventana dinámica. Las segunda, introduce el diseño estructural realizado, comentando la descomposición en esquemas JDE que se ha realizado, y las subsiguientes detallan cada esquema en particular, como son `linefield`,`obstAvoider` y `gui`.

4.1 Enfoque de Ventana Dinámica

El método de navegación local basado en la ventana dinámica (DWA) de Sebastian Thrun [10] es un derivado del método CVM (Curvature Velocity Method) desarrollado por Reid Simmons [16]. La característica principal de estos métodos, es la búsqueda en cada ciclo de control del comando motor que guiará al robot en el siguiente instante, directamente sobre el espacio de velocidades. En este espacio de ejes x-velocidad de traslación, y-velocidad de rotación nuestra velocidad actual esta representada como un punto que permanecerá invariable hasta que cambiemos nuestra velocidad. El análisis se realiza recorriendo el espacio de velocidades, encontrando para cada ciclo la velocidad que se ajusta mejor a la navegación deseada. Sobre esta directriz, cada uno de los métodos derivados utiliza diferentes criterios para la búsqueda en el espacio de velocidades.

El enfoque de la ventana dinámica, realiza la búsqueda del comando motor sobre un subconjunto del espacio de velocidades denominado ventana dinámica. Este subconjunto, surge al tener en cuenta las limitaciones dinámicas del robot, es decir, limita

el espacio de búsqueda a aquellas velocidades que dada la velocidad actual y la aceleración máxima del robot, serán alcanzables en un determinado intervalo de tiempo. Sobre este subconjunto, se busca aquella velocidad que maximiza una función objetivo que sopesa su bondad, aplicando varios criterios, que en esencia valoran una trayectoria segura y sin colisiones y con una velocidad en la dirección de avance alta. Aplicada la velocidad óptima hallada, se repite todo el proceso es sucesivas iteraciones de control. Con ello, obtenemos un comportamiento reactivo, sensible a las condiciones del entorno, y orientado a un objetivo.

La situación imaginaria de la figura 4.1 nos servirá de ejemplo en la descripción detallada que viene a continuación. En ella podemos ver al robot atravesando un pasillo con puertas a los lados, y un punto objetivo al que éste debe llegar.

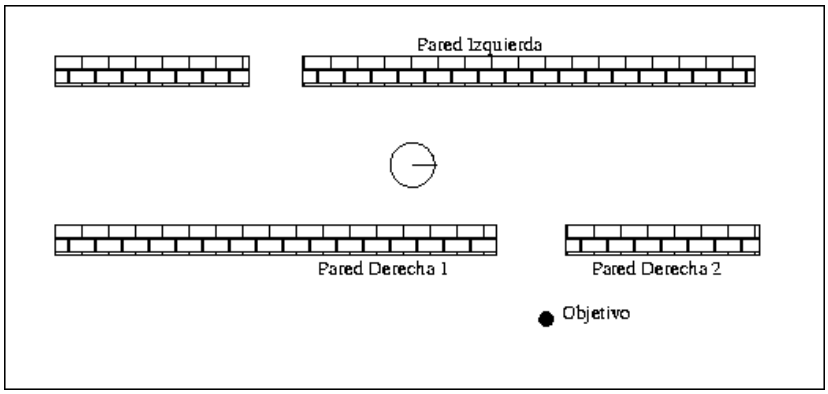

Figura 4.1: Situación imaginaria.

El espacio de búsqueda, como se ha mencionado, está restringido por las limitaciones dinámicas inherentes al robot, pero además a modo de simplificación sólo se tendrán en cuenta las trayectorias circulares definidas por el par v-w (v = velocidad de traslación, w = velocidad de rotación) con radio $\frac{v}{w}$. Esta simplificación, primero, agiliza los cálculos sobre las trayectorias que seguirá el robot, y segundo, limita a dos dimensiones (v y w) el espacio de búsqueda, con el consiguiente ahorro computacional. Además, dicha simplificación se ajusta bastante bien a la realidad puesto que, la variación de velocidad entre iteraciones es muy pequeña (dado que el tiempo entre éllas es muy pequeño), y suponemos que la velocidad permanecerá constante si no se especifica lo contrario. Así, la trayectoria que seguirá nuestro robot, será un conjunto de arcos de circunferencia. En la figura 4.2 podemos ver un ejemplo.

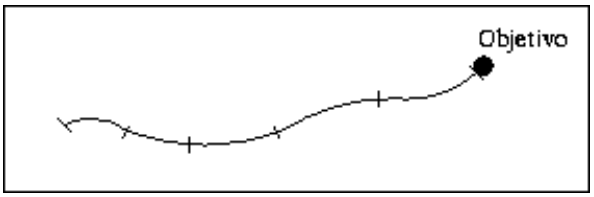

Figura 4.2: Una trayectoria.

Con esto, podemos definir el subconjunto denominado ventana dinámica, como:

$$V_d = \{(v, w) | v \in [v_a - \dot{v} \cdot t, v_a + \dot{v} \cdot t] \wedge w \in [w_a - \dot{w} \cdot t, w_a + \dot{w} \cdot t]\} \qquad (4.1)$$

, donde $\dot{v}$ y $\dot{w}$ son las aceleraciones de traslación y rotación, respectivamente, v_a y w_a definen la velocidad actual, y t es el tiempo que se aplican las aceleraciones.

Una vez calculado este subconjunto de velocidades, nos quedaremos sólo con aquellos pares (v,w) que sean *admisibles*, es decir, aquellas que si fuesen aplicadas permitirían detener al robot antes de chocar con el obstáculo más cercano en la trayectoria que define. En la figura 4.3 vemos el espacio de velocidades completo que obtenemos al analizar el supuesto de la figura 4.1. En él observamos zonas sombreadas, que representan las velocidades *no admisibles*. Sobre la figura, se han señalado los obstáculos que producen dichas zonas.

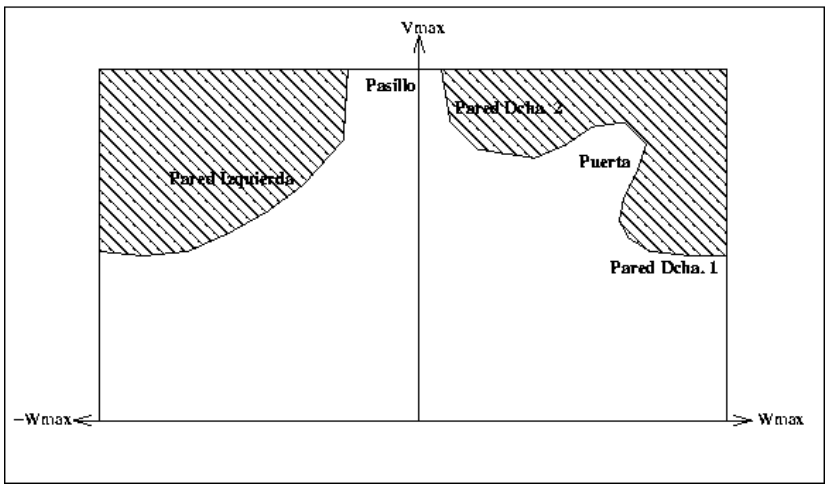

Figura 4.3: Espacio de velocidades.

El criterio para decidir si una velocidad es admisible o no, consiste en saber si al aplicar la velocidad (v,w), el robot será capaz a posteriori de detenerse antes de toparse con el obstáculo más próximo. Para ello, aplicaremos la deceleración máxima sobre el

robot y calcularemos la distancia recorrida, la cual cotejaremos con la distancia mínima a un obstáculo de la trayectoria seguida.

La figura 4.4 muestra el análisis del espacio de velocidades V_s, sobre el que distinguimos los subconjuntos V_a de velocidades admisibles, V_d o ventana dinámica y $V_r = V_a \cap V_d$ de velocidades posibles para el siguiente intervalo.

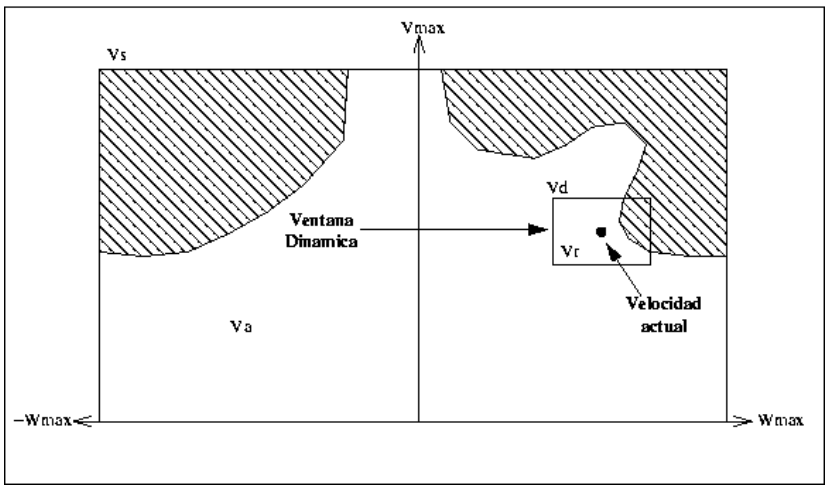

Figura 4.4: Ventana dinámica.

Una vez obtenido el subconjunto de velocidades candidatas V_r contenido en la ventana dinámica, el algoritmo busca aquel par(v,w) que maximiza una función objetivo, que valorará la bondad de dicho par frente a los siguientes criterios:

1. Mejor orientación final al objetivo.
2. Mínima proximidad a un obstáculo.
3. Máximo avance al objetivo.

Dichos criterios se combinan en la citada función objetivo, que se define como:

$$G(v, w) = \alpha \cdot Heading(v, w) + \beta \cdot Clearance(v, w) + \gamma \cdot Speed(v, w) \qquad (4.2)$$

donde α, β, γ son los pesos de los criterios, y *Heading*, *Clearance* y *Speed* son los criterios 1,2 y 3 comentados anteriormente.

Tras encontrar el par (v,w) óptimo, se comanda y finaliza la iteración, repitiendo todo el proceso descrito en cada ciclo de control.

La adecuación del algoritmo, se fundamenta principalmente en los pesos aplicados a cada uno de los criterios de la función objetivo, modificando en gran medida el comportamiento descrito por el robot. Con esto se quiere remarcar que dichos valores son determinantes en el éxito de este comportamiento y por ello requerirán de un ajuste fino y sistemático hasta hallar el conjunto de parámetros óptimo que materialice el comportamiento buscado.

4.2 Ventana Dinámica en esquemas JDE

El problema planteado, como se ha expuesto ya, es el desarrollo de un comportamiento de navegación local capaz de seguir una dirección estipulada o ir a un punto marcado, esquivando si fuera necesario los obstáculos que se interpongan es su trayectoria. Para ello, vamos a utilizar el algoritmo de ventana dinámica, de Sebastian Thrun [10], implementándolo siguiendo la arquitectura de control JDE, basada en la interacción de diferentes esquemas, perceptivos y motores, para la consecución de determinado objetivo. Por ello debemos encontrar la descomposición de nuestro problema en diferentes esquemas, perceptivos y motores, de manera que juntos, sobre JDE sean capaces de resolver el comportamiento deseado. Nuestro diseño del enfoque de ventana dinámica sobre JDE, se divide en dos esquemas: `linefield` y `obstAvoider`.

En primer lugar, como esquema perceptivo, necesitamos una perspectiva local del entorno algo más rica que la obtenida por la lectura directa de los sonares instantáneos. Por ello, `linefield` modelará el entorno, construyendo lo que se describe en [10] como campo de líneas.

En cuanto a los, esquemas motores, sólo necesitaremos diseñar uno, y será el encargado de utilizar toda la información sensorial recabada por el esquema perceptivo. Este realiza los cálculos necesarios sobre los datos sensoriales, a fin de conseguir los comandos motores que guiarán al robot Pioneer. Dicho esquema se denomina `obstAvoider`.

Con estos esquemas, más los básicos incluidos en el software JDE utilizado(`motors` y `sensationsotos`), somos capaces de generar el comportamiento deseado. La depuración así es prácticamente imposible, por lo que, se incorpora un esquema más, que facilitará esta tarea. Dicho esquema, no se ajusta a ninguno de los tipos descritos (perceptivos y motores), sino que es un esquema de servicio que aplicaremos a la depuración de los esquemas diseñados. Así, el esquema que denominaremos `gui` (Graphical User Interface), nos presenta una interfaz gráfica de usuario con la que podremos inte-

ractuar con el resto de esquemas, variando sus parámetros o su estado actual (activo o dormido).

Con la estructura comentada, la jerarquía de esquemas que genera el comportamiento queda como se muestra en la figura 4.5. En funcionamiento real, el esquema `gui` no está activo, para ahorrar recursos computacionales.

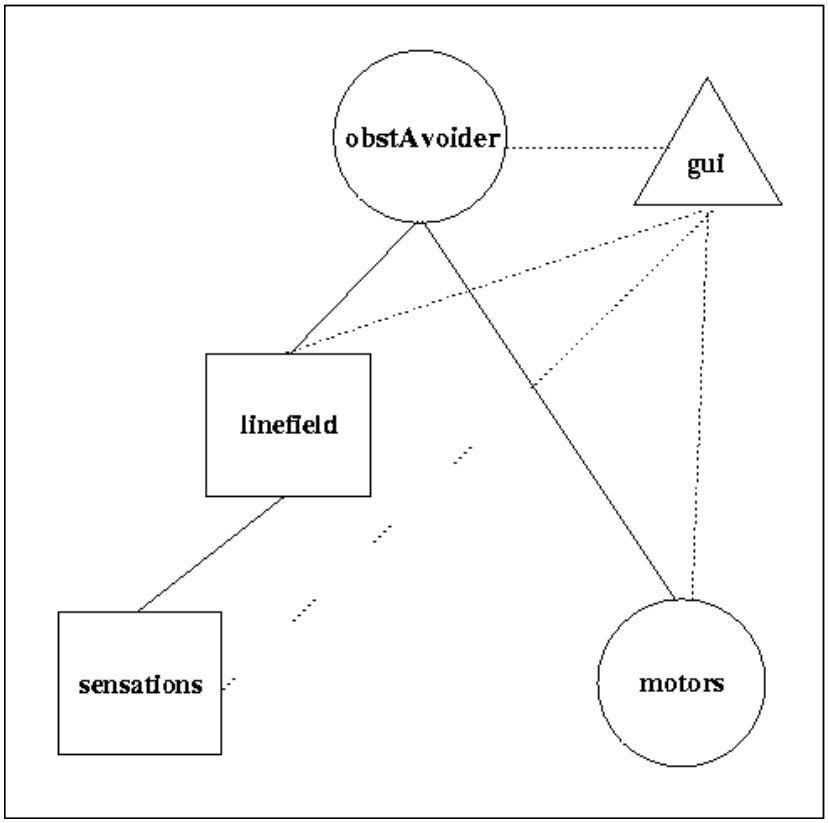

Figura 4.5: Estructura de la jerarquía.

Todo este diseño conceptual, se ha materializado desde el punto de vista informático en un programa con varias hebras de kernel, que siguen la estructura de la plataforma software JDE. En concreto, serán las citadas hebras posix (pthreads [2]), para las que el sistema operativo linux ofrece soporte. La implementación de JDE ofrece un esqueleto para la incorporación de nuevos esquemas, que define la interfaz mínima que requiere un esquema JDE. Así, cada esquema contiene funciones para la creación, pausa y reanudación, por ello, estos detalles se obviarán, señalando sólo aquellos propios de cada esquema.

La estructura que se ha seguido en las siguientes secciones, incluye una descripción

de las entradas y salidas del esquema, una descripción formal de su funcionamiento interno, y una descripción de los detalles más significativos de la implementación.

4.3 Esquema linefield

El esquema perceptivo `linefield`, está encargado de crear la representación del espacio local circundante al robot, que utilizará nuestro algoritmo. Está apoyado directamente sobre el esquema básico`sensationsotos`, del cual obtiene continuamente las mediciones de los ultrasonidos y la posición del robot a través encoders. En concreto, el mundo local creado por `linefield` es una descripción en dos dimensiones de la información sensorial en coordenadas absolutas. Cada medición de los sonares se resume en una línea normal al eje del haz del sonar. Dicha representación almacena tres instantáneas, la actual y dos anteriores. Cada instantánea, se compone de dieciséis lineas, correspondientes a cada una de las lecturas de los sonares (recordemos que el robot Pioneer incorpora una corona con dieciséis sensores de ultrasonidos). Así, manteniendo las dos últimas instantáneas más la actual, conseguimos una pequeña memoria a corto plazo, que hará un poco más robusta la representación frente a defectos puntuales en los datos, mientras que se mantienen valores muy actualizados. La figura 4.6 muestra un ejemplo del modelo construido por `linefield`, a su lado podemos ver el entorno que lo genera.

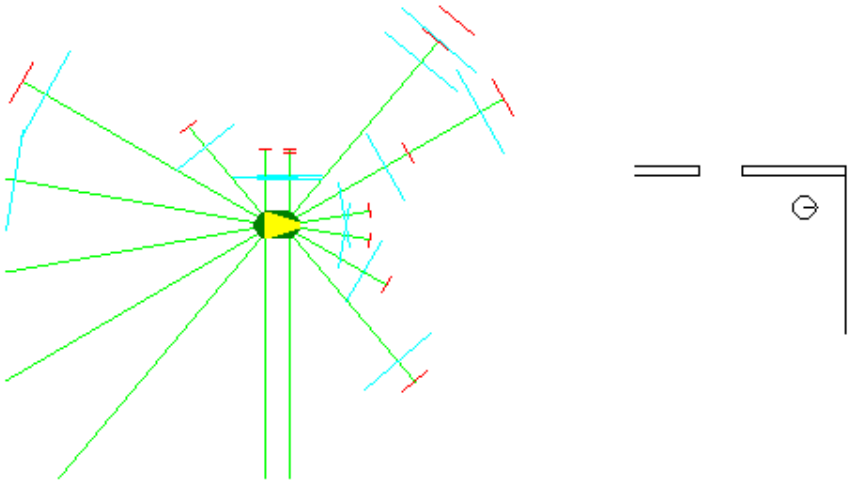

Figura 4.6: Ejemplo de linefield.

A medida que nos alejamos del origen del sensor, la apertura del haz es mayor.

Como se explicó en el capitulo 3, la zona abarcada por el sensor sonar se expande como un cono, por lo que a medida que nos alejamos del origen del sensor, mayor es la zona abarcada. Para el Pioneer se utiliza un haz con un ángulo de 10º. Así, una medición con valor d del sonar nos señala que existe un obstáculo *en algún lugar* del segmento perpendicular al haz que está a una distancia d del origen del sensor.

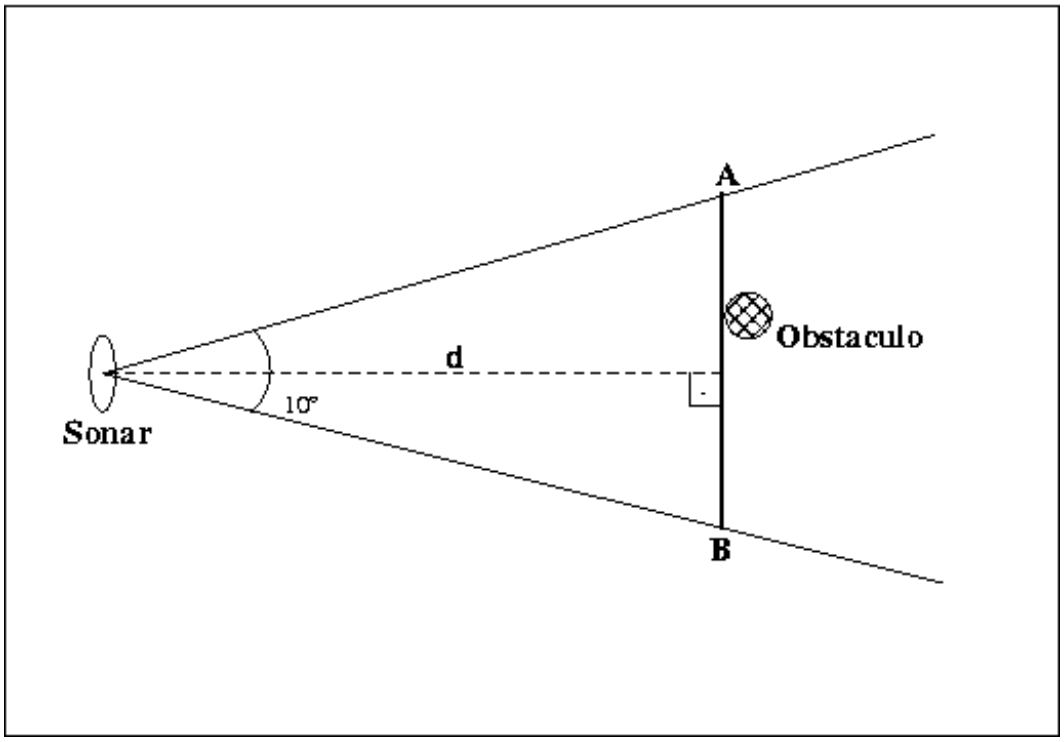

Figura 4.7: Linea calculada con el valor d del sonar.

Utilizando las relaciones trigonométricas:

$$\begin{cases} A_x = Sonar_x + d \\ A_y = Sonar_y + (\tan\frac{10}{2} \cdot d) \end{cases} \quad \begin{cases} B_x = Sonar_x + d \\ B_y = Sonar_y - (\tan\frac{10}{2} \cdot d) \end{cases} \tag{4.3}$$

calculamos de manera sencilla las coordenadas de los extremos del segmento con respecto a las coordenadas del sensor. Dicho par de valores representan al segmento y nos permitirán hacer cálculos sobre él. Puesto que nuestro modelo esta referenciado con coordenadas absolutas, será necesario aplicar los cambios de sistema de referencia necesarios (sensor-robot, robot-sistema absoluto).

Hasta ahora, hemos hablado del robot, como un objeto con cuerpo (aproximadamente una circunferencia de 25cm de radio) dentro de nuestro mundo. Este plan-

teamiento, complica todos los cálculos, por lo que con el afán de simplificar dichos cálculos, se aplica una modificación al modelo. Dicha modificación, consiste en considerar a nuestro robot como un punto dentro del modelo. Así, el cálculo de trayectorias y el rastreo en busca de obstáculos es mucho más sencillo. Para que esto sea valido, los obstáculos detectados se *"ensanchan"* el tamaño que ha *"adelgazado"* nuestro robot. Es decir, si nuestro robot se hace más pequeño, el resto de objetos deben hacerse más grandes para que se pueda seguir calculando correctamente la existencia o no de colisiones. Para ser exactos, los obstáculos se *"ensanchan"* el radio del robot más una distancia de seguridad que impida que nuestro robot se acerque demasiado a algún objeto.

Con esta simplificación, y aplicando una sencilla transformación a la línea calculada anteriormente, a partir de ahora *real_line*, obtenemos una línea que tiene en cuenta la simplificación realizada. Esta la denominaremos *safe_line*. Una *safe_line* es al robot

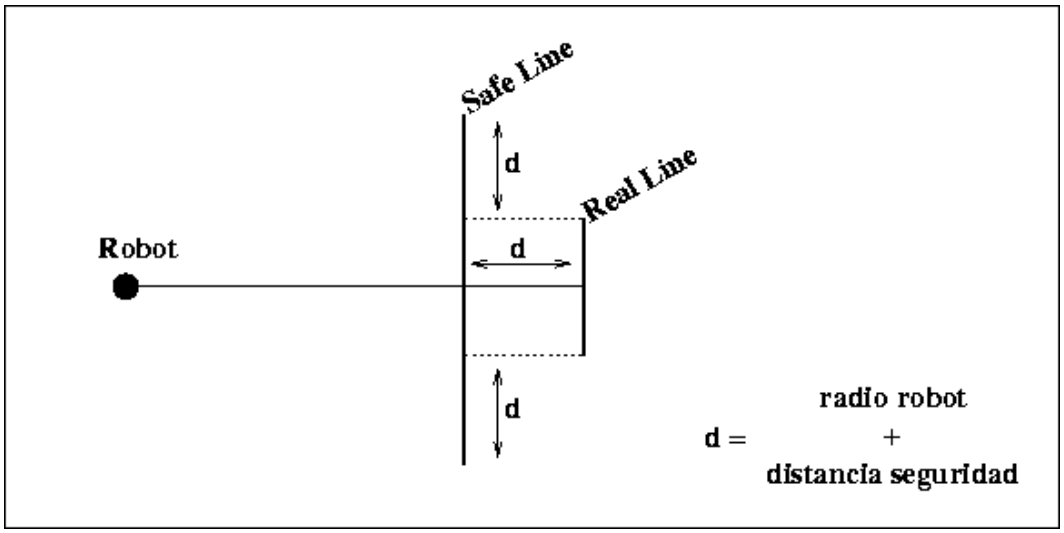

Figura 4.8: Safe line y real line.

puntual (considerado como un punto), lo que una *real_line* es al robot real.

Aunque a la hora de realizar cálculos, sólo se utilizan las *safe_line*, nuestra estructura de datos almacena también las *real_line* asociadas por motivos de depuración. Nuestra estructura de datos es una cola FIFO con un límite. A medida que se van introduciendo nuevas instantáneas, las más antiguas se van desechando. Como se comentó, cada instantánea se compone de dieciséis mediciones, por lo que la estructura tiene una capacidad de $3\ instantaneas \cdot 16\ mediciones = 48\ registros$.

La estructura de datos se define en el fichero linefield.h, que incluye las declaraciones mostradas en la tabla 4.1. La variable global `world`, como se ha comentado, mantiene el

Declaración	**Descripción**
world	Variable global que mantendrá el modelo local del espacio.
linefield_cicle	Entero que modula la duración de una iteración del esquema linefield.
beamBreadth	Entero que modula el ángulo del haz del ultrasonido.
maxDist	Entero que fija la distancia máxima que puede medir el sensor de ultrasonidos.
usError	Real que permite aplicarle un error a la medida obtenida del sensor de ultrasonido.
TLine	Tipo que define una linea en el espacio. En concreto almacena los puntos de corte con el haz. Véase la figura 4.7
TLineField	Tipo que define el modelo local del espacio. Como se ha comentado es una cola con una capacidad de tres instantáneas.
createLineField()	Función para crear un modelo de lineas. Devuelve un puntero a una estructura TLineField correctamente inicializada.
resetLineField(TLineField* lf)	Función para resetear un modelo de lineas. Lo inicializa destruyendo la información almacenada.
insertLine(TLineField* lf, TLine* realObst, TLine* safeObst)	Función para la inserción manual de lineas en el modelo. Debe usarse con cuidado, ya que puede introducir incoherencias en el modelo.
insertLine(TLineField* lf, int usN)	Función para la inserción automática de lineas en el modelo. Dado el número de un sonar se calculan *real_line* y *safe_line*.
newLine(float _p1x, float _p1y, float _p2x, float _p2y)	Función para crear lineas. Devuelve una estructura TLine.
copyLine(TLine* origL, TLine* destL)	Función para la copia completa de lineas. Reserva memoria para la copia.

Cuadro 4.1: Declaraciones de linefield.h

modelo captado del entorno, es una estructura de datos de tipo `TLineField` y se maneja con las operaciones aportadas. Dado su ámbito, es visible para los demás esquemas, que la leerán para extraer su información continuamente actualizada. El grupo de las cuatro siguientes declaraciones conforma el conjunto de parámetros del esquema, a través del cual se modulará su comportamiento. El parámetro principal es la duración del ciclo del esquema, fijado por `linefield_cicle`. Este valor estará entre los 300 y los 500 ms, produciendo de dos a tres iteraciones por segundo. El resto de parámetros modulan el comportamiento de los ultrasonidos.

Como esquema JDE, éste se implementa como un flujo de ejecución iterativo. Para ello, y como declaración privada (en linefield.c) tenemos la función `linefield_iteration()`. Esta engloba las acciones que se realizan en cada iteración y es ejecutada periódicamente. Dichas acciones son simplemente creación de una instantánea, e introducción de la instantánea en el modelo.

4.4 Esquema obstAvoider

El esquema motor `obstAvoider`, es el encargado de generar los comandos motores que harán moverse al robot Pioneer. Así, es el consumidor de las percepciones generadas por el esquema `linefield`,y aplica un análisis ad-hoc de la información, a fin de conseguir que dichos comandos motores generen el comportamiento deseado en este proyecto. El esquema `obstAvoider` construye la estructura que da nombre al método de navegación local, la ventana dinámica.

Por tanto el principal objetivo de este esquema, es el cálculo de la ventana dinámica. Dicha estructura, es una matriz bidimensional que mantiene los valores de la función objetivo para cada par (v,w), del subespacio que engloba la ventana. El tamaño de esta matriz limita la precisión de los cálculos por una parte, e influye en el tiempo de cómputo por otra. En cuanto a precisión, influirá en la mayor o menor riqueza de trayectorias posibles en un estado determinado, aunque ésta también estará dada por la precisión de los motores del robot. Por ello existe un compromiso entre precisión y ciclo de iteración, es decir, a más precisión mayor coste computacional, puesto que será necesario analizar muchas más trayectorias. Dado que un requisito de este proyecto, es conseguir una alta reactividad del robot, se debe encontrar un valor de compromiso adecuado. Los experimentos realizados, indican que dimensiones de 30x30 a 50x50 permiten el cálculo de hasta cuatro iteraciones por segundo, más que suficientes para nuestro propósito. Hablando de números, una matriz de 30x30 requiere de 43.200 cálculos de trayectoria, puesto que cada celda analiza los 48 obstáculos obtenidos por *linefield*, y una de 50x50 requiere de 120.000 cálculos. La precisión en cuanto a unidades de velocidad, dependerá de la aceleración del robot, y como ejemplo sobre una matriz de 30x30, con una velocidad de traslación actual de 100 mm/s, una aceleración de 50 mm/s^2 y un intervalo de tiempo de 200 ms, analizaremos velocidades desde los 90 mm/s a los 110 mm/s en intervalos de 0.66 mm/s.

El algoritmo utilizado es el mismo que el apuntado en la sección de descripción de DWA, aunque aplica alguna modificación, en pos de una simplificación en la implementación. El algoritmo sigue los pasos:

1. Calculamos los límites de la ventana dinámica.
2. Obtenemos todas las velocidades admisibles que caen dentro de la ventana dinámica.

3. Calculamos el valor de la función objetivo para cada velocidad admisible.
4. Tras haber recorrido la ventana dinámica aplicamos la velocidad (v,w) que obtuvo mejor puntuación.

Este proceso iterativo se repetirá de modo continuo para conseguir el objetivo marcado, bien seguir una dirección fija marcada, bien llegar a un punto objetivo cercano.

Los principales cálculos de este algoritmo, son la determinación de si una velocidad es admisible o no, y el cálculo de la función objetivo. Para el primer cálculo, debemos ver si el par (v,w) produce una trayectoria segura para el robot. El criterio original descrito en la sección anterior, introduce algunos cálculos complejas, puesto que debemos tratar con las trayectorias curvilíneas producidas por el robot en su frenada, que no pueden ser aproximadas por arcos circulares sin cometer un error mas que considerable. La figura **??** muestra lo comentado gráficamente. T_c representa la trayectoria

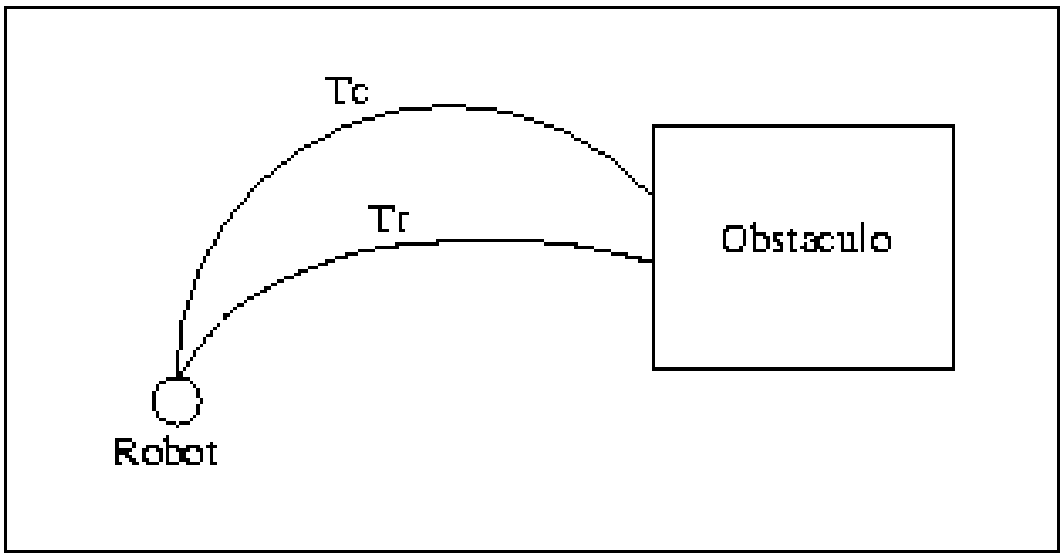

Figura 4.9: Trayectoria en frenada.

circular y T_r la trayectoria seguida realmente. Por ello se ha optado por relajar el criterio, considerando que una velocidad es admisible si es posible mantenerla durante X intervalos sin colisionar. De esta manera, la trayectoria a estudiar sí es circular, y por tanto requiere de cálculos más ligeros. El valor de X determina la actitud de nuestro robot ante trayectorias con obstáculos cercanos. A valores altos, su actitud será más

conservativa, y evitará zonas muy pobladas, y a valores bajos se arriesgará más. El valor más adecuado, obtenido experimentalmente es 20, que produce un comportamiento *"extrovertido"* pero sin excesivos riesgos.

Dado que las trayectorias descritas son circulares, el problema se reduce al cálculo de intersecciones entre rectas (generadas por los obstáculos) y círculos (trayectorias estimadas). Una vez calculada la intersección al obstáculo podemos deducir el radio y ángulo del arco que sigue la trayectoria, con lo que aplicando la formula $arco = R \cdot \alpha$ obtenemos la distancia de choque. Si esta distancia es mayor a la que el robot recorre, a la velocidad analizada, en X intervalos, diremos que es una trayectoria segura, y por tanto la velocidad que la produce es admisible.

El otro cálculo importante del algoritmo, es la función objetivo, que basándose en ciertos criterios, devuelve la puntuación de un par (v,w). La función objetivo implementada es la siguiente:

$$G(v,w) = \alpha \cdot \left(\frac{\pi - \theta}{\theta} \cdot 0{,}7 + \left(1 - \frac{|w|}{w_{max}} \right) \cdot 0{,}3 \right) + \beta \cdot \left(\frac{d}{d_{max}} * 0{,}5 + g * 0{,}5 \right) + \gamma \cdot \frac{|v|}{v_{max}} \tag{4.4}$$

donde α, β, γ son los pesos de cada criterio, θ es el ángulo mínimo entre la orientación del robot y la dirección objetivo. d es la distancia al obstáculo más próximo en la trayectoria. d_{max} es la distancia máxima de una trayectoria. Y g representa la bondad de la trayectoria en función de su aproximación a los obstáculos.

El primer termino de la suma, es el criterio de orientación al objetivo. En él se valoran positivamente aquellas trayectorias que producen una buena orientación al final del intervalo, muy parecida a la deseada, y que requieren de una menor velocidad de rotación. Esto último confiere estabilidad a la trayectoria seguida por el robot, reduciendo las oscilaciones en base a la dirección objetivo. Este criterio alcanza su máximo para trayectorias totalmente orientadas al objetivo y sin giro, es decir, avanzando "sobre" la dirección objetivo. Este criterio, se divide en dos términos balanceados, el primero premia las orientaciones previstas mas parecidas a la orientación objetivo, y el segundo valora aquellas trayectorias que tienen una menor velocidad de rotación, que como se ha comentado eliminan las oscilaciones en la navegación producidas por la inercia del robot. Tal es así, que trayectorias con orientación final muy buena, pero con una alta velocidad de rotación son muy penalizadas, por que la inercia del robot, desviaría mucho la orientación en los siguientes intervalos.

El segundo término en la función objetivo valora la claridad de la trayectoria, es de-

cir, cuan lejos queda de un obstáculo interpuesto en el camino del robot. Este criterio nos permitirá valorar aquellas trayectorias que permanecen lo suficientemente alejadas de los obstáculos, produciendo así una navegación segura. El criterio alcanza su máximo para trayectorias en las que no existen obstáculos que produzca colisión en un margen determinado de metros. Como el criterio anterior, se divide en dos términos, el primero de los cuales valora la distancia a un obstáculo en la trayectoria, y el segundo aplica un factor de bondad calculado para esa trayectoria. Este factor de bondad valora positivamente aquellas trayectorias que están más alejadas de cualquier obstáculo.

El tercer término, valora las velocidades más altas, de manera que nuestro robot intentará alcanzarlas en la medida de lo posible, obteniendo velocidades medias de navegación elevadas. El criterio alcanza su pico cuando la velocidad es máxima (V_{max}).

La combinación de los tres criterios, nos permite valorar las velocidades admisibles que contiene la ventana dinámica, de manera que somos capaces de encontrar la mejor velocidad aplicable en cada iteración buscando aquella que maximiza nuestra función objetivo. Las mejores puntuaciones, las obtendrán aquellas velocidades que produzcan trayectorias orientadas al objetivo, mantengan alejado a nuestro robot de los obstáculos, y sean cercanas a la máxima velocidad de avance. La figura 4.10 muestra varias trayectorias de ejemplo. La número 1, obtiene la mejor valoración porque mantiene una buena orientación, evita el obstáculo y avanza a una velocidad alta. La número 2, obtiene una puntuación buena, aunque su acercamiento al obstáculo la descarta reduciendo su valoración final. Por último, la número 3 obtiene una valoración muy baja por culpa de la orientación que produciría, opuesta a la dirección objetivo.

La combinación de los criterios, se basa en los pesos que aplicamos a cada uno, de manera que pesos diferentes producen valoraciones muy diferentes, y por consiguiente comportamientos distintos. El valor asignado a estos pesos se ha obtenido mediante ensayo sobre el simulador, en un proceso de ajuste fino y manual, y puede variar en función de los entornos a recorrer. Como relación a destacar tenemos que el peso asignado al criterio de orientación al objetivo debe ser superior a los otros de manera que prime por encima de todo el avanzar hacia el destino marcado. Los otros dos pesos influirán en las velocidades alcanzadas y el acercamiento a los obstáculos.

El esquema `obstAvoider` se implementa en los ficheros obstAvoider.c y obstAvoider.h. En la tabla se muestran las declaraciones más destacables.

Como puede verse en la tabla 4.2, el esquema `obstAvoider` cuenta con un gran

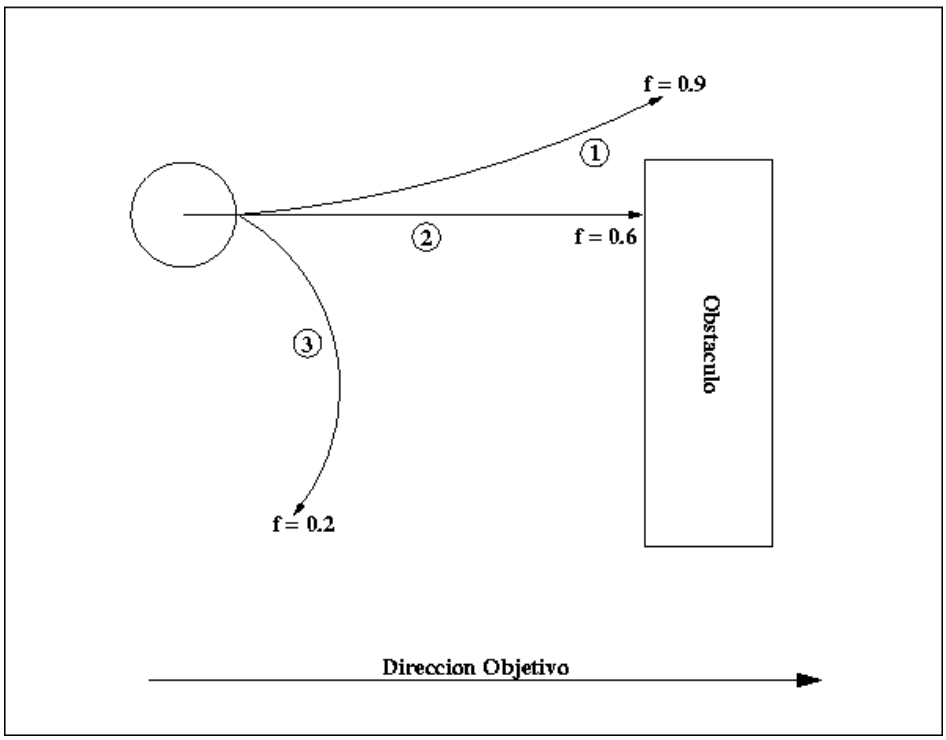

Figura 4.10: Ejemplo de trayectorias.

Declaración	**Descripción**
dynamicWindow[]	Matriz que contiene la información de la ventana dinámica.
targetDirection	Indica la dirección objetivo en radianes. EL valor 0 se identifica con la dirección Este.
targetPoint	Indica el punto objetivo en caso de utilizar este método de orientación.
aph,beta,gam	Reales que representan los pesos de los criterios de la función objetivo.
criteria	Mascara de bits que nos permite seleccionar los criterios que se aplican en la función objetivo.
tSpeeMin,tSpeedMax	Mínimo y máximo de la velocidad de traslación que se aplicará.
rSpeeMin,rSpeedMax	Mínimo y máximo de la velocidad de rotación que se aplicará.
tAcceleration, rAcceleration	Aceleraciones aplicadas a la velocidad actual para calcular los límites de la ventana dinámica.
dWTimeInterval	Intervalo de tiempo que utilizaremos para el cálculo de los límites de la ventana dinámica.
robotRadius	Indica el tamaño del robot.
safeDist	Distancia de seguridad aplicada para que el robot no se aproxime demasiado a los obstáculos.
obstAvoider_cicle	Intervalo de tiempo empleado para cada iteración.
ranking(float _v, float _w, float dir, float dist)	Obtiene el valor de la función objetivo para una determinada velocidad, dirección y distancia a un obstáculo.
apply_speed(float _v, float _w)	Aplica la velocidad obtenida al final de la iteración.
obstDist(float _v, float _w, TLine*)	Calcula la distancia mínima de una trayectoria definida por v y w hasta un obstáculo, si es que existe.
intersectionR_C(..)	Calcula la intersección entre una línea y un círculo.
intersectionR_R(..)	Calcula la intersección entre dos líneas.
angleVW(float v[2], float w[2])	Devuelve el ángulo más pequeño existente entre dos vectores v y w.
parametric2implicit(..)	Transforma la representación paramétrica de la recta en su ecuación implícita.

Cuadro 4.2: Declaraciones de obstAvoider

número de declaraciones, destacando entre ellas, la variable global `dynamicWindow[]` que representa a la ventana dinámica como una matriz. Su contenido será los valores obtenidos por la función objetivo para cada uno de sus pares, y será utilizado principalmente para labores de depuración en el esquema `gui`. De entre todos los parámetros, merecen especial atención los encargados de mantener la dirección objetivo, que en función del método escogido (ir a punto, seguir dirección) serán tenidos en cuenta o no. Destacar también aquellos que fijan los límites de velocidad, que nos permitirán mantener al robot en un cierto rango.

4.5 Esquema gui

El esquema `gui`, como ya se apuntó en la sección de diseño de la estructura, es un esquema de servicio. Su misión, es ayudarnos en la depuración del resto de esquemas, mostrándonos bien sus percepciones, bien sus actuaciones. También es capaz de modular el funcionamiento de un esquema variando sus parámetros de configuración sobre la marcha, característica permitida en JDE(modulación continua). Estas funciones se muestran a través de una interfaz gráfica de usuario. Esta interfaz se ha desarrollado con el toolkit gráfico Qt descrito en el capítulo 3. Su aspecto y distribución se muestra en la figura 4.11.

A continuación se describe cada área en concreto, con sus características y funcionalidades:

1 Área de visualización Este área, muestra la representación del robot en tiempo real. Es capaz de mostrar el movimiento del robot cuando este avanza o retrocede. Las lineas verdes representan los valores de los sonares. Sobre esta visualización, se muestra también el espacio de lineas generado por `linefield`(lineas azules = *safe_lines*, lineas rojas = *real_lines*), permitiendo depurar su funcionamiento y ajuste.

2 Área de teleoperación Este área permite teleoperar al robot Pioneer mediante el joystick de la parte superior. Su funcionamiento radica en el esquema básico `teleoperator` incluido en el software JDE y que permite la teleoperación en velocidad del robot Pioneer. El botón de *Stop* produce la parada inmediata del robot. Además, en este área se indica la velocidad comandada al robot en cada instante.

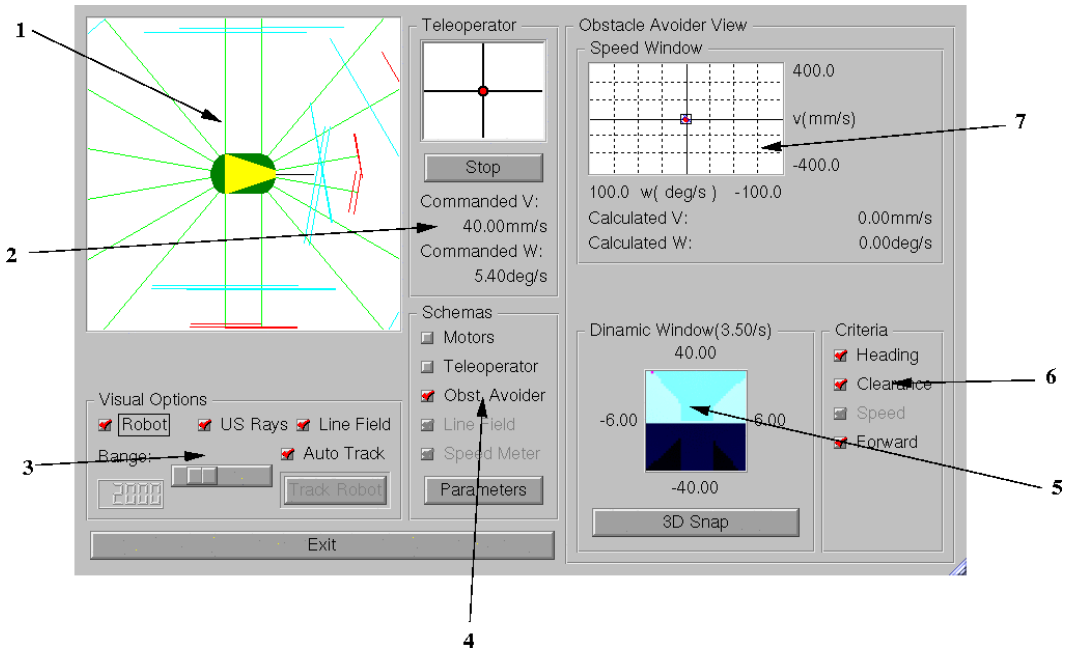

Figura 4.11: Interfaz gráfica del esquema gui.

3 Área de opciones de visualización Este área permite variar ciertas opciones de la visualización mostrada en el área 1. En concreto, permite ocultar/mostrar cualquiera de los objetos de la visualización, variar el rango de visualización y activar/desactivar la función *"auto track"*. Con la variación del rango variamos el escalado de los objetos, permitiendo mostrar más o menos espacio visual. La opción *"auto track"* permite que el robot sea centrado automáticamente cuando salga del campo visual.

4 Área de esquemas En este área, podemos activar o desactivar los diferentes esquemas y modificar sus parámetros a través del diálogo *parameters* mostrado en la figura 4.12

Área de ventana dinámica Este área muestra una representación de la ventana dinámica en tiempo real, basada en una escala de colores. El criterio seguido en la visualización, es que cuanto más claro sea el color mejor es la puntuación obtenida por el par(v,w) en la función objetivo. De modo especial se resalta el máximo con un pixel morado, que nos indica la velocidad escogida para esa iteración. A los lados de la ventana dinámica se muestran sus límites absolutos en mm/s

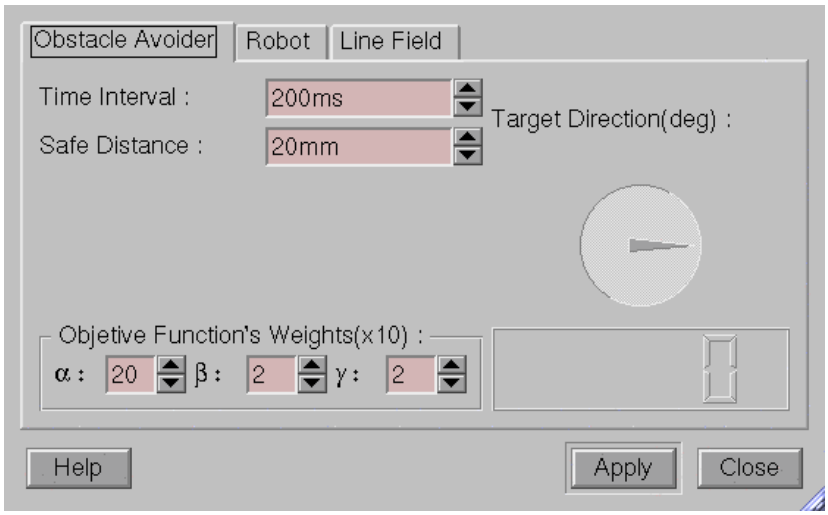

Figura 4.12: Dialogo de parámetros.

y *grados/s*. El botón *3D Snap* escribe una representación en 3D de la ventana dinámica y a través del programa `gnuplot` genera una instantánea tridimensional que nos permite estudiar valores concretos. La figura 4.13 muestra una captura. Este área es especialmente útil en la depuración de `obstAvoider`, ya que nos permite observar su funcionamiento en un instante concreto.

6 Área de criterios Este área permite seleccionar los criterios que queremos que se apliquen en la función objetivo y así observar su influencia en el cálculo de la ventana dinámica en tiempo real.

7 Área de espacio de velocidades El recuadro cuadriculado representa el espacio de velocidades completo. El cuadrado interior, que en la figura 4.11 se muestra en el centro, representa el subconjunto abarcado por la ventana dinámica. Este varía en función de los limites en cada momento. En este área también se muestra la velocidad calculada por los esquemas básicos de JDE.

Como hemos visto, el esquema `gui` nos muestra todos los detalles, que a la hora de la depuración nos pueden ayudar. Como se indicó, este esquema es un esquema de servicio, y tras la fase de depuración no será precisa su presencia en la jerarquía. Así, para el funcionamiento normal del comportamiento sobre el robot solo serán necesarios los esquemas `linefield` y `obstAvoider`.

Este esquema se ha implementado en los ficheros gui.h y gui.c, aunque la interfaz

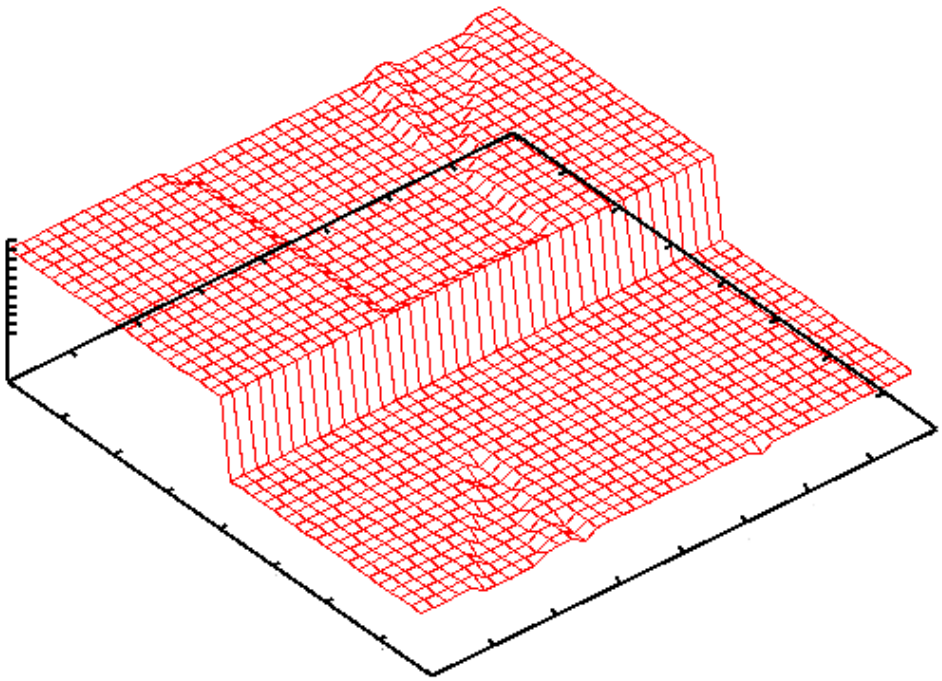

Figura 4.13: Captura de la función 3D Snap

como tal se define en los ficheros jdeForm.ui, jdeForm.ui.h, parametersForm.ui y parametersForm.ui.h. Todas las operaciones definidas se aplican al manejo de la interfaz, por lo que no se entrara en detalle.

4.6 Resultados experimentales

Hasta aquí, se han comentado todos los detalles referentes a la implementación del algoritmo de la ventana dinámica con esquemas JDE. En esta última sección, se comentan los resultados experimentales obtenidos y los puntos críticos encontrados a la hora de implementar el algoritmo.

A continuación, se muestran los resultados obtenidos en varios pasillos, con configuraciones diferentes. Fijada la velocidad máxima en $0.20\frac{m}{s}$ y una dirección objetivo Este, los resultados se muestran en la figura 4.14. Sobre ella podemos ver que los pasillos aumentan incrementalmente su dificultad, permitiéndonos una depuración de menor a mayor dificultad. El pasillo 1 está totalmente libre de obstáculos consiguiendo en general la velocidad máxima permitida. Además, independientemente de la posición de partida, más o menos próxima a la pared, el robot tiende a centrarse, cosa natural atendiendo al criterio de alejamiento de los obstáculos. El pasillo 2, incorpora

dos obstáculos bastante separados, sobre él, el robot tiende también a centrarse en el espacio libre, sorteando convenientemente los obstáculos. Y por último, en el pasillo 3, observamos como la trayectoria es más brusca por la existencia de obstáculos más cercanos, comportándose también correctamente. Cabe destacar, que los obstáculos son imprevistos para el robot, y que no los conoce de antemano, sino que reacciona ante ellos cuando los detecta.

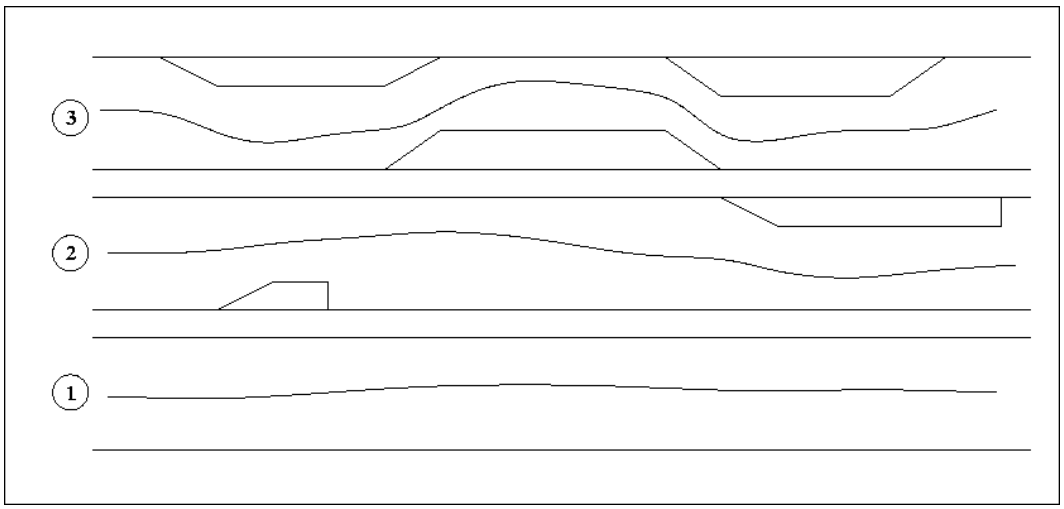

Figura 4.14: Experimentos sobre el simulador.

Las velocidades medias obtenidas en cada uno de los tres experimentos se muestran en la tabla 4.3, en ella podemos ver, que la velocidad disminuye a medida que aparecen más obstáculos. Esto refleja, que el algoritmo elige velocidades más prudentes cuando la densidad de obstáculos es mayor.

Experimento	**Velocidad Media**
1	0.19 m/s
2	0.15 m/s
3	0.10 m/s

Cuadro 4.3: Velocidades obtenidas

Con los primeros experimentos, apreciamos un punto crítico de este algoritmo, la necesidad de una estimación de la velocidad *muy fiable*. En una primera aproximación, se construyó un esquema encargado de esta tarea. El esquema `speedMeter`, calculaba la velocidad actual del robot a partir de la información de posición (x,y) y orientación (θ) en cada instante. Esta información de localización, era entregada por el servidor

`otos` sin ningún tipo de información temporal acerca de su obtención, es decir, la temporización recaía en los tiempos de llegada de los mensajes al cliente JDE. Dado que la comunicación entre `otos` y el cliente JDE se produce a través de una red TCP/IP, los retardos variables que en élla pudieran existir afectaban drásticamente a la exactitud en las estimaciones de velocidad. Así, en las primeras pruebas sobre la red *WireLess* que comunica el servidor `otos` con el cliente JDE, se vio la necesidad de introducir sellos temporales en los mensajes de localización con el tiempo de captura. Con ello, las estimaciones de velocidad son mucho más precisas, que las obtenidas anteriormente, porque las variaciones en los tiempos de trasmisión de los mensajes entre servidor y cliente, no afectan ahora a la estimación de velocidad, que se apoya en los tiempos de captura, y no en los de llegada de los mensajes.

Con el afán de evitar manejar datos de tan bajo nivel en los esquemas, como son los sellos temporales, se decidió colocar la estimación en el esquema básico `sensationsotos`. Éste es receptor directo de los mensajes de `otos`, y nos permite además el máximo ritmo de estimación posible, justo, a la llegada de los datos.

Por estas razones, el esquema `speedMeter` se desechó, consolidando la estimación en el esquema básico `sensationsotos` bajo las variables globales `tspeed` y `rspeed` (traslación y rotación respectivamente) utilizando los siguientes cálculos:

Para v:

$$|v| = \frac{\sqrt{(x_t - x_{t-1})^2 + (y_t - y_{t-1})^2}}{\Delta T} \tag{4.5}$$

, con signo del producto escalar del vector orientación y el vector de avance.

Para w:

$$w = \frac{\theta_t - \theta_{t-1}}{\Delta T} \tag{4.6}$$

, teniendo en cuenta la discontinuidad entre 0 y 2π.

Continuando con las pruebas, se detectó otro punto débil en el algoritmo, la actuación incierta y lenta de los motores. Se trata de la precisión en la obtención de una determinada velocidad comandada y del tiempo requerido para su obtención. En efecto, el algoritmo calcula un par (v,w) óptimo para el siguiente intervalo, y nuestro sistema motor debe ser capaz de alcanzarlo en el tiempo previsto. Se observo, que la velocidad obtenida era un 10 % inferior a la comandada, y que este error sistemático se

conservaba en todo el rango de velocidades. Para su solución , simplemente se compenso el error, aplicando velocidades un 10 % superiores a las deseadas. En cuanto al tiempo de reacción requerido por los motores, se observó que el robot simulado necesita de al menos intervalos de un segundo para alcanzar la velocidad deseada, y el robot real intervalos de hasta dos segundos. Por el momento, esta limitación es insalvable, y nos obliga por el momento a descartar la implantación del comportamiento sobre el robot real, dada la baja reactividad que nos confiere un intervalo de actuación tan grande. Por ello, el funcionamiento del comportamiento aquí descrito, sólo se ha conseguido sobre el robot simulado y con velocidades moderadas, que permiten una navegación fiable.

Capítulo 5

Conclusiones y trabajos futuros

Este capítulo presenta las conclusiones obtenidas tras el desarrollo de este proyecto, comentando de manera escueta los puntos relevantes que han permitido el desarrollo del trabajo, y las principales dificultades encontradas. También se comentarán posibles trabajos futuros en pos de la mejora y ampliación del comportamiento aquí descrito.

5.1 Conclusiones

Como conclusión principal extraemos, que se ha logrado la implementación, con esquemas JDE, del comportamiento de navegación local de la ventana dinámica descrito en [10]. Dicho comportamiento, reacciona de manera adecuada, en tiempo real, permitiendo la navegación segura a velocidades en torno a la máxima permitida, 0.20 m/s (a un ritmo de 1 iteración/s). Todo el código fuente del desarrollo, se encuentra disponible en la web del grupo *http://gsyc.escet.urjc.es/robotica* bajo licencia GPL. Por último indicar, que dadas las limitaciones encontradas solo ha sido posible su implantación sobre la plataforma simulada.

Su diseño, como se esperaba, se ajusta a la arquitectura JDE, desglosando el comportamiento, en dos esquemas. Uno perceptivo, *linefield* que construye una representación del entorno local al robot, basada en líneas, a partir de la información de los sensores sonar. Y otro de control *obstAvoider* encargado de generar los comandos motores aplicando el algoritmo de la ventana dinámica. Este diseño permitirá la reutilización de cada uno de los esquemas, bien por separado, bien como subconjunto de comportamientos más complejos construidos sobre éste.

En cuanto a la implementación, los esquemas desarrollados se ajustan y explotan las características de la plataforma móvil Pioneer simulada. En concreto, se ha aprovechado

la capacidad holonómica, permitiendo a nuestro robot movimientos tanto de avance como de retroceso. Dicha característica se incorpora como añadido al método original descrito en [10]. Por otra parte, y también como añadido se incorporan dos métodos de guiado diferentes y seleccionables según el caso. Uno, el método original, basado en una dirección objetivo y otro basado en un punto objetivo.

Como principales dificultades, se han identificado, por una parte la gran incertidumbre introducida por el sistema sonar, al no haberse aplicado ninguna técnica de filtrado sobre estos. En concreto, se han encontrado muchos problemas en la detección de obstáculos con cierta inclinación con respecto a los sensores, que producen medidas erróneas sobre su distancia. Por ejemplo, la detección de una esquina con este método es prácticamente imposible (véase la figura 4.6), debido a los efectos de rebote del sonar. Del mismo modo la detección de huecos relativamente pequeños depende de la orientación del robot en ese instante.

Otro gran inconveniente, motivo por el cual, sólo se ha podido aplicar el comportamiento sobre el simulador, es la dificultad de actuación encontrada, debida principalmente al gran retraso en la obtención de la velocidad comandada, un segundo en el simulador (hasta dos segundos en el robot real), limitando la reactividad del comportamiento, y debiendo fijar velocidades máximas no muy altas. También, se tuvieron dificultades en la obtención de la velocidad comandada, que no se acercaba nunca a la prevista, aplicando para su solución una rectificación sobre la velocidad comandada del 10 %, obtenida experimentalmente. Y por último, se ha detectado la necesidad de una estimación muy fiable y precisa, punto solucionado en parte, con la mejora del protocolo de comunicación entre el servidor `otos` y su cliente, `sensationsotos`, esquema básico de JDE, incluyendo sellos temporales con el tiempo de captura en los datos e posición. Esto ha mejorado notablemente la precisión en la estimación de velocidad.

Por otra parte, la extremada sensibilidad del comportamiento a los parámetros que incorpora la función objetivo, han hecho especialmente difícil su ajuste. Por ello, se ha realizado un ajuste fino manual, a base de experimentación. Como problema menor, la potencia de cálculo requerida es alta, por lo que se han determinado unos límites (50x50) que permiten el cálculo de tres a cuatro instantáneas por segundo, más que suficientes para nuestro caso.

5.2 Trabajos futuros

Las principales lineas de trabajo futuro apuntan a la resolución de los problemas detectados. Entre éllas, se maneja la posibilidad de añadir una representación del entorno mucho más robusta que la actual, basada en rejillas de ocupación, que aportarán una mayor fiabilidad a la navegación, o incluso extraída de un sensor mas fiable como un láser.

Con el afán de lograr una actuación más rápida y reducir el intervalo de actuación del robot real, se buscará un ajuste adecuado para hacer posible la implantación de este método de navegación. Este ajuste, se basará en la búsqueda de las constantes adecuadas que gobiernan los bucles PID que manejan los motores, y de los cuales depende el comportamiento del robot.

Se explorarán otros métodos de estimación de la velocidad instantánea, que nos aporten la precisión deseada para nuestro algoritmo, extrayendo, por ejemplo la información directamente de los cuenta vueltas. Este método, nos permitiría unos cálculos mucho más exactos, al calcular la verdadera longitud de la trayectoria recorrida, y no la secante de ésta, obtenida con las coordenadas de posición usadas actualmente.

En cuanto a mejoras en la eficiencia, otra línea de trabajo sería la optimización de las funciones de cálculo de trayectorias, apoyándolas en una librería matemática, que reduzca los tiempos de cálculo empleados.

Y por último, como ya se comentó, se espera en un futuro próximo la migración al software Aria [1], nuevo estándar de programación del robot Pioneer que sustituye a Saphira [12], de manera que todo el software utilizado por el grupo de robótica de la URJC cuente con licencia GPL.

Bibliografía

[1] Aria reference manual. http://robots.activmedia.com/aria/Aria-Reference.pdf.

[2] Posix threads programming. http://www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/MAIN.html.

[3] Qt reference documentation. http://doc.trolltech.com/3.0/.

[4] Carlos E. Agüero. Protocolo de comunicaciones para el robot eyebot. Proyecto fin de carrera, Universidad Rey Juan Carlos, 2002.

[5] Carlos Balaguer y Rafael Aracil Antonio Barrientos, Luis Felipe Peñin. *Fundamentos de Robótica*. McGraw-Hill/Interamericana de España,S.A.U., 1999.

[6] Javier Albertos Benayas. Control reactivo para robots móviles en entornos no modelados. Proyecto fin de carrera, Escuela Técnica Superior de Ingenieros Industriales y Minas de Vigo, 2001.

[7] José M. Cañas. Manual del robot móvil hermes. Technical report, Instituto de Automática Industrial CSIC, 2001.

[8] José M. Cañas and Vicente Matellán. Dynamic schema hierarchies for an autonomous robot. In *VIII Iberoamerican Conference on Artificial Intelligence IBERAMIA 2002*, 2002.

[9] Karel Capek. *Rossum's Universal Robots*. F.R. Borový, 1935.

[10] Dieter Fox, Wolfram Burgard, and Sebastian Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics and Automation Magazine*, 1997.

[11] Víctor M. Gómez. Comportamiento sigue pared en un robot con visión local. Proyecto fin de carrera, Universidad Rey Juan Carlos, 2002.

[12] K.
Konolige and K. Myers. The saphira architecture for autonomous mobile robots.

[13] Félix San Martín. Comportamiento sigue pelota en un robot con visión local. Proyecto fin de carrera, Universidad Rey Juan Carlos, 2002.

[14] Esther García Morata. Construcción de un teleoperador para el robot eyebot. Proyecto fin de carrera, Universidad Carlos III, 2002.

[15] R. Simmons and N.Y.Ko. The lane-curvature method for local obstacle avoidance. In *Conference on Intelligent Robots and Systems (IROS)*, 1998.

[16] Reid Simmons. The curvature-velocity method for local obstacle avoidance. In *International Conference on Robotics and Automation*, April 1996.

[17] Vicente Matellan y Jose Maria Cañas. Interacción persona robot hoy. In *III Congreso Interacción persona-ordenador, Universidad Carlos III de Madrid*, Mayo 2002.

[18] J. Borenstenin y Y. Koren. Real-time obstacle avoidace for fast mobile robots. *IEEE Transactions on Systems, Man, and Cybernetics*, 19(5):1179–1187, Sept./Oct. 1989.