MÁSTER UNIVERSITARIO
EN SISTEMAS TELEMÁTICOS E INFORMÁTICOS

Escuela Técnica Superior de Ingeniería de Telecomunicación

Curso Académico 2009/2010

Trabajo de Fin de Máster

Autolocalización visual en la RoboCup con algoritmos basados en muestras

Autor: Eduardo Perdices García
Tutor: José María Cañas Plaza

A mi familia y a todos mis amigos.

Gracias.

Agradecimientos

Quiero dar las gracias a todos los miembros del Grupo de Robótica de la Universidad Rey Juan Carlos por su apoyo y colaboración. En especial a Julio, Sara, Darío, Pablo y Gonzalo por haberme ayudado en numerosas ocasiones durante el desarrollo del proyecto.

También quisiera agradecer a José María todo el trabajo, dedicación y ayuda que me ha prestado durante todos estos meses de trabajo.

Como no podía ser de otra forma quisiera dar las gracias al equipo de fútbol robótico $S\pi Team$, en especial a los miembros de la URJC, Paco, Carlos y el propio José María por haberme permitido ser parte del equipo.

Además quisiera dar las gracias a mis mejores amigos por haber estado siempre a mi lado en los buenos y malos momentos, a Diego, Javier, Carolina, Marina y en especial a Estefanía, sin olvidar a otros muchos que también merecerían ser nombrados.

A todos, muchas gracias!

Resumen

La robótica autónoma engloba a los robots que son capaces de tomar decisiones sin la intervención humana. En este área de la robótica se desarrolla la RoboCup, donde un grupo de robots autónomos deben jugar al fútbol de forma cooperativa variando su comportamiento en función de su posición en el campo, por lo que es muy importante que el robot conozca su posición en todo momento. La autolocalización de robots en entornos conocidos es actualmente uno de los retos más importantes dentro del campo de la robótica. A partir de los distintos sensores con los que cuenta el robot, como sensores de ultrasonido, sensores láser o cámaras, el robot tiene que estimar su posición en el mundo que le rodea. En este proyecto hemos desarrollado nuevas técnicas para autolocalizar a un robot humanoide Nao dentro del campo de fútbol de la plataforma estándar de la RoboCup, utilizando sólo una cámara como sensor externo.

Se ha realizado un análisis 2D de cada una de las imágenes recibidas desde la cámara utilizando visión artificial con el fin de detectar las líneas y porterías del campo. Mediante estas imágenes y el mapa del mundo que rodea al robot hemos sido capaces de extraer información de posición que nos ha permitido conocer qué posiciones son plausibles para el robot dada una observación. Una vez realizado el análisis de cada observación de forma individual, se han desarrollado dos algoritmos de autolocalización para acumular la información de las imágenes, el primero de ellos utilizando el método de Monte Carlo y el segundo mediante un algoritmo evolutivo.

Los algoritmos se han desarrollado utilizando dos plataformas de desarrollo software diferentes, por un lado JdeRobot, que es la utilizada en el grupo de robótica de la URJC de forma generalizada, y por otro BICA, especialmente diseñada para hacer que el robot humanoide real juegue al fútbol en las distintas competiciones oficiales.

Índice general

Índice de figuras

Capítulo 1

Introducción

En este capítulo vamos a mostrar el contexto en el que se desarrolla nuestro proyecto, que está relacionado con la robótica, la visión artificial y la autolocalización. También haremos una descripción detallada de la competición a la que está enfocado nuestro trabajo, la plataforma estándar de la RoboCup.

1.1. Robótica

Un robot es un sistema electromecánico que utiliza una serie de elementos hardware, como por ejemplo, actuadores, sensores y procesadores, que rige su comportamiento por un software programable que le da la vida y la inteligencia. Se define la robótica como la ciencia y la tecnología de los robots, en la cual se combinan varias disciplinas como la mecánica, la informática, la electrónica y la ingeniería artificial, que hacen posible el diseño hardware y software del robot.

El primer robot programable se construyó en 1961, fue conocido como Unimate y servía para levantar piezas industriales a altas temperaturas, sin embargo no sería hasta la década de los 70 cuando se comenzase a desarrollar del todo esta tecnología, creándose en 1973 el primer robot con 6 ejes electromecánicos (conocido como Famulus) y en 1975 el primer brazo mecánico programable (PUMA).

A partir de los años 80 se comenzaron a utilizar en masa este tipo de robots, ya que proporcionaban una alta rapidez y precisión, y se utilizaron sobre todo para la fabricación de coches (figura 1.1(a)), empaquetamiento de comida y otros bienes, y la producción de placas de circuitos impresos.

Desde entonces se han realizado grandes avances en este campo, existiendo actualmente robots para todo tipo de propósitos, siendo ampliamente utilizados sobre todo en tareas industriales, donde realizan el trabajo de una forma mucho más precisa y barata que los humanos. También se utilizan robots en campos como el rescate de personas y la localización de minas, salvando muchas vidas humanas, además de utilizarse en diversos campos de la medicina, como la cirugía de alta precisión.

Además de los mencionados anteriormente, la robótica también se utiliza en un amplio espectro de aplicaciones, como la ayuda en las tareas del hogar, vigilancia, educación, o con fines militares.

Uno de los campos donde destaca la utilización de robots es en las misiones espaciales a otros planetas, que nos sirven para explorar la superficie de éstos sin la necesidad de la presencia de los humanos.

En los últimos años se han desarrollado diferentes robots especializados en la movilidad sobre un terreno, ya sea conocido o desconocido. Para ello, los robots hacen uso de distintos sensores que les permiten captar la información del exterior, y a partir de ellos son capaces de localizar su posición en un escenario conocido o de navegar a un destino solicitado.

Un ejemplo de este tipo de retos es la *Urban Challenge* [1], donde se realiza una competición con vehículos totalmente autónomos capaces de moverse entre el tráfico de una ciudad para alcanzar una serie de objetivos, y en la cuál es muy importante para el robot conocer dónde está.

Otro de los robots que destaca por su movilidad en superficies es *Roomba*, un robot de limpieza automático, que para utilizarlo sólo necesitamos colocarlo en el suelo y él automáticamente se encarga de recorrer toda la superficie del lugar donde nos encontremos.

También están proliferando en la actualidad muchos robots humanoides, que intentan simular la forma de andar de los humanos e interactúan con éstos utilizando multitud de sensores. Un ejemplo de este tipo de robots es el robot *Asimo*, desarrollado por Honda y cuya imagen puede verse en la figura 1.1(b).

[1] http://www.darpa.mil/grandchallenge/index.asp

(a)

(b)

Figura 1.1: Robots industriales fabricando coches (a). Robot *Asimo* (b).

1.2. Visión artificial en robótica

Para que el robot tenga información del entorno que le rodea, se pueden utilizar diversas fuentes de información, como sensores de ultrasonido, sensores láser, etc, que nos presentan información sobre distancias a objetos, o se pueden utilizar cámaras, con las que podemos extraer cierta información del mundo mediante la visión artificial.

La visión artificial es un campo de la inteligencia artificial que pretende obtener información del mundo a partir de una o varias imágenes, que normalmente vendrán dadas en forma de matriz numérica. La información relevante que se puede obtener a partir de las imágenes puede ser el reconocimiento de objetos, la recreación en 3D de la escena que se observa, el seguimiento de una persona u objeto, etc.

El inicio de la visión artificial se produjo en 1961 por parte de Larry Roberts, quien creó un programa que podía ver una estructura de bloques, analizar su contenido y reproducirla desde otra perspectiva, utilizando para ello una cámara y procesando la imagen desde un computador. Sin embargo, para obtener este resultado las condiciones de la prueba estaban muy limitadas, por ello otros muchos científicos trataron de solucionar el problema de conectar una cámara a un computador y hacer que éste describiese lo que veía. Finalmente, los científicos de la época se dieron cuenta de que esta tarea no era tan sencilla de realizar, por lo que se abrió un amplio campo investigación, que tomó el nombre de visión artificial, con el que se pretende conseguir lo anteriormente descrito para dar un

gran paso en la inteligencia artificial.

Así, se intenta que el computador sea capaz de reconocer en una imagen distintos objetos al igual que los humanos lo hacemos con nuestra visión. Sin embargo, se ha demostrado que este problema es muy complejo y que algo que para nosotros puede resultar automático, puede tardarse años en resolver para una máquina.

Por otra parte, a pesar del alto precio computacional que se paga al utilizar cámaras como fuente de información, si se consigue analizar correctamente la imagen, es posible extraer mucha información que no podría obtenerse con otro tipo de sensores.

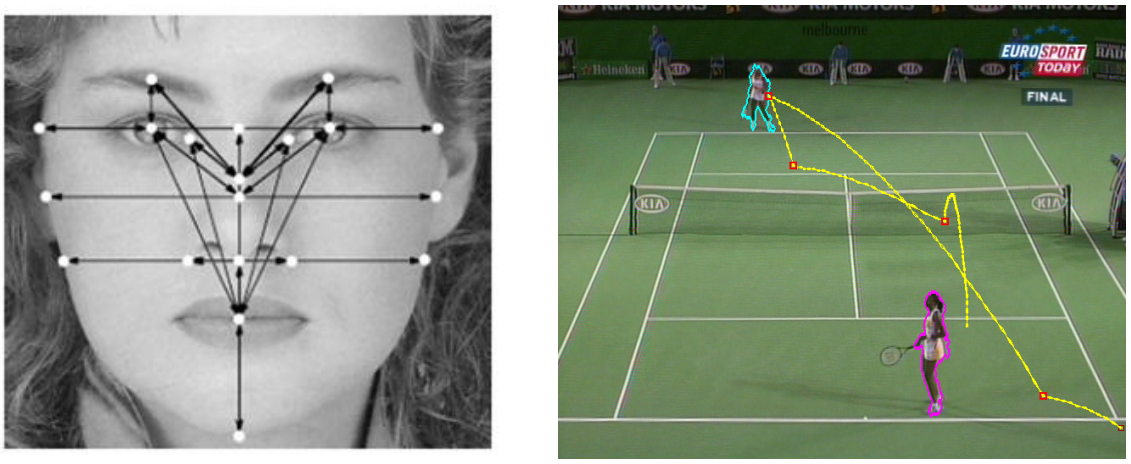
A comienzos de los años 90 comenzaron a aparecer ordenadores capaces de procesar lo suficientemente rápido imágenes, por lo que comenzaron a dividirse los posibles problemas de la visión artificial en otros más específicos que pudiesen resolverse. Actualmente se utiliza en muchos procesos científicos, militares o industriales, para el reconocimiento de objetos o en el seguimiento de éstos:

- Reconocimiento de objetos: A partir de una serie de características de un objeto, como puede ser su forma, su color o cualquier otro patrón, pueden compararse mediante algoritmos estas características con la imagen obtenida, para determinar si se encuentra algún objeto que siga este patrón (Figura 1.2(a)).

- Seguimiento de objetos: Una vez detectado un objeto, podemos realizar tareas de seguimiento de éste teniendo en cuenta que el objeto tiene 6 posibles grados de libertad, 3 de traslación y 3 de rotación. El seguimiento del objeto puede realizarse utilizando como fuente sus propiedades (bordes, esquinas, texturas, etc) o bien seleccionando una serie de puntos característicos del objeto y realizando únicamente el seguimiento de estos puntos. El seguimiento de objetos puede tener múltiples finalidades, desde la vigilancia en centros comerciales hasta la ayuda en algunos deportes (Figura 1.2(b)).

A pesar de que obtener información mediante visión artificial tiene una gran complejidad, las cámaras son el sensor más utilizado en los robots para percibir lo que les rodea. Esto es debido a que es un sensor muy barato comparado con el resto, y además la información que podemos obtener a partir de él es muy grande.

Algunos de los robots que vimos en la sección anterior utilizan las cámaras como sensor principal. Éste es el caso de los vehículos autónomos utilizados en la *Urban Challenge*, que necesitan la información que les proporcionan las cámaras para poder ver las señales de tráfico y evitar a otros vehículos.



(a) (b)

Figura 1.2: Reconocimiento de cara en imagen (a). Seguimiento de pelota durante un partido de tenis (b).

1.3. RoboCup

La RoboCup [2] es un proyecto a nivel internacional para promover la inteligencia artificial (IA), la robótica y otros campos relacionados. Se trata de promocionar la investigación sobre robots e IA proporcionando un problema estándar donde un amplio abanico de tecnologías pueden ser integradas y examinadas. La meta final de la RoboCup es: "Para el año 2050, desarrollar un equipo de fútbol de robots humanoides totalmente autónomos capaces de jugar y ganar contra el mejor equipo humano del mundo que exista en ese momento."

Para llegar a esta meta se deben incorporar diversas tecnologías, a las que debe hacer frente cada robot, tales como razonamiento en tiempo real, utilización de estrategias, trabajo colaborativo entre robots, uso de múltiples sensores o, el campo en el que está centrado nuestro proyecto, la autolocalización visual.

La primera persona en pensar en robots que jugasen al fútbol fue Alan Maxworth en 1992, de la universidad de British Columbia de Canadá. Al año siguiente, y de forma independiente a la idea de Alan, se creó la primera competición de fútbol robótico en japón, llamada Robot J-League, pero viendo el interés internacional que surgió por este proyecto decidieron renombrarla a Robot World Cup Initiative o simplemente RoboCup. Finalmente en 1997, se realizó la primera competición de la RoboCup en el formato

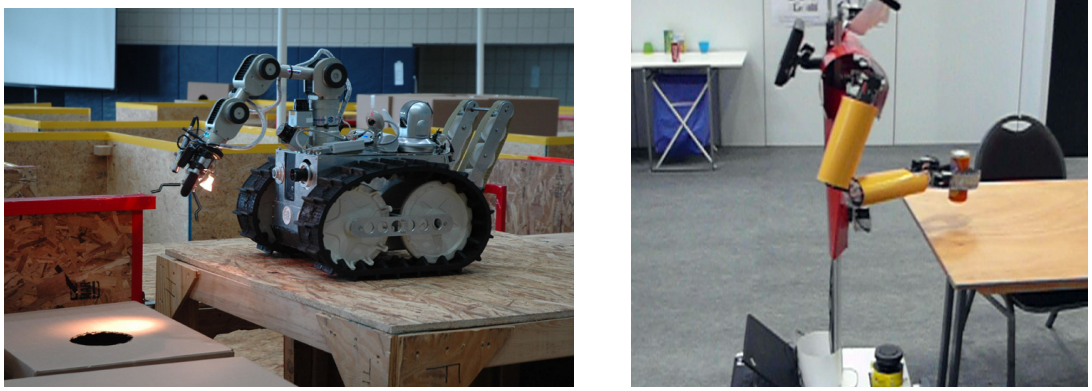
[2] http://www.robocup.org/

existente actualmente.

Las actividades realizadas por la RoboCup no son únicamente las competiciones entre robots, sino que también se componen de conferencias técnicas, programas educativos, infraestructuras de desarrollo, etc.

A pesar de que inicialmente el fin de la RoboCup era el anteriormente explicado, se han creado nuevos campos de investigación relacionados con los robots y que forman parte del mismo proyecto, haciendo que la iniciativa se divida en cuatro grandes competiciones:

- RoboCupSoccer: Proyecto principal del que ya hemos hablado, donde se realizan competiciones de fútbol entre robots autónomos.
- RoboCupRescue: Se pone a prueba a los robots en tareas de búsqueda y salvamento en terrenos desfavorables. En esta ocasión, los robots pueden ser tanto autónomos como guiados por control remoto (Figura 1.3(a)).
- RoboCupJunior: Trata de acercar las metas de la RoboCup a estudiantes de educación primaria y secundaria.
- RoboCup@Home: Centrada en la utilización de robots autónomos para realizar tareas del hogar y la vida diaria (Figura 1.3(b)).



(a) (b)

Figura 1.3: Robot de rescate para la RoboCupRescue (a). Robot que compite en una de las ligas de la RoboCup@Home (b).

Dentro de cada competición existen diversas categorías. A continuación vamos a explicar las distintas ligas existentes para la RoboCupSoccer, ya que es la utilizada en nuestro proyecto:

- Liga de simulación: No existen robots físicos, por lo que sólo se enfrentan robots en simulaciones virtuales.

- Liga de robots de tamaño pequeño: Categoría centrada en la cooperación multiagente, donde sólo pueden utilizarse robots de menos de 15 cm de altura, con un diámetro menor de 18 cm (Figura 1.4(a)) y que cuentan con una cámara cenital como sensor .

- Liga de robots de tamaño mediano: Se utilizan robots con una altura desde los 30 a los 80 cm y con un peso máximo de 40 kg. Los robots utilizados deben ser totalmente autónomos y pueden comunicarse entre ellos, siendo la visión su sensor principal.

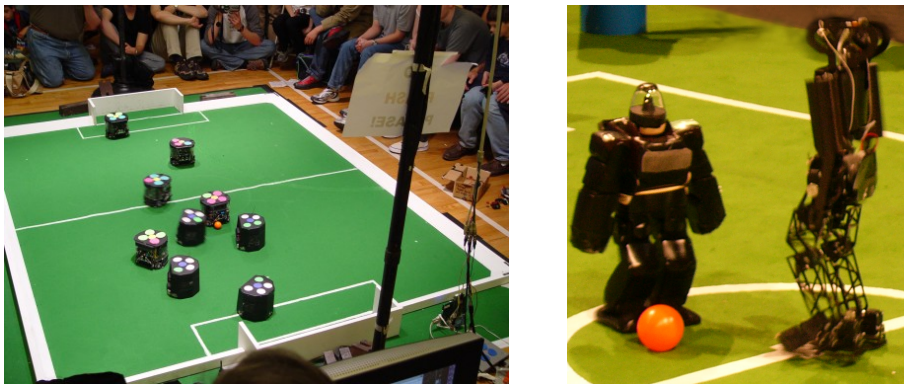
- Liga de humanoides: Centrada en el desarrollo de robots humanoides, de ahí que se haga más hincapié en el hardware utilizado que en el comportamiento software (Figura 1.4(b)).

- Plataforma estándar: Categoría en el que todos los participantes utilizan el mismo robot y sólo se centran en el desarrollo del software de éste. Actualmente se utiliza el robot Nao de Aldebaran Robotics (1.5), aunque hasta 2007 se utilizó el robot Aibo de Sony. Al igual que en la liga anterior, los robots utilizados deben ser totalmente autónomos y deben utilizar una cámara como sensor principal. Esta liga, junto con la liga de humanoides, es una de las que más ha progresado y la más cercana al reto fijado para 2050.

Además de la competición oficial de la RoboCup celebrada anualmente, también se han creado otros torneos a nivel mundial en los que poder competir con los robots. Así, ya dentro de la plataforma estándar, se han creado nuevas competiciones que siguen las mismas reglas que la RoboCup, como el German Open, el Japan Open, el US Open o el Mediterranean Open.

En estas competiciones, uno de los factores determinantes para el robot jugador es saber qué hacer cuando percibimos la pelota, puesto que la actuación del robot deberá ser distinta cuando se encuentre en su propia área (intentará despejar) que cuando esté en el área contraria (intentará tirar a portería). Por ello, uno de los problemas más importantes que se plantean es la localización dentro del campo, que se debe realizar utilizando visión artificial.

Conocer la posición del robot dentro del terreno de juego es muy importante, puesto que esta posición condiciona enormemente cual es el comportamiento adecuado a la situación en la que se encuentre el robot.



(a) (b)

Figura 1.4: Liga de robots de tamaño pequeño (a). Liga de humanoides (b).

(a)

Figura 1.5: Robots Nao durante la competición de la RoboCup 2009.

Desde el año 2001, el grupo de Robótica de la URJC trabaja en proyectos relacionados con la RoboCup, estudiando cómo generar los comportamientos de los robots dentro del campo.

Así, inicialmente se desarrollaron proyectos para la liga de simulación, como [Álvarez, 2001], quien desarrollo un comportamiento para un equipo utilizando lógica borrosa, y [Martínez, 2003], que programó un equipo de fútbol software capaz de jugar en la liga simulada de la RoboCup.

Más tarde se comenzaron a realizar proyectos con los robots Eyebot y los robots Aibo, como por ejemplo, [Crespo, 2003], que planteó el problema de la localización en el campo de la RoboCup con los robots Eyebot, o [Martín, 2008], proponiendo una solución a la localización para los robots Aibo mediante métodos markovianos.

Finalmente, desde 2009, se están realizando proyectos con los robots humanoides Nao. Ejemplo de ello es mi proyecto de fin de carrera [Perdices, 2009], donde se planteó un método visual de detección en 3D de las porterías, o la tesis doctoral [Agüero, 2010], quien desarrolló un método para la detección de la pelota de forma compartida y un mecanismo de intercambio de roles para los robots.

Tras analizar estos proyectos, hemos podido comprobar que aún no existía ningún método que permitiese localizar a los robots Nao en cualquier situación de forma eficiente y robusta, ya que en el proyecto realizado en [Perdices, 2009], los métodos propuestos, o sólo se podían utilizar al ver las porterías completas, algo que no suele ocurrir, o bien eran demasiado costosos, al utilizar cubos de probabilidad, que no podrían ser utilizados en tiempo real.

Así pues, partiendo de los proyectos anteriores, hemos decidido enfocar nuestro proyecto en la autolocalización de los robots Nao dentro del campo de la RoboCup, dada su gran importancia para el comportamiento de los robots en el campo.

1.4. Técnicas de Autolocalización

Una vez presentado el problema que abordamos en este proyecto, vamos a mostrar un breve estado del arte de las técnicas que existen para resolverlo y que han sido típicamente usadas en este problema.

La localización consiste en determinar con cierta certeza en qué posición se encuentra el robot en un momento concreto a partir de su información sensorial. Para ello, un robot puede utilizar distintos mecanismos: emplear la odometría, usar sensores especializados como los GPS, o utilizar otros sensores, como las cámaras.

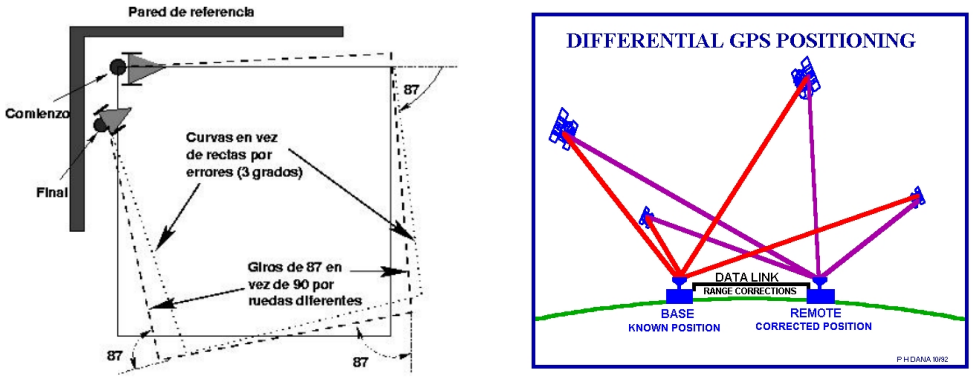
1. Odometría: Consiste en determinar la posición del robot conociendo el movimiento de los motores de que dispone y de su posición anterior. Por ejemplo, en el caso de un robot que se mueva con ruedas, puede determinarse la nueva posición conociendo cuántas vueltas ha dado cada rueda y las dimensiones de éstas. En el caso de robots humanoides o en robots con patas, los cálculos son mucho más complejos

puesto que hay que tener en cuenta un gran número de actuadores que se mueven simultáneamente.

Además, esta técnica es poco fiable, ya que puede haber errores de cálculo debido a deslizamientos, holguras de los actuadores, etc. Un ejemplo de lo anterior puede verse en la figura 1.6(a), donde un robot intenta moverse haciendo un cuadrado pero varía su trayectoria por errores de precisión.

2. GPS: El sensor GPS recibe señales de satélites que le permiten determinar la posición del robot en cualquier parte del mundo, con una precisión de pocos metros. El sistema GPS se compone de 27 satélites que orbitan alrededor de la tierra con trayectorias sincronizadas. Para determinar la posición se recibe la señal de al menos 3 satélites, que indican la situación y la hora de cada uno de ellos, con estos datos y sincronizando el retraso de las señales se puede calcular la localización por triangulación. Los cálculos pueden ser imprecisos debido a fallos en la obtención de datos, como errores de reloj, de órbita, rebotes de la señal, etc.

   Existe una técnica que permite conocer la posición actual con más precisión, y es la que más se utiliza en robótica, se conoce como GPS diferencial y consiste en corregir la desviación del robot en función de la desviación recibida en una posición conocida (Figura 1.6(b)).



(a) (b)

Figura 1.6: Error en la estimación de la posición usando odometría (a). Funcionamiento del GPS diferencial (b).

Este tipo de técnicas suelen utilizarse sobre todo en exteriores para localizar al robot con precisión, como por ejemplo en la competición *Grand Challenge*, donde varios vehículos autónomos compiten por llegar desde un punto de Estados Unidos hasta otro en el menor tiempo posible y sin intervención humana.

3. Balizamiento: Utilizando otros sensores, como las cámaras, puede determinarse la posición del robot en un mundo conocido analizando la información obtenida. Para ello se establecen una serie de balizas que pueden ser percibidas por el robot en posiciones previamente conocidas. Estas balizas pueden ser tanto objetos con unas determinadas características como marcas realizadas sobre paredes o suelos, aunque no tienen porqué ser necesariamente visuales, ya que también pueden utilizarse otros sensores como el sensor láser.

   Una vez que se han localizado una o varias balizas se puede determinar la posición del robot respecto a las balizas, ya sea por triangulación o por trilateración:

   - Triangulación: Cálculo de la posición utilizando el ángulo en el que encuentras las balizas.
   - Trilateración: Cálculo de la posición utilizando la distancia que existe hasta las balizas. Este tipo de cálculo es difícil de realizar utilizando cámaras, y es más común cuando se utilizan sensores de ultrasonido o sistemas GPS.

   Este tipo de técnicas han sido utilizadas en la RoboCup para facilitar la localización de los robots dentro del campo, para lo cual se utilizaban una serie de cilindros en distintos puntos conocidos con un código de colores característico (Figura 1.7). Estas balizas se han eliminado recientemente y ahora sólo pueden utilizarse para la localización elementos naturales del campo, como las líneas o las porterías.

(a)

Figura 1.7: Localización con balizas en la RoboCup.

4. Localización probabilística: Se determina la posición del robot por probabilidad, haciendo que cada posible posición en el mundo tenga una probabilidad entre 0 y 1 que indique si el robot puede encontrarse en esa posición, utilizando para ello la información recibida de los sensores.

   Esto se realiza utilizando un mapa de probabilidad en el que calculamos la probabilidad de cada observación instantánea, y donde se acumula la probabilidad de la última observación con las previamente calculadas.

   Esta técnica debe permitirnos tanto localizar al robot en una posición absoluta dentro del mundo en el que se encuentra, como recuperarnos de posibles errores cuando el robot es movido a otra posición (lo que se conoce como problema del secuestro).

   Dentro del marco probabilístico, algunos de los algoritmos más utilizados hasta el momento son los filtros de partículas o los filtros de Kalman. Este tipo de localización será la utilizada en nuestro proyecto y será detallada en profundidad en el capítulo 5.

También se han realizado diversos proyectos en el grupo de Robótica relacionados con la localización y ajenas a la RoboCup, como [López, 2005], que desarrolló un algoritmo de localización con visión para el robot Pioneer utilizando el método de Monte Carlo, o [Kachach, 2005], cuyo proyecto utilizaba para la localización en 2D un sensor láser en el robot Pioneer, y que constituyen antecedentes inmediatos de nuestro trabajo.

En el próximo capítulo veremos los objetivos concretos que debe alcanzar el proyecto, mostrando en el capítulo 3 el entorno y las plataformas de desarrollo utilizadas durante su realización. En el capítulo 4 nos centraremos en el análisis de imágenes individuales, para después explicar en el capítulo 5 su acumulación mediante los algoritmos de localización desarrollados. Finalmente, en el capítulo 6 veremos en qué grado se han alcanzado los objetivos propuestos y plantearemos algunos trabajos futuros con los que continuar el trabajo desarrollado.

Capítulo 2

Objetivos

Una vez presentado el contexto en el que se desarrolla nuestro trabajo, en este capítulo vamos a detallar los objetivos concretos que pretendemos alcanzar con nuestro proyecto, así como los requisitos que debe cumplir nuestra solución.

2.1. Descripción del problema

El principal objetivo del proyecto es la autolocalización de un robot dentro del terreno de juego de la RoboCup mediante la percepción espacial de las porterías y las líneas del campo, utilizando para ello la visión artificial con una sola cámara y validando experimentalmente nuestros resultados en simuladores y en robots reales. El robot utilizado tanto en las simulaciones como en la realidad será el robot Nao de Aldebaran Robotics.

El objetivo global lo hemos articulado en 4 subobjetivos que nos permitirán alcanzar el objetivo principal y que se describen a continuación:

1. A partir de los fotogramas recibidos desde la cámara, tendremos que detectar los puntos significativos de la imagen, como las porterías y las líneas del campo, teniendo en cuenta el reglamento de la RoboCup, lo que nos servirá para tener puntos de referencia para la localización.

2. Tras detectar los distintos elementos del campo, deberemos ser capaces de extraer información espacial en 3D a partir del análisis 2D realizado.

3. Una vez extraída la información 3D de cada observación, tendremos que calcular la posición del robot mediante acumulación de observaciones, siempre que las imágenes nos aporten la información necesaria para poder localizarnos.

4. Los algoritmos desarrollados deberán ser validados experimentalmente. Primero en simuladores, lo que nos permitirá mejorar los algoritmos sin necesidad de usar hardware real, y después en el robot real, asegurando su funcionamiento en situaciones reales, en las que previsiblemente las observaciones contendrán ruido.

   Además, estos algoritmos deberán estar integrados en el código del equipo de fútbol robótico de la URJC, con el objetivo de poder ser utilizado en competiciones oficiales.

Además de lo anterior, tendremos que tener en cuenta varios factores que harán que la complejidad del problema sea mayor. El primero de ellos es la utilización de la visión como único sensor externo, no pudiendo ayudarnos de otros sensores como el láser que nos facilitarían la tarea a la hora de, por ejemplo, medir distancias. Asimismo, el mundo del robot será en 3D, ya que la cámara del robot se encuentra a una altura considerable, al contrario de lo que sucedía hasta ahora, donde se simplificaba a un mundo bidimensional, puesto que se utilizaba el robot Aibo que cuenta con su cámara a muy baja altura.

Otro factor a tener en cuenta es que el escenario utilizado (el campo de la RoboCup) es un entorno difícil, puesto que los elementos que rodean al robot son dinámicos y están en continuo cambio y además el robot puede sufrir oclusiones, cambios de luminosidad, secuestros en su posición, etc.

2.2. Requisitos

Teniendo en cuenta los objetivos de la sección anterior, el proyecto deberá alcanzar los siguientes requisitos:

- Plataforma: Nuestro proyecto deberá estar integrado en la arquitectura BICA [Martín *et al.*, 2010], que es la utilizada por el equipo de fútbol robótico de la URJC. Esta arquitectura está desarrollada en el lenguaje de programación C++ y es multiplataforma, pudiendo utilizarse tanto en ordenadores convencionales como en el robot Nao. Así, se establece como requisito de partida el desarrollo del proyecto en este lenguaje de programación.

- Visión artificial: Para la localización del robot, deberemos hacer un amplio uso de la visión artificial, utilizando para ello únicamente una de las cámaras del robot.

- Eficiencia de los algoritmos: Los algoritmos utilizados deberán ser lo suficientemente eficientes como para ser ejecutados en el robot Nao sin perjudicar al resto de

funcionalidades del robot. El robot cuenta con un hardware muy limitado, como se describirá en el capítulo 3.1, por lo que los algoritmos deben estar altamente optimizados, considerando como válido un tiempo medio de ejecución menor a 50 ms.

- Precisión: La precisión conseguida en los distintos algoritmos que desarrollemos deberá ser del orden de centímetros, siendo aceptable un error medio en la localización en 3D de menos de 30 centímetros.

- Robustez: Los algoritmos perceptivos desarrollados deberán ser capaces de detectar posibles elementos externos que nos dificulten la detección, como la pelota, otros jugadores o los objetos que se encuentran fuera del campo, afectando lo menos posible a la localización.

- Licencia: El código del proyecto es software libre y será liberado bajo doble licencia, **GPLv3**[1] y **BSD**[2]. Se aplicará doble licencia para que se pueda emplear aquella que más interese al que utilice nuestro código.

2.3. Metodología de desarrollo

En el desarrollo de los componentes software de nuestro trabajo, el modelo de ciclo de vida utilizado ha sido el modelo en espiral basado en prototipos, ya que permite desarrollar el proyecto de forma incremental, aumentando la complejidad progresivamente y hace posible la generación de prototipos funcionales.

Este tipo de modelo de ciclo de vida nos permite obtener productos parciales que puedan ser evaluados, ya sea total o parcialmente, y facilita la adaptación a los cambios en los requisitos, algo que sucede muy habitualmente en los proyectos de investigación.

El modelo en espiral se realiza por ciclos, donde cada ciclo representa una fase del proyecto software. Dentro de cada ciclo del modelo en espiral se pueden diferenciar 4 partes principales que pueden verse en la figura 2.1, y donde cada una de las partes tiene un objetivo distinto:

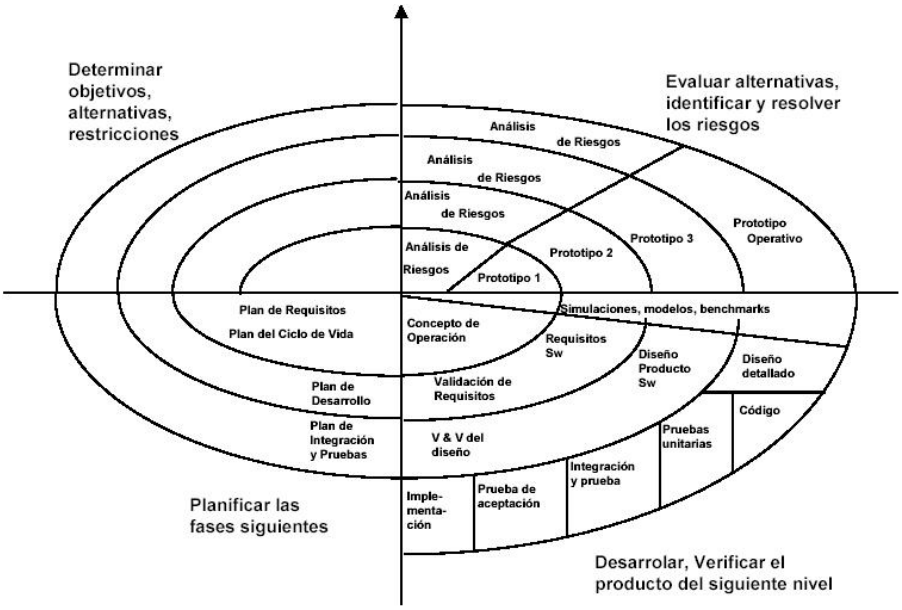
- Determinar objetivos: Se establecen las necesidades que debe cumplir el producto en cada iteración teniendo en cuenta los objetivos finales, por lo que según avancen las iteraciones aumentará el coste del ciclo y su complejidad.

[1] http://www.gnu.org/licenses/gpl-3.0.html
[2] http://www.opensource.org/licenses/bsd-license.html

- Evaluar alternativas: Determina las diferentes formas de alcanzar los objetivos que se han establecido en la fase anterior, utilizando distintos puntos de vista, como el rendimiento que pueda tener en espacio y tiempo, las formas de gestionar el sistema, etc. Además, se consideran explícitamente los riesgos, intentando reducirlos lo máximo posible.

- Desarrollar y verificar: Desarrollamos el producto siguiendo la mejor alternativa para poder alcanzar los objetivos del ciclo. Una vez diseñado e implementado el producto, se realizan las pruebas necesarias para comprobar su funcionamiento.

- Planificar: Teniendo en cuenta el funcionamiento conseguido por medio de las pruebas realizadas, se planifica la siguiente iteración revisando posibles errores cometidos a lo largo del ciclo y se comienza un nuevo ciclo de la espiral.



(a)

Figura 2.1: Modelo en espiral.

Los ciclos que se han seguido en nuestro proyecto están relacionados con cada una de las etapas que se describirán en la siguiente sección. A lo largo de estas etapas, se han realizado reuniones semanales con el tutor del proyecto para negociar los objetivos que se pretendían alcanzar y para evaluar las alternativas de desarrollo.

En mi ficha web personal del grupo de robótica de la URJC [3] pueden encontrarse imágenes y vídeos que muestran el avance realizado a lo largo del tiempo.

[3] http://jderobot.org/index.php/User:Eperdes

2.3.1. Plan de trabajo

Durante el transcurso del proyecto se han marcado progresivamente una serie de pasos a cumplir dividiendo el trabajo a realizar en distintas etapas:

1. Primera versión en JdeRobot: JdeRobot es la plataforma utilizada para la mayoría de proyectos dentro del grupo de robótica de la URJC, y que ya era conocida por nosotros, puesto que fue la utilizada en nuestro proyecto de fin de carrera [Perdices, 2009]. Por ello, decidimos reutilizar el código desarrollado en ese trabajo y crear un nuevo componente que partiese de él.

2. Primer modelo de observación: Realizamos el primer modelo de observación en el simulador para detectar en las imágenes tanto las líneas como las porterías del campo. De esta forma, a partir de los elementos observados y una posición dada, generamos una funcionalidad capaz de evaluar la probabilidad de que el robot se encontrase en esa posición. Además, desarrollamos herramientas de depuración con las que evaluar la calidad de nuestro modelo de observación.

3. Primer modelo de movimiento: Modelo de movimiento inicial, en el que sólo teníamos en cuenta el movimiento del cuello del robot, puesto que el desplazamiento del robot aún no era posible calcularlo, y que serviría para actualizar la posición del robot en cada iteración.

4. Algoritmo de Monte Carlo: Una vez que dispusimos de los modelos de observación y movimiento iniciales, pudimos realizar una primera implementación completa del algoritmo de Monte Carlo, con la que poder comprobar el funcionamiento de la acumulación de observaciones. El algoritmo de Monte Carlo fue desarrollado partiendo de proyectos anteriores que lo utilizasen como [Crespo, 2003] o [Domínguez, 2009].

5. Algoritmo evolutivo: Tras comprobar el funcionamiento del algoritmo de Monte Carlo, vimos la necesidad de mejorar el comportamiento de la localización frente a simetrías, por lo que decidimos implementar un algoritmo evolutivo que partía de los trabajos [García, 2007] y [Marugán, 2010], aunque adaptado al contexto de la RoboCup.

6. Adaptación del código a BICA: Una vez desarrollados los dos algoritmos, decidimos portar nuestra implementación a la arquitectura BICA, en la que la forma de interactuar con los sensores (como las cámaras) cambiaba con respecto a JdeRobot, por lo que tuvimos que aprender a utilizar esta nueva arquitectura. Además, al utilizar

BICA, pudimos probar por primera vez nuestros algoritmos en el robot real y darnos cuenta de posibles mejoras.

7. Mejoras en la observación y el movimiento: Al utilizar el robot real, tuvimos que mejorar la eficiencia de los algoritmos, para lo que decidimos optimizar el modelo de observación añadiendo preprocesado de información y variando la forma de detectar los elementos en la imagen. Además, también pudimos añadir al modelo de movimiento el desplazamiento del robot, gracias a las nuevas interfaces que nos proporcionaba BICA.

8. Detección de falsos positivos: Para lograr que nuestro modelo de observación pudiese utilizarse, tuvimos que hacerlo robusto frente a falsos positivos. Para ello añadimos a los elementos observados en la imagen una validación en la que entraban en juego diversas comprobaciones, como por ejemplo el cálculo del horizonte, el fin del campo, el análisis de *Blobs* de portería en la imagen o la detección de otros robots. Esto hizo que mejorase nuestro modelo de observación y que pudiese utilizarse en entornos reales y dinámicos.

9. Mejoras finales: Con el fin de mejorar la precisión del algoritmo, utilizamos otros sensores que nos proporcionaba el robot, como los *FSR* para mejorar el funcionamiento de los algoritmos ante los secuestros, y modificamos el algoritmo evolutivo añadiéndole más información de contexto para aumentar su eficiencia y precisión.

Capítulo 3

Entorno y plataforma de desarrollo

En este capítulo vamos a describir los elementos empleados en el desarrollo del proyecto, tanto hardware como software. Además describiremos algunas de las tecnologías utilizadas que nos han ayudado alcanzar los objetivos del proyecto.

El sistema operativo utilizado para el desarrollo software ha sido Ubuntu 9.10, una de las distribuciones de GNU/Linux más extendidas actualmente, y que está basada en Debian. Este sistema operativo es de libre distribución y soporta oficialmente las arquitecturas hardware Intel x86 y AMD64, que se corresponden con las arquitecturas de los ordenadores que hemos utilizados.

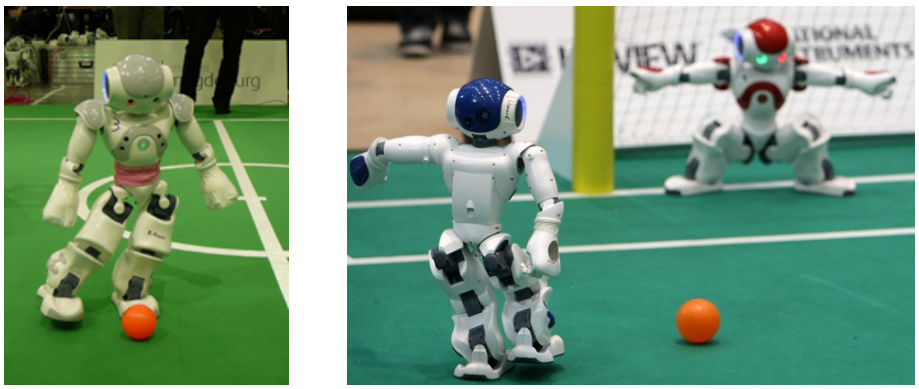
3.1. Robot Nao

El robot Nao es el robot utilizado en la liga de la plataforma estándar de la RoboCup desde el año 2008, donde sustituyó al robot Aibo de Sony. Se trata de un robot humanoide desarrollado por Aldebaran Robotics que actualmente se encuentra en su versión Nao RoboCup Edition V3+, lanzada a comienzos de 2010. Podemos ver algunas imágenes de este robot en la figura 3.1.

Este robot es utilizado tanto en la realidad como en simulación, por lo que es necesario conocer en profundidad sus características, ya que condicionan la solución final. Algunas de sus características principales son:

- 21 grados de libertad, permitiendo una gran amplitud de movimientos (figura 3.2(a)).
- Detectores de presión en pies y manos, conocidos como FSR.

- 2 cámaras con distintas zonas de visión (figura 3.2(b)).
- 4 sensores de ultrasonido.
- Inerciales, de gran utilidad en caso de caídas.
- Ordenador incorporado con comunicación vía Ethernet o Wi-Fi, permitiendo así las comunicaciones inalámbricas.
- LEDs en diversas partes del cuerpo, de gran ayuda para conocer el estado del robot.



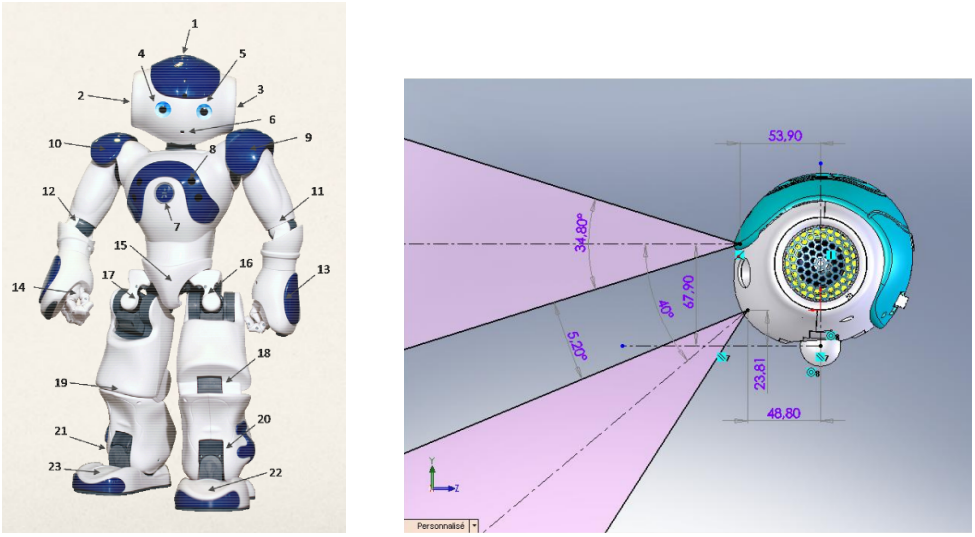
(a) (b)

Figura 3.1: Robot Nao de Aldebaran Robotics (a). Imagen de la RoboCup 2009 (b).

Además de las características anteriores, el robot también dispone de un sistema multimedia (compuesto por 4 micrófonos y 2 altavoces hi-fi) que le permite comunicarse mediante voz, un localizador de sonido y algoritmos de reconocimiento facial, aunque estas características no son utilizadas por el momento en el desarrollo de software para la RoboCup.

Como procesador utiliza una CPU modelo x86 AMD GEODE 500MHz y usa como sistema operativo Linux. Para el movimiento autónomo usa una batería que le otorga una autonomía de 45 minutos en espera o de 15 minutos caminando.

El robot Nao dispone de una interfaz de programación que permite interactuar con él de una forma intuitiva, utilizando la arquitectura software Naoqi que veremos en la próxima sección.



(a) (b)

Figura 3.2: Grados de libertad del robot Nao (a). Cámaras del robot Nao (b).

3.1.1. Naoqi

Naoqi es una arquitectura software multiplataforma para la comunicación y el desarrollo de software del robot Nao, y que permite crear un entorno distribuido donde varios binarios pueden comunicarse entre sí. Soporta el desarrollo en los lenguajes de programación C++, Python y Urbi, siendo además compatible con la plataforma de simulación Webots.

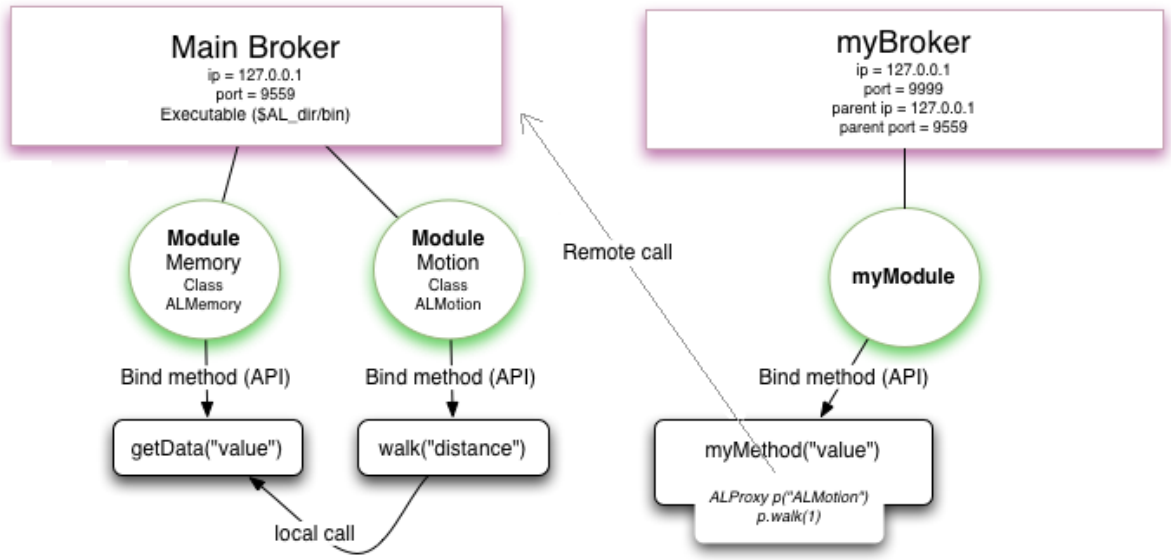
Esta arquitectura software es utilizada tanto para la comunicación con el robot Nao real como con el simulado.

Naoqi se compone de tres componentes básicos, los *brokers*, los módulos y los *proxies*:

- Brokers: Son ejecutables que se conectan a un puerto e IP determinados. Todos los *brokers* están conectados formando un árbol, cuya raíz es un *broker* principal (MainBroker), como puede verse en la figura 3.3.
- Módulos: Son clases que derivan de ALModule y que presentan una funcionalidad concreta, como por ejemplo ALMotion para el acceso a los motores y ALVideoDevice para el acceso a la cámara del robot. Estos módulos pueden ser cargados por un *broker*.
- Proxies: Sirven para crear un representante de un módulo a través del cual utilizamos las operaciones que nos proporciona ese módulo.

Algunos de los módulos principales que hemos utilizado son:

- ALMotion: Módulo encargado de las funciones de locomoción. Se utiliza principalmente para establecer los ángulos de apertura de las distintas articulaciones. En general, los parámetros que acepta para cada articulación son el grado final en el que quedará la articulación, la velocidad de la transición, o bien, la duración que tendrá la transición. También es el módulo encargado de controlar el centro de masas, además de permitirnos llamar a funciones que directamente hacen al robot andar o girar.
- ALVideoDevice: Es el módulo encargado de comunicarse con las cámaras del robot Nao. Mediante este módulo obtenemos las imágenes de la cámara en el formato y dimensiones que pongamos como parámetros, permitiendo así conseguir la imagen en el formato necesario para procesarla después.



(a)

Figura 3.3: Comunicación de *brokers* y módulos en Naoqi.

3.2. JdeRobot

La fase inicial del proyecto se ha desarrollado en la plataforma JdeRobot [1]. Esta plataforma ha sido creada por el grupo de robótica de la URJC, se trata de una plataforma de desarrollo de software para aplicaciones con robots y visión artificial. Se ha elegido JdeRobot por la facilidad con la que se pueden utilizar otras aplicaciones ya creadas para esta plataforma de desarrollo, como por ejemplo nuestro proyecto de fin de carrera [Perdices, 2009], pudiendo reutilizarse partes del código de otros proyectos.

JdeRobot está desarrollado en C, C++ y Python y proporciona un entorno de programación donde la aplicación se compone de distintos hilos de ejecución asíncronos llamados esquemas. Cada esquema es un *plugin* que se carga automáticamente en la aplicación y que realiza una funcionalidad específica que podrá ser reutilizada.

Existen dos tipos de esquemas en JdeRobot: los perceptivos, que son los encargados de realizar algún tipo de procesamiento de datos para proporcionar información sobre el robot y el mundo en el que opera, y los esquemas de actuación, que se encargan de tomar decisiones para alcanzar una meta, como lanzar órdenes a los motores o lanzar nuevos esquemas, ya que pueden combinarse formando jerarquías.

JdeRobot simplifica el acceso a los dispositivos hardware desde el programa, de modo que leer la medida de un sensor es simplemente leer una variable local y lanzar órdenes a los motores es tan simple como escribir en otra. Para generar este comportamiento se han desarrollado una serie de drivers, que actúan como *plugins* y son los encargados de dialogar con los dispositivos hardware concretos.

En particular, para desarrollar nuestros algoritmos hemos utilizado algunos de los drivers y esquemas que proporciona la plataforma, que son:

- Video4linux: Driver encargado de la obtención de imágenes desde las cámaras que actualiza de forma automática las imágenes recibidas. Este driver ha sido empleado en diversas pruebas realizadas con cámaras reales.
- Naobody: Driver que nos permite comunicarnos con el robot Nao para obtener tanto las imágenes como el valor de sus sensores de movimiento. Utilizado a la hora de comunicarnos con el simulador Webots.
- Gazebo: Driver que nos permite mover y obtener imágenes de robots en el simulador Gazebo, el cual ha sido utilizado durante el desarrollo de nuestro trabajo.

[1] http://jderobot.org/

- Naooperator: Esquema que nos permite teleoperar al robot para manejar tanto su cuello como sus desplazamientos, lo que nos ha facilitado la evaluación de los modelos de movimiento utilizados.

La versión utilizada de JdeRobot ha sido la 4.3.0, lanzada en abril de 2009, y que se compone de 17 esquemas y 12 drivers.

3.3. GTK+ y Glade

GTK+ es un conjunto de bibliotecas multiplataforma empleadas para crear interfaces gráficas de usuario en múltiples lenguajes de programación como C, C++, Java, Python, etc. GTK+ es software libre bajo licencia LGPL y es parte del proyecto GNU. Entre las bibliotecas que componen a GTK+, destaca GTK, que es la que realmente contiene los objetos y funciones para la creación de la interfaz de usuario.

Son numerosas las aplicaciones desarrolladas con esta librería, algunas muy conocidas como el navegador web Firefox o el editor gráfico GIMP, por lo que puede comprobarse su gran potencia y estabilidad.

Para facilitar la creación de las interfaces existe el diseñador de interfaces Glade, una herramienta de desarrollo visual de interfaces gráficas de GTK. Esta herramienta es independiente del lenguaje de programación, ya que genera un archivo en XML que debe ser leído e interpretado utilizando la librería Glade del lenguaje que se vaya a utilizar.

En nuestro proyecto, hemos utilizado GTK+ para la realización de las interfaces gráficas para la arquitectura JdeRobot, ayudándonos del diseñador de interfaces Glade para simplificar el desarrollo.

3.4. BICA

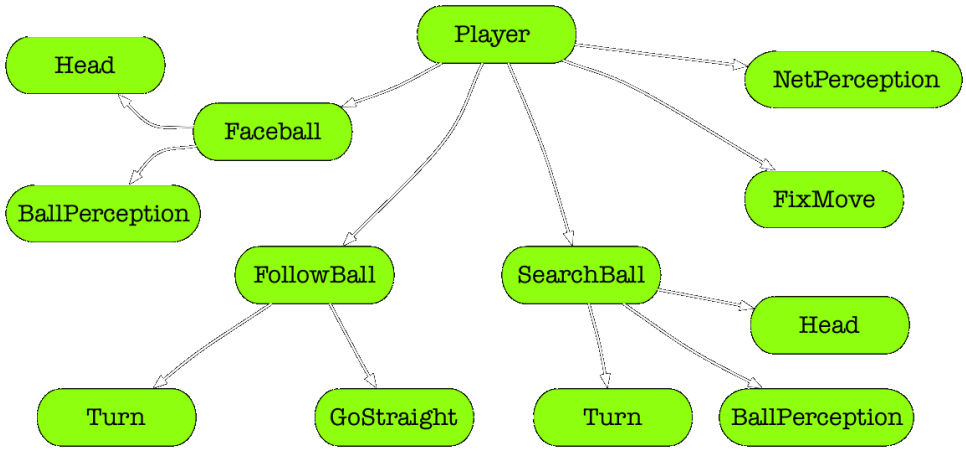
La segunda arquitectura de desarrollo utilizada en el proyecto ha sido la arquitectura BICA (Behavior-based Iterative Component Architecture). BICA es una arquitectura en C++ basada en componentes que se ejecutan de forma iterativa en un único thread y que se comunican entre sí mediante llamadas a métodos.

Cada componente contiene un método *step* que se ejecuta de forma iterativa a una frecuencia determinada, esta frecuencia vendrá dada por una frecuencia máxima que se

establece en cada componente y dependerá del tiempo de cómputo que consuman los otros componentes.

En el caso de que un componente dependa de otros, se distinguirá entre componentes perceptivos (de los que depende el componente) y de actuación (los que modula el componente). Independientemente de la frecuencia que tenga el componente, antes de la ejecución de su propio *step*, se llamará a los *steps* de sus componentes perceptivos, y después de la ejecución, se llamará a los *steps* de los componentes de actuación.

Esto hace que a partir del *step* de un componente raíz, se propaguen las llamadas a todos los componentes utilizados durante la ejecución del programa, como se muestra en la imagen 3.4



(a)

Figura 3.4: Propagación de llamadas en BICA.

Para la comunicación con Naoqi, BICA implementa un módulo llamado *Coach* que es cargado por Naoqi al inicio de su ejecución. Una vez que se está ejecutando el *Coach* en Naoqi, los componentes desarrollados pueden hacer llamadas a otros módulos de Naoqi para utilizar los sensores y actuadores del robot.

En nuestro proyecto, hemos implementado un componente de BICA por cada algoritmo de localización utilizado, así como otros componentes que nos han facilitado la implementación de la localización. Estos componentes a su vez han utilizado diversos componentes perceptivos para obtener la información del entorno del robot, entre los que destacan:

- Perception: Componente encargado de obtener las imágenes del robot y de aplicarle diversos filtros para facilitar el análisis de la imagen 2D.

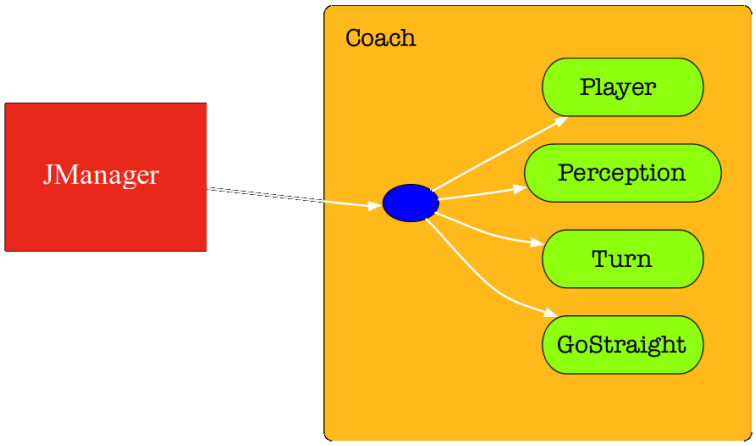
- Body: Componente que actúa sobre los motores del robot para controlar su desplazamiento, utilizado para conocer el movimiento del robot en cada instante.
- NetDetector: Componente que detecta regiones de la imagen en las que se encuentra la portería, lo que ha sido de gran ayuda a la hora de validar nuestra detección de las porterías del campo.

La versión de BICA utilizada en el proyecto ha sido la 2.4.1, publicada en marzo de 2010.

3.5. JManager

La comunicación con la arquitectura BICA se ha realizado mediante la herramienta JManager. Esta herramienta está desarrollada en Java y se utiliza principalmente para depurar o interactuar con cada uno de los componentes que integran la arquitectura.

Esta comunicación se realiza a través de una conexión TCP o UDP con el módulo de Naoqi que implementa BICA (llamado *Coach*), que es el encargado de llamar al método del componente correspondiente analizando el mensaje recibido (figura 3.5).



(a)

Figura 3.5: Conexión entre JManager y BICA.

Esta herramienta ha sido utilizada para interactuar con los componentes que hemos desarrollados para BICA, permitiendo depurar y mostrar los resultados de los algoritmos de localización de forma gráfica, al igual que lo hacíamos en JdeRobot.

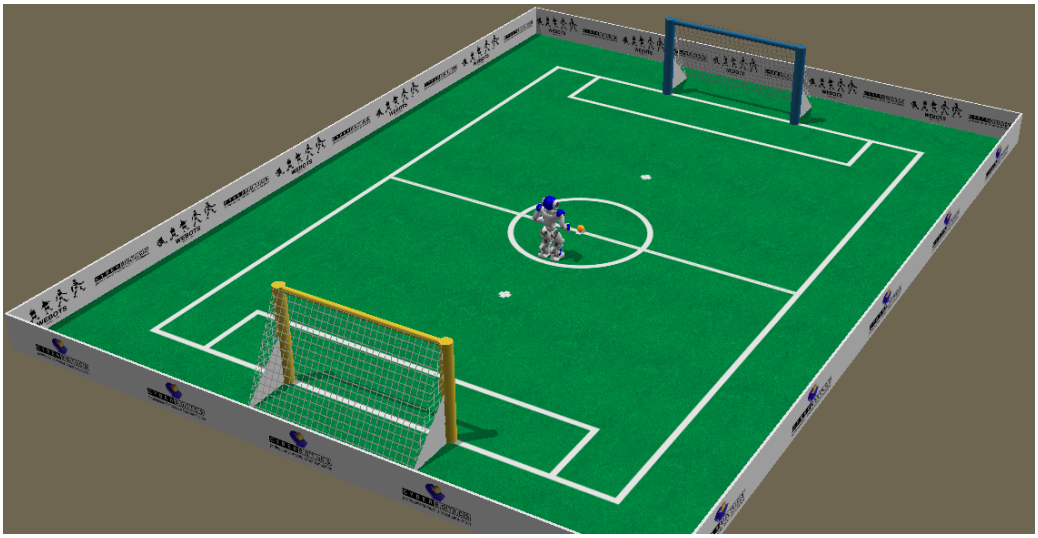
3.6. Webots y Gazebo

Para probar los desarrollos realizados para el robot Nao sin necesidad de utilizar el robot real, hemos utilizado dos simuladores: Webots[2] y Gazebo[3]. Se han utilizado dos simuladores mejor que uno solo para probar la robustez de los algoritmos, que funcionan sin cambios en ambas plataformas, y comprobar que estaban suficientemente depurados.

El primero de los simuladores, Webots, es un entorno de desarrollo creado por la compañía Cyberbotics utilizado para modelar, programar y simular robots móviles.

Con Webots se pueden configurar una serie de robots y objetos que interactúan entre sí en un entorno compartido. Además, para cada objeto se pueden establecer sus propiedades, tales como forma, color, textura, masa, etc. También hay disponibles un amplio conjunto de actuadores y sensores para cada robot, de esta manera podemos probar el comportamiento de la física del robot dentro de un mundo realista.

Dentro de la plataforma de desarrollo existen varios modelos de robots y objetos preestablecidos, entre los que se encuentra el robot Nao y los objetos necesarios para simular el campo de la RoboCup (figura 3.6).



(a)

Figura 3.6: Campo de la RoboCup en el simulador Webots.

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

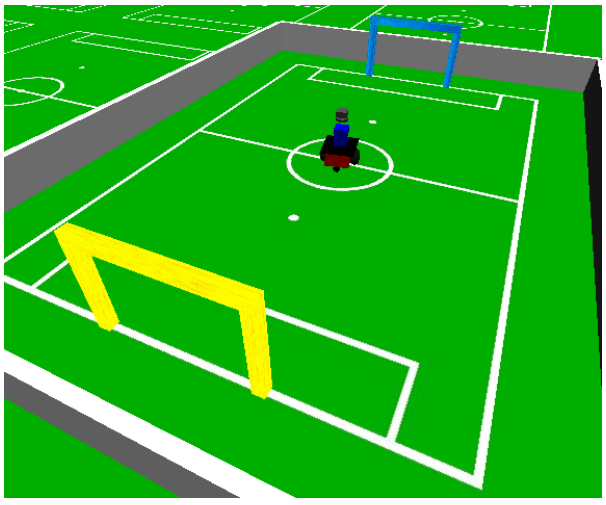
[3]http://playerstage.sourceforge.net/gazebo/gazebo.html

La comunicación con el robot Nao en Webots se ha realizado de dos formas diferentes dependiendo de la plataforma utilizada. En el caso de JdeRobot hemos utilizado el driver Naobody, con el que podemos acceder a la cámara y los motores del robot, mientras que en el caso de BICA podemos realizar la conexión sin utilizar ningún elemento extra.

Por su parte, Gazebo es un simulador de múltiples robots en 3D, permitiendo simular también sensores y objetos con los que obtener realimentación e interactuar. Gazebo es software libre bajo licencia GPL y forma parte del proyecto Player/Stage. Proporciona varios modelos de robots, como el Pioneer2DX, Pioneer2AT y SegwayRMP, y diversos mundos en los que probar estos robots.

Además, al igual que Webots, permite configurar nuestros propios mundos, robots y objetos, en los que se pueden establecer las propiedades que creamos necesarias. Sin embargo, al contrario que en Webots, aún no existen el robot Nao ni el campo de fútbol de la RoboCup previamente diseñados, aunque se está trabajando en ello en otros proyectos[4].

Por ello, para realizar nuestras pruebas, hemos utilizado un mundo para Gazebo creado por Francisco Rivas[5], en el que simula el campo de fútbol de la RoboCup según las reglas establecidas para la plataforma estándar en el año 2009 (figura 3.7), y donde se utiliza como robot un Pioneer2DX, que si bien no es igual que el robot Nao, es válido para la simulación puesto que dispone de una cámara con la que ver el mundo y puede moverse a través de él.



(a)

Figura 3.7: Campo de la RoboCup en el simulador Gazebo.

[4]http://jderobot.org/index.php/Jbermejo-pfc-itis
[5]http://jderobot.org/index.php/Frivas-pfc-itis

Para comunicarnos con el robot en el simulador Gazebo hemos tenido que utilizar obligatoriamente la arquitectura JdeRobot, puesto que en este caso no es posible realizar la comunicación desde BICA. Para ello, hemos utilizado el driver Gazebo que ya está disponible en JdeRobot, lo que nos ha permitido teleoperar el robot y obtener sus imágenes.

3.7. OpenGL

OpenGL es una especificación estándar que define una API multiplataforma e independiente del lenguaje de programación para desarrollar aplicaciones con gráficos en 2D y 3D. A partir de primitivas geométricas simples, como puntos o rectas, permite generar escenas tridimensionales complejas. Actualmente es ampliamente utilizado en realidad virtual, desarrollo de videojuegos (figura 3.8) y en multitud de representaciones científicas.

Hemos utilizado esta API en nuestras aplicaciones para la representación en 3D del campo de la RoboCup, con el objetivo de mostrar la localización calculada por nuestros algoritmos y facilitar la depuración, como se mostrará en la sección 5.3.

(a)

Figura 3.8: Captura del videojuego Counter Strike, desarrollado en OpenGL.

3.8. Componente BICA Kinematics

Para el desarrollo de los algoritmos que vamos a describir en los siguientes capítulos hemos necesitado desarrollar varios componentes de BICA que han servido como infraestructura. Estos componentes se han integrado en la versión oficial de BICA para poder ser reutilizados en otras aplicaciones distintas a las de nuestro proyecto, y forman parte de los aportes realizados a lo largo de éste.

El primer componente desarrollado surgió de la necesidad de realizar numerosos cálculos geométricos que nos permitiesen obtener puntos en 3D a partir de los píxeles en 2D de las imágenes observadas y viceversa, para poder extraer información espacial relativa desde las imágenes.

Estos cálculos pueden realizarse de forma sencilla utilizando la librería *Progeo*, desarrollada por el grupo de Robótica de la URJC, y que está disponible tanto en la plataforma JdeRobot como en BICA. Esta librería permite transformaciones entre puntos en 3D y 2D siempre que le proporcionemos información sobre los parámetros intrínsecos y extrínsecos de la cámara.

Los parámetros intrínsecos de la cámara (foco de atención y centro óptico), se obtienen como veremos en la sección 3.9, mientras que para los extrínsecos (posición y orientación en 3D de la cámara) se deben obtener de la geometría interna del robot.

Para abstraer estos cálculos, se ha generado un componente para BICA llamado *Kinematics*, que se encarga de calcular en todo momento cuál es la posición relativa de la cámara respecto del centro del robot. Antes de ver cómo se calcula esta posición, vamos a describir los grados de libertad con los que cuenta la cámara del robot.

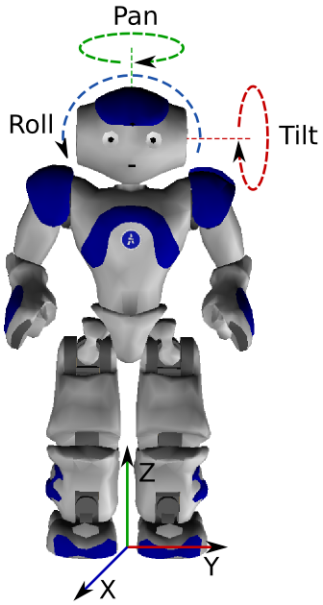
Los grados de libertad de la cámara que debemos tener en cuenta son la posición (*X*, *Y*, *Z*) en coordenadas relativas al robot, y los ángulos (*Pan*, *Tilt*, *Roll*). Estos grados de libertad pueden verse más claramente en la imagen 3.9.

Como ya hemos comentado, hemos necesitado calcular en todo momento la posición de la cámara respecto del centro del robot (RT_{cam_robot}). Este cálculo podría realizarse calculando con matrices de rotación y traslación (RT) la posición de cada una de las articulaciones del robot hasta llegar a la cámara. Sin embargo, Naoqi ya nos proporciona diversas funciones de utilidad que nos permiten conocer la odometría del robot dentro del módulo *ALMotion*:

- *vector< float> getPosition (string pChainName, int pSpace)*: Permite conocer la posición en 6 dimensiones (X,Y,Z,Pan,$Tilt$,$Roll$) de los brazos, las piernas, y el cuello

respecto del cuerpo del robot o respecto del centro del robot.

- *vector< float> getForwardTransform (string pChainName, int pSpace)*: Obtenemos la matriz RT de brazos, piernas y cuello respecto del cuerpo del robot o respecto del centro del robot.



(a)

Figura 3.9: Grados de libertad de la cámara del robot Nao.

Con estas funciones podemos calcular la posición del cuello respecto del centro del robot (RT_{cuello_robot}) mediante la función *getForwardTransform*. Además, también podemos calcular la RT de la cámara que estemos utilizando en el robot respecto del cuello (RT_{cam_cuello}), puesto que conocemos las distancias y ángulos que hay entre el cuello del robot y las cámaras.

De esta forma, multiplicando las matrices RT_{cuello_robot} y RT_{cam_cuello} podemos obtener la posición de la cámara respecto del centro del robot, como se ve en la ecuación 3.1, donde α es la inclinación de la cámara respecto al cuello ($Tilt$) y (X_{cc}, Y_{cc}, Z_{cc}) la diferencia entre la posición del cuello y la cámara.

$$RT_{cam\_robot} = RT_{cuello\_robot} * \begin{pmatrix} cos\alpha & 0 & -sen\alpha & X_{cc} \\ 0 & 1 & 0 & Y_{cc} \\ sen\alpha & 0 & cos\alpha & Z_{cc} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.1}$$

También es necesario obtener los valores de los ángulos (*Pan*, *Tilt*, *Roll*), que se obtienen directamente de la función *getPosition* del cuello respecto del centro del robot, puesto que estos ángulos son iguales tanto en el cuello como en la cámara.

Para poder utilizar la odometría con la librería *Progeo*, también necesitamos calcular el foco de atención al que está mirando la cámara del robot. Esto puede obtenerse fácilmente una vez que ya conocemos la RT de la cámara, como vemos en la ecuación 3.2:

$$Foa_{camara} = RT_{cam\_robot} * \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} \tag{3.2}$$

Con esto ya tendremos los parámetros extrínsecos de la cámara respecto del centro del robot, por lo que podremos realizar transformaciones entre 2D y 3D en posiciones relativas.

El componente que hemos desarrollado también permite calcular una nueva matriz RT que represente la cámara en coordenadas del mundo (RT_{cam_mundo}), pudiendo realizar así transformaciones entre 2D y 3D en posiciones absolutas.

Para ello, los algoritmos de localización desarrollados nos proporcionarán la posición (X_r, Y_r) del robot en el mundo y su rotación θ_r (RT_{robot_mundo}), puesto que el resto de grados de libertad $(Pan, Tilt, Roll, Z)$ son comunes a ambos sistemas de referencia.

Estos valores deberán utilizarse para calcular los valores extrínsecos de la cámara respecto de las coordenadas del campo, mediante la ecuación 3.3.

$$RT_{cam\_mundo} = \begin{pmatrix} cos\theta_r & -sen\theta_r & 0 & X_r \\ sen\theta_r & cos\theta_r & 0 & Y_r \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} * RT_{cam\_robot} \tag{3.3}$$

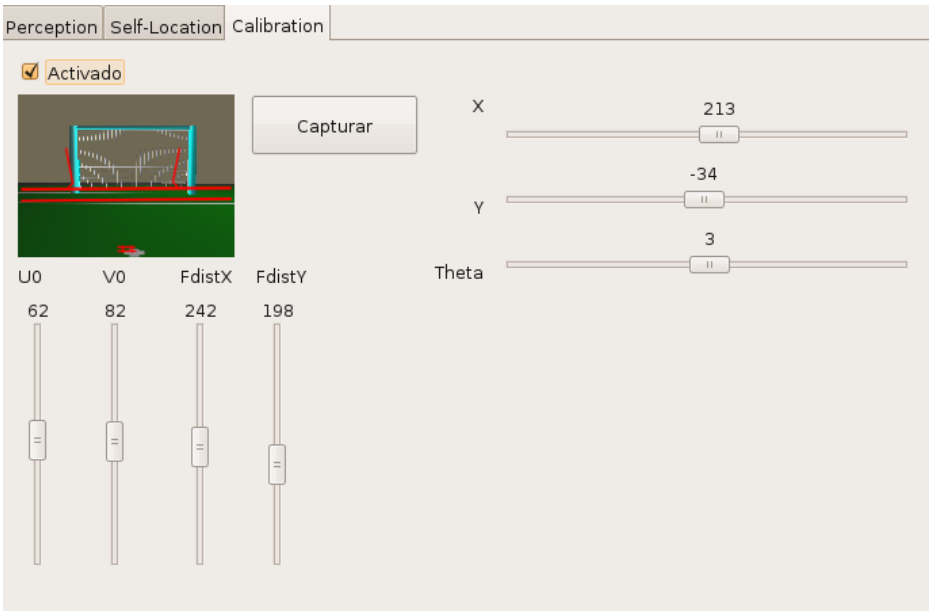
Con esta nueva RT, de nuevo debemos calcular el foco de atención con la ecuación 3.2, lo que nos permitirá configurar *Progeo* para realizar las transformaciones geométricas que necesitemos.

3.9. Componente BICA Calibration

Se ha realizado otro componente de BICA para poder obtener los parámetros intrínsecos de la cámara de una forma rápida y sencilla. Para poder utilizar *Progeo*, necesitamos conocer dos parámetros intrínsecos de las cámaras: el foco de atención $(Fdist_x, Fdist_y)$, y la posición del centro óptico (u_0, v_0).

Estos valores pueden obtenerse mediante cálculos matemáticos, como se realizó en el proyecto [Kachach, 2008], donde a partir de una imagen tomada con la cámara que queríamos calibrar y la señalización en la imagen de una serie de puntos, se obtenían los valores que buscamos.

Otra opción para obtener estos parámetros es la calibración a mano mediante prueba y error, que es la que utilizamos en este componente. Como se puede ver en la figura 3.10, el componente muestra la imagen que está viendo actualmente el robot, y permite configurar a mano tanto los valores extrínsecos del robot (X, Y, θ), como los intrínsecos candidatos.



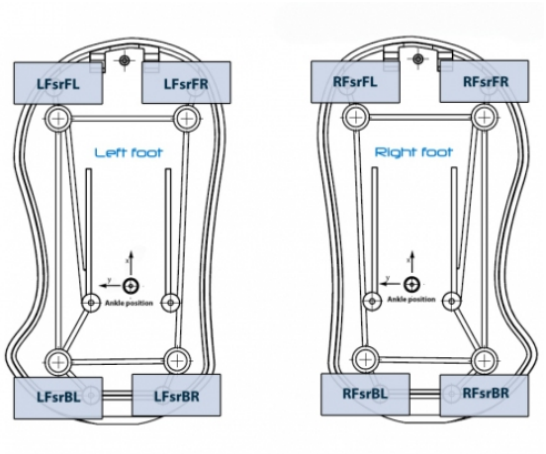
(a)

Figura 3.10: Componente para la calibración de las cámaras.

Según modificamos estos valores, podemos ver en rojo sobre la imagen real la imagen que veríamos con esa configuración de la cámara, por lo que probando con distintos valores nos quedaremos con los que más se ajusten a la imagen observada, que serán los que configuraremos en *Progeo*.

3.10. Componente BICA FSR

El último de los componentes necesarios para tener la infraestructura necesaria con la que poder trabajar, es el componente *FSR*. Este componente lee en todo momento el valor de los 8 sensores de presión con los que cuenta el robot Nao en sus pies, que son los que pueden verse en la figura 3.11.



(a)

Figura 3.11: Sensores FSR del robot Nao.

Este componente se utiliza para conocer si actualmente el robot está tocando el suelo. Para ello, en el caso de que los 8 FSR del robot sean cercanos a cero durante un periodo de tiempo determinado (y configurable), el componente indica al resto de componentes de BICA que el robot no toca el suelo, lo que significa que o está levantado por alguna persona, o está caído en el suelo.

Esta funcionalidad es utilizada por nuestros algoritmos, ya que cuando el robot deja de tocar el suelo consideramos que se ha producido un secuestro y forzamos a que se reinicien los algoritmos de autolocalización, acortando así el tiempo de recuperación ante secuestros.

Capítulo 4

Análisis de observaciones

Para poderse localizar el robot necesita obtener la información del mundo que le rodea a través de sus sensores. En nuestro proyecto, esta información se va a obtener en su mayoría de una de sus cámaras, mediante visión artificial, aunque también se utilizarán secundariamente los sensores de presión FSR del robot y la odometría de sus articulaciones.

A pesar de que el robot Nao cuenta con dos cámaras, hemos decidido utilizar solamente una de ellas (la cámara inferior) debido a que no es posible realizar visión estéreo con ellas, y que el hecho de cambiar de una cámara a otra dependiendo de lo que queramos ver es muy costoso computacionalmente y aumenta la complejidad de los algoritmos.

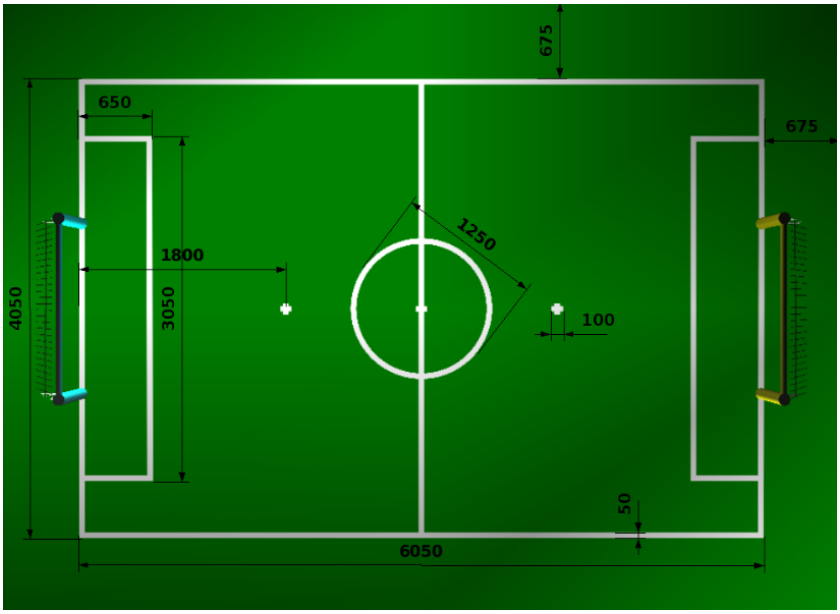
Con independencia del algoritmo que se utilice para acumular probabilidad, el primer paso es extraer información sobre los elementos relevantes de la imagen observada por el robot. Con esta información podemos conocer qué posiciones son plausibles para la localización del robot de acuerdo con la observación actual, lo que suele expresarse en forma de probabilidad.

Este análisis de las observaciones será utilizado en los dos algoritmos de acumulación que veremos en el capítulo 5, siendo usado en el algoritmo de Monte Carlo como modelo de observación, y en el algoritmo evolutivo como función de salud.

En esta sección primero vamos a hacer una descripción de los elementos que componen el mundo que rodea al robot, y que nos ayudarán a la hora de analizar la imagen de forma eficiente. A continuación explicaremos en profundidad los pasos necesarios para realizar el análisis de la imagen en 2D y por último veremos como relacionar el análisis realizado con la posición del robot.

4.1. Terreno de juego de la RoboCup

El campo de la plataforma estándar de la RoboCup es similar al que se puede ver en la imagen 4.1. Se compone de dos porterías, una azul y otra amarilla, y de una serie de líneas blancas que delimitan el campo. Hasta hace unos años, también se utilizaban 6 balizas situadas en sitios estratégicos del campo que facilitaban la autolocalización del robot, pero se eliminaron para que la competición fuese más real, aumentando así la complejidad.

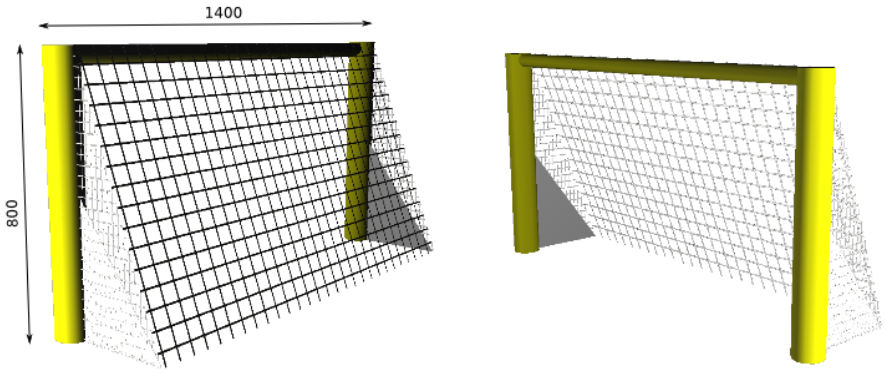


(a)

Figura 4.1: Dimensiones del campo establecidas en el reglamento de la RoboCup.

Las porterías tienen un aspecto como el de la figura 4.2, donde también pueden verse sus dimensiones. Además, hay que tener en cuenta que la pelota utilizada es de color naranja y que los partidos se componen de 6 jugadores (3 de cada equipo), por lo que nuestro robot puede ver también a los otros jugadores.

Fuera de los límites del campo puede haber cualquier tipo de objetos, como espectadores, carteles publicitarios, etc, que deberán ser tenidos en cuenta a la hora de realizar la detección de los distintos elementos del campo.



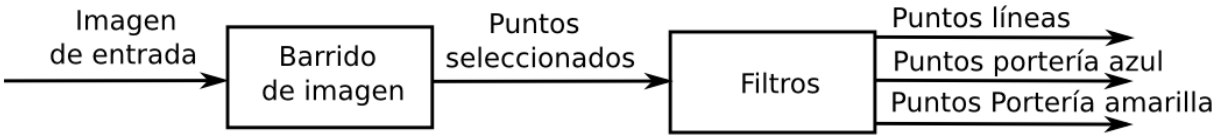
(a)

Figura 4.2: Aspecto y dimensiones de la portería.

4.2. Análisis 2D de la imagen

El primer paso realizado en la observación consiste en analizar la imagen en busca de objetos que podamos identificar y que nos permitan tener información sobre el mundo que rodea al robot.

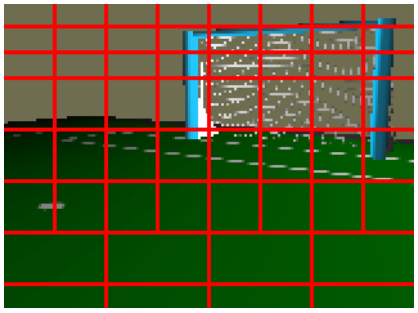
Los pasos a seguir son los que se muestran en el diagrama de la figura 4.3. Primero se realiza un barrido de la imagen en busca de puntos característicos, para después someter a cada uno de los puntos a una serie de filtros que deberán pasar para ser validados.



(a)

Figura 4.3: Esquema del análisis 2D.

En nuestro caso, hemos decidido identificar puntos característicos en la imagen que formen parte de las líneas del campo o de las porterías. Para ello, hemos realizado un barrido asimétrico de la imagen, de forma que los puntos más alejados (que estarán en la parte superior de la imagen) se comprueban de forma más exhaustiva, ya que tendrán menor resolución en la imagen. De esta forma, barreremos las partes de la imagen que se ven la figura 4.4, recorriendo más píxeles de la parte superior de la imagen, puesto que estarán más alejados.



(a)

Figura 4.4: Zonas de búsqueda para los puntos característicos.

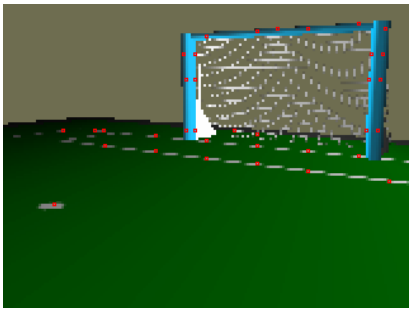
Ha sido necesario recorrer sólo ciertas zonas de la imagen puesto que uno de los factores más determinantes a la hora de realizar el análisis de la imagen es el de la eficiencia, por ello, si hubiésemos recorrido toda la imagen, el tiempo de cómputo hubiese sido mucho mayor.

A lo largo de cada una de las rectas verticales y horizontales con las que recorremos la imagen, realizamos para cada píxel varios filtros de color con el fin de identificar el color del píxel que estamos analizando. Los filtros de color realizados se corresponden con los distintos elementos que podemos encontrarnos en el campo, así, se identifican los colores amarillo y azul para las porterías, el color blanco para las líneas y el color naranja para la pelota, clasificando como desconocido el color de cualquier píxel que no concuerde con ninguno de los anteriores.

Al realizar esta identificación en cada recta, buscamos los puntos donde se produce un cambio de color, detectando de esta forma los bordes en la imagen. Así pues, para encontrar por ejemplo la portería azul, nos quedaremos con los píxeles donde haya un cambio de color entre el azul y cualquier otro color.

Con esto obtendremos una serie de puntos, como los que se destacan en rojo en la imagen 4.5.

Entre los puntos obtenidos pueden encontrarse falsos positivos que harían que el algoritmo se comportase de un modo incorrecto, por ello, el siguiente paso a realizar es intentar encontrar estos falsos positivos y eliminarlos, consiguiendo así que los puntos seleccionados sean más fiables. Esto se ha realizado con varios métodos que se describen a continuación.



(a)

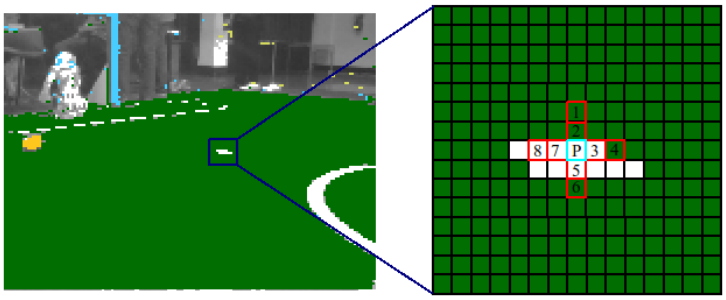
Figura 4.5: Puntos característicos seleccionados en el barrido.

Puntos aislados

El primer filtro que debe pasar cada uno de estos puntos, es el de no estar aislado en la imagen. Es decir, una vez detectado un punto de un color determinado, este punto debe estar rodeado de varios puntos del mismo color.

Este filtro es necesario puesto que, en las imágenes reales, puede producirse ruido y pueden detectarse puntos sueltos de un color que en la realidad no corresponden a ningún objeto relevante para la detección.

Esta comprobación se realiza identificando el color de los píxeles que rodean en la imagen al píxel seleccionado, como se muestra en la figura 4.6. En esta imagen podemos ver cómo, para un píxel P, se comprueban los píxeles de su alrededor marcados del 1 al 8.



(a)

Figura 4.6: Reconocimiento de puntos aislados.

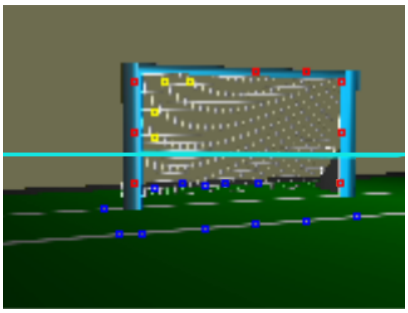
Para considerar un punto válido, hemos establecido que al menos 3 de los puntos comprobados alrededor del píxel seleccionado deben de ser del mismo color, en cualquier otro caso, consideramos a este punto como no válido.

Horizonte

En este caso, nos interesa descartar los puntos detectados que estén demasiado lejos del robot. Al conocer la odometría del robot, podemos calcular puntos en el espacio 3D respecto a su posición relativa utilizando el componente *Kinematics* que describimos en el capítulo anterior.

Gracias a esto y al conocimiento del campo, podemos saber que a partir de una determinada distancia los puntos detectados en el suelo no serán válidos, ya que se saldrán de los límites del campo.

Así, hemos establecido que los puntos blancos que se encuentren a más de 6 metros del robot se considerarán que están fuera del campo, y por lo tanto serán falsos positivos. En la imagen 4.7 podemos ver un ejemplo de lo anterior, donde se puede apreciar como los puntos blancos que sobrepasan la línea de horizonte de 6 metros son descartados (los dibujados en amarillo).



(a)

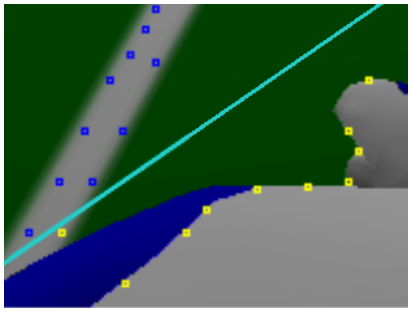
Figura 4.7: Eliminación de puntos demasiado alejados.

Este método no sirve para las porterías, puesto que al estar estos puntos por encima del suelo, no es posible calcular la distancia que les separa del robot.

Cuerpo del robot

Al igual que hemos calculado los puntos que se encuentran demasiado lejos del robot, también podemos calcular los que se encuentran muy cerca de éste, puesto que al detectar un punto muy cercano, en realidad podemos estar confundiéndonos con el propio cuerpo del robot.

Este caso se muestra en la imagen 4.8, donde los puntos que estén por debajo de 15 cm respecto al robot son descartados.



(a)

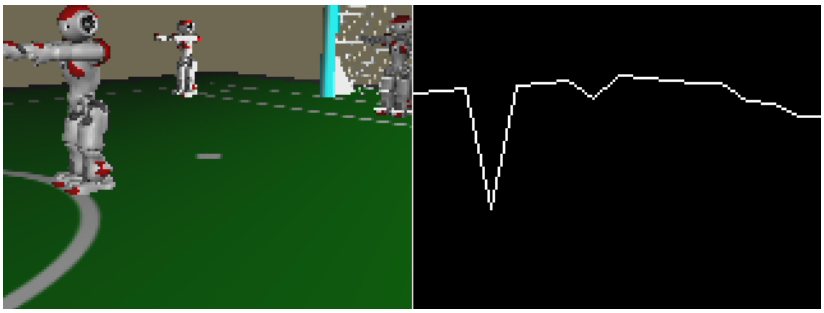
Figura 4.8: Eliminación de puntos demasiado cercanos.

En este caso se descartan tanto las líneas blancas como las de las porterías, puesto que el robot también puede tener partes de color azul.

Fin del campo

En algunas ocasiones, no nos es suficiente con utilizar el horizonte para descartar puntos alejados del robot, como por ejemplo cuando nos encontramos cerca de una portería, lo que hará que tengamos bastantes falsos positivos. Por ello, hemos decidido también utilizar el fin del campo, es decir, calculamos dónde termina el campo en función del color verde detectado en la imagen.

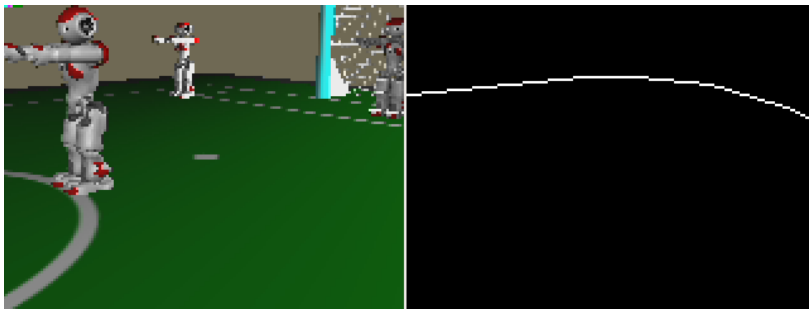
Este cálculo se compone de dos pasos que se realizan por separado. Primero se recorre la imagen en vertical cada 10 píxeles y se comprueba dónde comienza el color verde para cada columna. Si utilizásemos directamente estos píxeles calculados, obtendríamos un fin del campo como el que puede verse en la imagen 4.9.



(a)

Figura 4.9: Primer paso para calcular el fin del campo.

Por ello, en necesario realizar un segundo paso con estos píxeles, mediante un algoritmo conocido como Convex Hull [1]. Este algoritmo consiste en que dados N puntos, debemos obtener el subconjunto convexo con el menor número de puntos pertenecientes a N que haga que los N puntos se encuentren en su interior. Es decir, en nuestro caso, dados los puntos calculados obtendremos el menor subconjunto convexo que haga que todos los puntos se encuentren por debajo del subconjunto obtenido, calculando de esta forma el fin del campo correctamente, como puede verse en la imagen 4.10.



(a)

Figura 4.10: Fin del campo tras calcular el Convex Hull.

Al igual que sucedía en el caso del horizonte, este filtro sólo nos permite eliminar los puntos pertenecientes a las líneas, puesto que las porterías pueden encontrarse lícitamente por encima del fin del campo.

Blobs de la portería

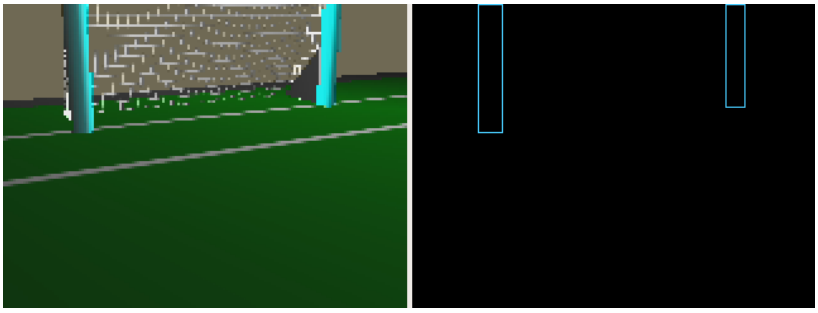
La mayoría de filtros que hemos visto hasta ahora sólo afectaban a las líneas del campo, por ello, hemos decidido añadir un nuevo filtro que sólo es aplicable a las porterías, y que se basa en un nuevo elemento conocido como *blob*.

Los *blobs* se calculan a partir de la librería *cvBlob*[2], que engloba en uno o varios objetos (*blobs*) los elementos que aparecen en la imagen. Para ello, a partir de la imagen filtrada con los colores de la portería utilizamos esta librería para que nos devuelva los objetos que aparecen en la imagen.

Los *blobs* obtenidos de la librería son posteriormente validados en función de distintos filtros, como por ejemplo su relación de aspecto, su tamaño, etc, para aceptar sólo los que

[1] http://en.wikipedia.org/wiki/Convex_hull
[2] http://code.google.com/p/cvblob/

puedan corresponderse con una portería. En la figura 4.11 podemos ver un ejemplo de cómo se detectan los postes de la portería en una imagen.



(a)

Figura 4.11: *Blobs* con los postes de la portería.

En nuestro algoritmo hemos utilizado directamente estos *blobs* ya validados, comprobando que todos los puntos detectados de la portería se encuentren dentro de algunos de los *blobs*, y descartándolos en otro caso.

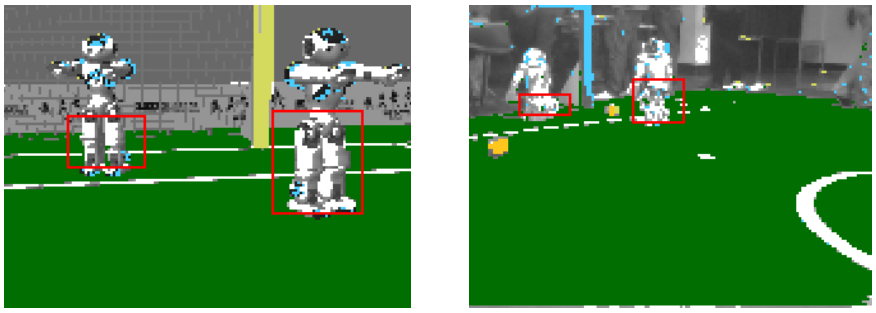
Obstáculos

El último filtro que hemos desarrollado ha sido el de la detección de obstáculos en la imagen. Ya que durante la competición real el robot va a jugar con otros robots en el campo, es importante detectar a estos robots, especialmente para no confundir el color blanco de estos robots con líneas del campo.

Para la detección de estos obstáculos, recorremos la imagen en columnas de forma similar a como lo hacíamos para detectar el fin del campo, pero esta vez buscamos segmentos de color blanco con una longitud mínima.

Una vez guardados estos segmentos por columnas, los agrupamos en bloques, uniendo los segmentos que se encuentren juntos en la imagen. Por último, comprobamos la distancia a la que se encuentran estos bloques utilizando el componente *Kinematics* y comprobamos sus dimensiones. Para clasificar los bloques obtenidos como obstáculos, tendrán que tener un ancho mínimo que dependerá de la distancia a la que se encuentre el bloque, clasificando como obstáculos los que sobrepasen el umbral. Podemos ver algunos ejemplos de la detección de obstáculos en las figura 4.12.

Tras detectar los obstáculos, comprobamos que cada punto detectado no se encuentre dentro de ninguno de los bloques detectados como obstáculo, y de ser así, también es descartado.



(a) (b)

Figura 4.12: Detección de obstáculos en el simulador (a) y en el robot real (b).

4.3. Información de posición

El siguiente paso es obtener información de posición a partir de la imagen observada, es decir, saber si una posición es verosímil a la luz de una imagen. Para ello, dada una posición (X, Y, θ), podemos calcular en qué grado se parece la imagen que vería el robot en esa posición (imagen teórica) con la imagen que estamos observando (imagen real). Si son parecidas esa posición es plausible, mientras que si son muy distintas esa posición es improbable.

Esto podría realizarse comprobando directamente la correlación píxel a píxel entre la imagen real y la teórica, sin embargo, esto es muy poco eficiente, por lo que se van a utilizar los puntos característicos obtenidos como hemos visto en la sección anterior.

Así, obtenemos la "correlación" entre las dos imágenes a partir de la distancia existente entre cada punto encontrado en la imagen real y su homólogo en la imagen teórica, lo que da como resultado una probabilidad que indica cómo de verosímil es la posición dada respecto de la imagen observada.

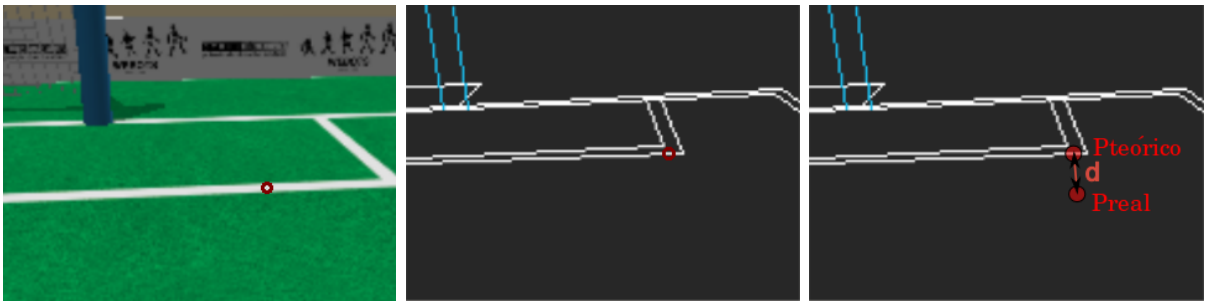
Es decir, para cada punto seleccionado en la imagen real, buscamos en la imagen teórica el punto más cercano del mismo color, lo que nos proporciona una distancia en píxeles que nos servirá para calcular la probabilidad final.

En la figura 4.13 podemos ver un ejemplo de lo anterior, donde dado un píxel en la imagen real P_{real} (4.13(a)), buscamos el píxel del mismo color más cercano en la imagen teórica $P_{teórico}$ (4.13(b)), que se encontrará a una distancia d, siendo $d \geq 0$.

Al realizar el mismo procedimiento para cada punto detectado en la imagen real, podremos calcular la probabilidad final mediante la media de las distancias, como se ve en la ecuación 4.1. De esta forma, cuando la distancia media sea igual o mayor a 50 píxeles,

la probabilidad sera 0, y aumentará a medida que disminuya la distancia media.

$$P = 1 - \frac{\sum_{i=0}^{N} \frac{D_i}{N}}{50} \tag{4.1}$$



(a) (b) (c)

Figura 4.13: Distancia entre la imagen real y la teórica para un punto dado.

Como las porterías son más concluyentes que las líneas a la hora de determinar la información de la posición del robot, hemos decidido calcular la probabilidad de las líneas y las porterías por separado mediante la ecuación anterior. De modo que en el caso de que veamos sólo las líneas o sólo las porterías, solamente se tendrá en cuenta la probabilidad del objeto que vemos, mientras que si vemos ambos, la probabilidad vendrá dada por su media:

$$P_{final} = \frac{P_{porterias} + P_{lineas}}{2} \tag{4.2}$$

Para comparar un punto de la imagen real con el de la imagen teórica, en un principio generábamos en tiempo de ejecución la imagen teórica completa para la posición del robot utilizando la librería *Progeo*, con lo que obteníamos una imagen simulada similar a la que hemos visto en la figura 4.13(b). Sin embargo, nos dimos cuenta de que este cálculo era muy costoso computacionalmente, así que decidimos calcular estas distancias mediante información precalculada como veremos a continuación.

4.3.1. Precálculo de puntos cercanos en 3D

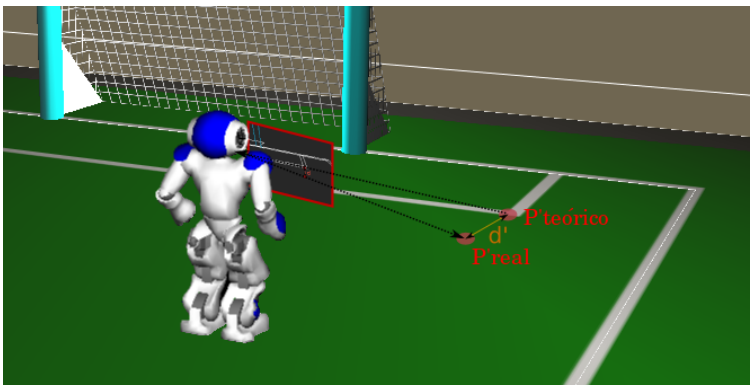
Una primera aproximación para ahorrar tiempo de cómputo es tener ya precalculadas las distancias para cada píxel en cada posible posición del robot, pero esto no es posible, puesto que al contar el robot con 6 grados de libertad las posiciones a tener en cuenta serían demasiadas y requeriría mucha memoria.

Sin embargo, como la posición candidata del robot es conocida, mediante el componente *Kinematics* es posible llevar los píxeles seleccionados en la imagen a 3D siempre que conozcamos un plano en el que retroproyectar los puntos. Así, en el caso de las líneas, sabemos que siempre se encuentran en el plano del suelo (es decir, con altura 0), mientras que las porterías siempre se encuentran en el plano vertical a ambos lados del campo.

De esta forma, al tener puntos en 3D, el número de puntos posibles es mucho menor si discretizamos el campo cada pocos centímetros. Así, como lo que nos interesa es poder comparar puntos en la imagen, hemos precalculado pares de puntos en 3D que relacionan un punto en 3D (que correspondería con el P_{real}) con el punto más cercano donde se encuentra un objeto de ese tipo (correspondiente al $P_{teórico}$).

Estos pares de puntos, calculados cada 2.5 centímetros, han sido guardados en dos archivos (uno para las líneas y otro para las porterías), y son cargados en memoria al comenzar el algoritmo.

Así, el nuevo mecanismo para calcular la distancia es el que se muestra en la figura 4.14. Dado el mismo punto P_{real} en la imagen que vimos en 4.13(a), retroproyectamos este punto 2D en el plano seleccionado para obtener el punto en 3D P'_{real}. Después, consultamos el punto precalculado en 3D más cercano a P'_{real}, y así obtenemos $P'_{teórico}$. Este $P'_{teórico}$ en 3D lo proyectamos en la imagen y obtenemos el punto $P_{teórico}$ en 2D que buscábamos, pudiendo así calcular la distancia entre ellos.



(a)

Figura 4.14: Distancia entre la imagen real y la teórica en 3D.

Con esto, el tiempo de cómputo del algoritmo se ha reducido considerablemente (cerca de 10 veces menor en esa parte del algoritmo), puesto que lo que antes se realizaba simulando una imagen teórica completa, ahora lo podemos conseguir llamando a dos funciones muy eficientes del componente *Kinematics* y consultando datos en memoria.

4.3.2. Descartes aleatorios

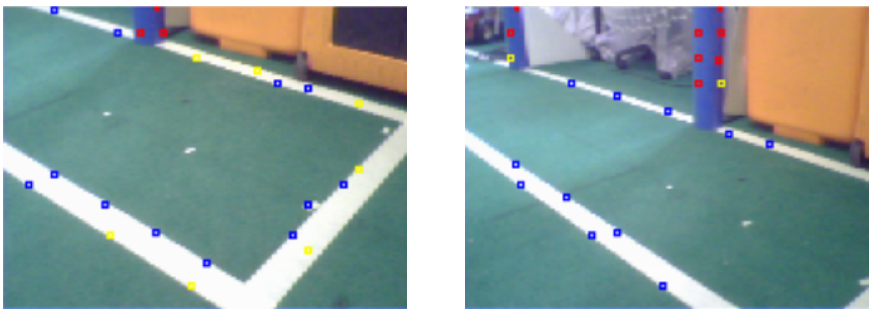
Como podemos ver, el coste temporal de la función aumenta de forma lineal según aumentamos el número de puntos seleccionados en la imagen. Para evitar esto, podríamos seleccionar un número de puntos máximo seleccionados de forma aleatoria entre el total de puntos encontrados, aunque esto podría hacer que la probabilidad resultante fuese distinta según los puntos que se descartasen.

Por ello, decidimos comprobar el comportamiento de la función tras este descarte aleatorio, como podemos ver la imagen 4.16.

Viendo que la pérdida de precisión es prácticamente nula independientemente de los puntos seleccionados, y los beneficios que nos puede proporcionar en cuanto a reducción de coste computacional, hemos decidido establecer un máximo de puntos a tener en cuenta en cada observación, que hemos fijado en 12 puntos para las líneas y 8 para las porterías, es decir, como máximo se tendrán en cuenta 20 puntos.

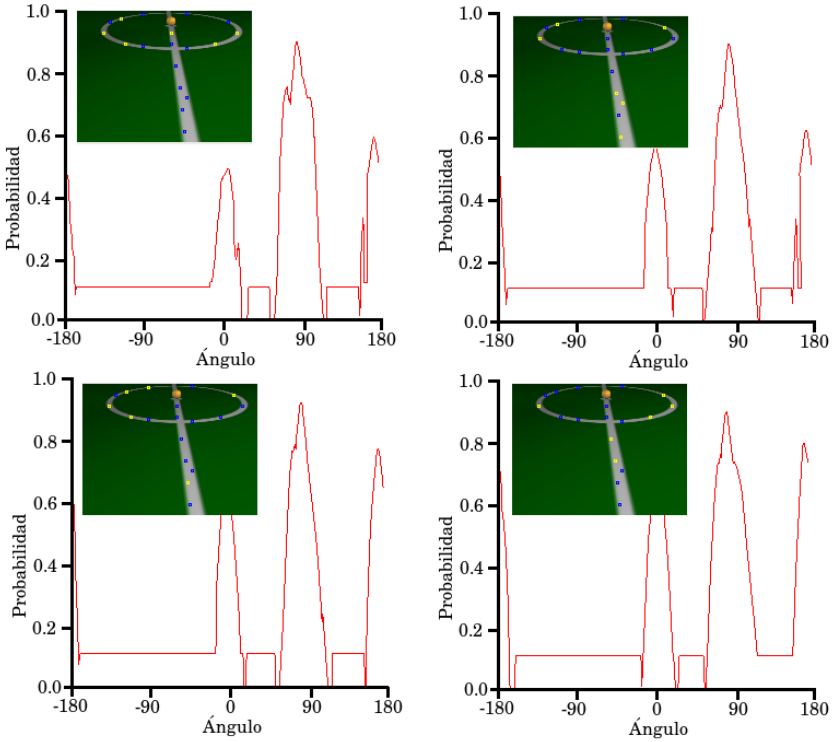
En el caso de que el número de puntos sea mayor, se descartarán puntos de forma aleatoria hasta llegar al máximo posible. Esto, además de hacer que el tiempo de ejecución se mantenga lo más bajo posible, también ayuda a la hora de evitar falsos positivos, puesto que en el caso de que se produzca alguno, es posible que quede descartado en la aleatoriedad.

En las imágenes de la figura 4.15 podemos ver en amarillo los puntos descartados de forma aleatoria en las líneas y las porterías.

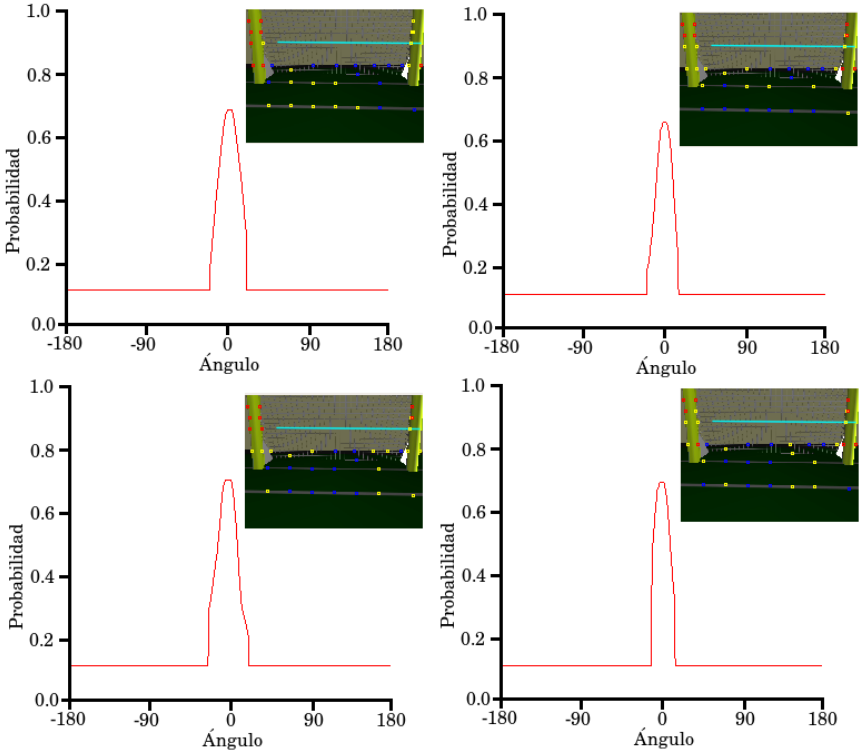


(a) (b)

Figura 4.15: Puntos descartados de forma aleatoria en las líneas y porterías.



(a)



(b)

Figura 4.16: Selección aleatoria en líneas (a) y porterías (b).

4.3.3. Fiabilidad de la observación

La localización del robot en el campo es tenida en cuenta por otros componentes de BICA a la hora de determinar la acción a realizar. Debido a esto, es necesario proporcionar un mecanismo para identificar cuándo la localización calculada es fiable y cuándo no lo es.

Como veremos en el siguiente capítulo, los algoritmos desarrollados proporcionan un parámetro entre 0 y 1 que determina la fiabilidad de la localización calculada. Este parámetro es calculado a partir de varios factores, entre los cuales se encuentra la calidad de la observación instantánea.

Por ello, una vez analizada la imagen como hemos visto, es necesario conocer la calidad de esta imagen en función de cuáles y cuántos son los puntos característicos obtenidos. Este cálculo proporciona mediante una heurística un valor entre 0 y 1 que determinará la calidad de la imagen y que viene dada por 4 aspectos:

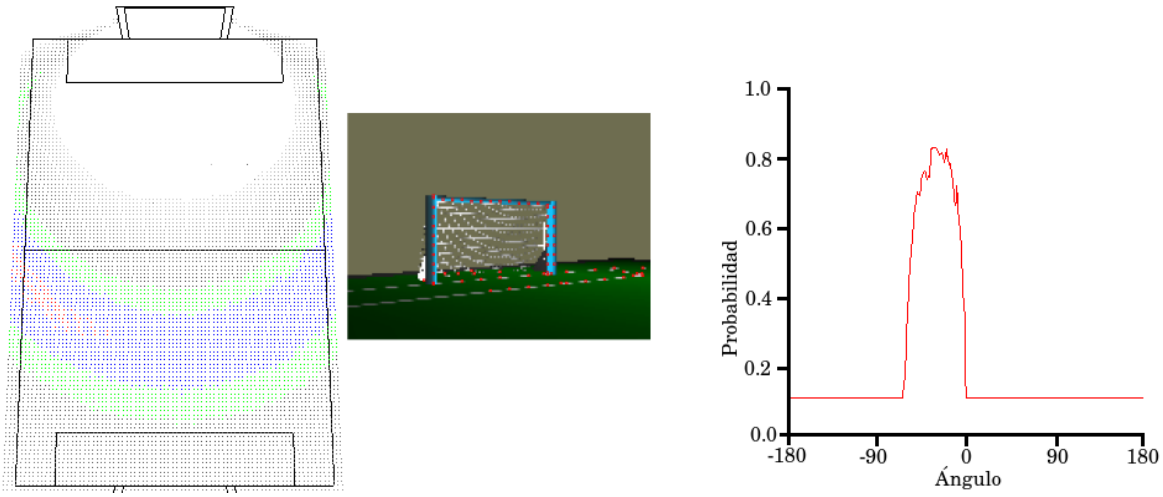
- Líneas encontradas: En el caso de ver algún punto correspondiente a las líneas, se sumará 0.2 a la calidad.
- Número de líneas visibles: Número de puntos pertenecientes a líneas (desde 0 hasta el máximo permitido, 12), puede sumar hasta 0.2.
- Porterías encontradas: En el caso de ver algún punto correspondiente a las porterías, se sumará 0.2 a la calidad.
- Número de porterías visibles: Número de puntos pertenecientes a porterías (desde 0 hasta el máximo permitido, 8), puede sumar hasta 0.4.

De esta forma, podemos medir la calidad de la imagen observada, pudiendo llegar a 1 (mejor calidad) en el caso de ver el máximo número de puntos permitidos tanto para las líneas como para las porterías.

4.4. Validación experimental

Hemos validado experimentalmente la información de posición propuesta tanto en simuladores como en el robot real, analizando su comportamiento ante diversas situaciones.

En las figuras mostradas en 4.17 podemos ver el resultado de un experimento típico. En la figura 4.17(a) mostramos cuál sería la probabilidad calculada para cada posible posición del robot a partir de la imagen de entrada (rojo es la mayor probabilidad y negro la menor), mientras que en la figura 4.17(b) vemos el resultado obtenido en cada ángulo posible dada ya una posición fija.



(a) (b)

Figura 4.17: Probabilidad calculada para todas las posiciones (a) y para todos los ángulos (b).

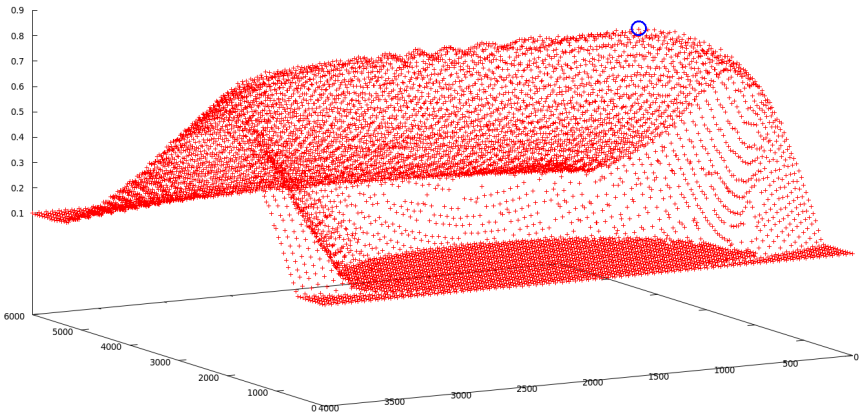
Puede verse cómo la zona de mayor probabilidad se corresponde con el lugar donde se encuentra realmente el robot (lateral izquierdo del campo), y que una vez establecida la posición correcta del robot, el ángulo de rotación que tiene una mayor probabilidad también es el adecuado (alrededor de -30 grados).

El tiempo de ejecución final del análisis 2D de la imagen desarrollado en el robot real ha sido de 1.5 milisegundos de media, con lo que hemos conseguido unos tiempos bastante eficientes que hacen que nuestro análisis pueda utilizarse junto con el resto del código sin perjudicar al resto de componentes de BICA. Por su parte, el tiempo que se tarda en obtener la información de posición en el robot real es de aproximadamente 100 microsegundos.

4.4.1. Discriminación de la información de posición

El hecho de que la discriminación en la información de posición fuese muy grande o muy pequeña afecta negativamente a la convergencia de las muestras, tanto en el algoritmo de Monte Carlo como en el evolutivo, por ello, debemos asegurarnos de que esto no suceda.

Para ello, hemos visualizado en 3D las probabilidades que acabamos de ver en la figura 4.17(a), lo que puede verse en la figura 4.18.



(a)

Figura 4.18: Probabilidad calculada en 3D.

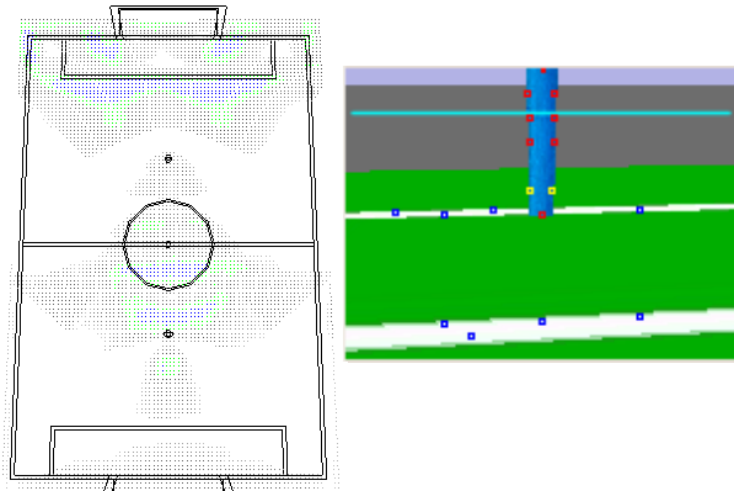
Como puede verse, el crecimiento de la probabilidad es progresivo y tiene su máximo donde debe (lateral izquierdo del campo), por lo que la discriminación realizada es suficientemente suave y firme como para permitir la convergencia de forma correcta. Sin embargo, también podemos ver cómo este crecimiento cuenta con máximos locales, lo que habrá de ser tenido en cuenta a la hora de diseñar los algoritmos de localización.

4.4.2. Efecto de las simetrías

Al existir simetrías en el campo, podemos encontrarnos con situaciones como la de la figura 4.19, en donde a partir de un único poste no podemos asegurar cuál es nuestra posición, y se obtiene una alta probabilidad en varias zonas del campo, sobre todo cerca de la portería.

Sorprende el hecho de que haya una alta probabilidad detrás del centro del campo, aunque es lógico puesto que en esa posición tendríamos una línea delante del robot (la línea del centro del campo) y veríamos alguno de los postes en la zona central de la imagen.

Para solucionar esto, podríamos haber tenido en cuenta el hecho de no ver otros objetos que deberíamos ver desde donde nos encontramos. Por ejemplo, en este caso, desde el centro del campo deberíamos ver el otro poste, algo que no sucede. Sin embargo, si añadiésemos esta característica tendríamos probabilidades muy bajas en el caso de existir oclusiones, por lo que decidimos no tenerlo en cuenta.



(a)

Figura 4.19: Efecto de las simetrías.

Mayor sería la incertidumbre en el caso de no ver ninguna de las dos porterías y ver únicamente líneas, ya que dependiendo de la observación, podría darse el caso de que tuviésemos una probabilidad muy alta en casi todo el campo.

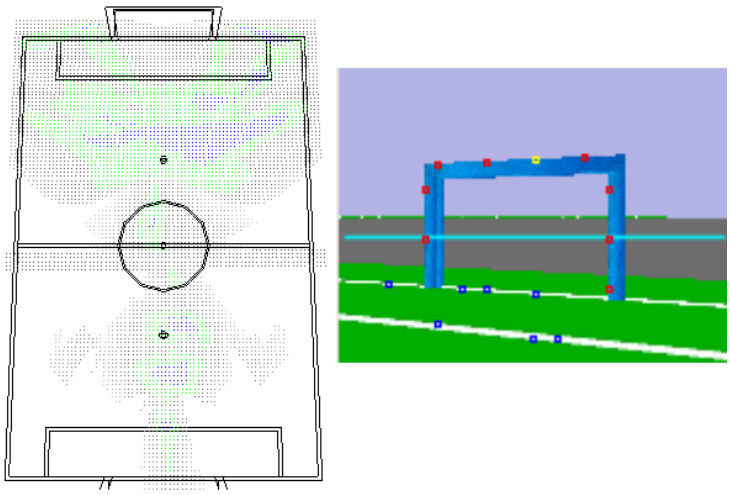
4.4.3. Efecto de las oclusiones

Como acabamos de decir, las oclusiones no afectan negativamente a probabilidad calculada, ya que sólo se mide la compatibilidad de lo que se ve en la imagen con lo que se debería ver en la posición dada, pero sin embargo, no se mide si todo lo que se debería ver desde esa posición realmente se ve.

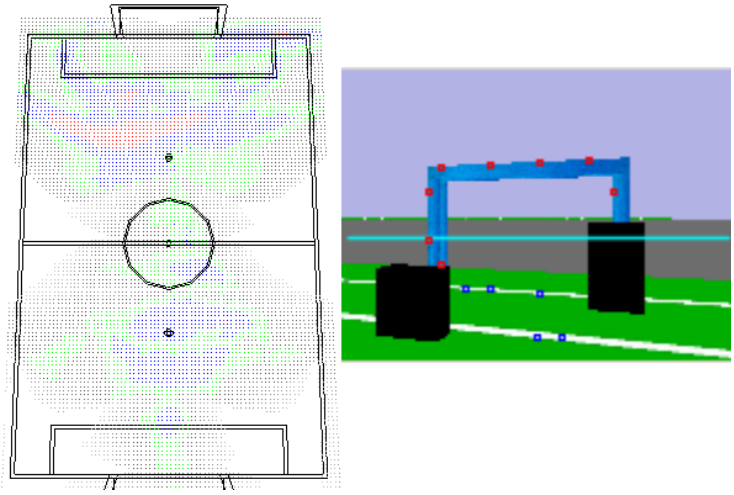
En la figura 4.20 podemos ver un ejemplo de cómo influirían las oclusiones en el resultado obtenido. En la primera imagen, mostramos cuál sería el resultado en caso de que no existiesen oclusiones, siendo la zona con mayor probabilidad la que se encuentra a 1-2 metros de la portería por la zona central del campo.

En la segunda imagen hemos situado dos elementos que simularían el hecho de que se encontrasen otros dos robots en el campo en esas posiciones. Como podemos ver, al no

penalizar lo que no vemos en la observación, las zonas con alta probabilidad de la primera imagen continúan, sin embargo, el área que abarcan estas zonas es mucho mayor, e incluso la probabilidad ha aumentado en algunos lugares simétricos a la posición real del robot (zona central-izquierda en color rojo).



(a)



(b)

Figura 4.20: Efecto de las oclusiones.

Así, el hecho de que se produzcan oclusiones dificulta la localización del robot, pudiendo hacer que su localización sea errónea si estas oclusiones se mantienen durante muchas observaciones.

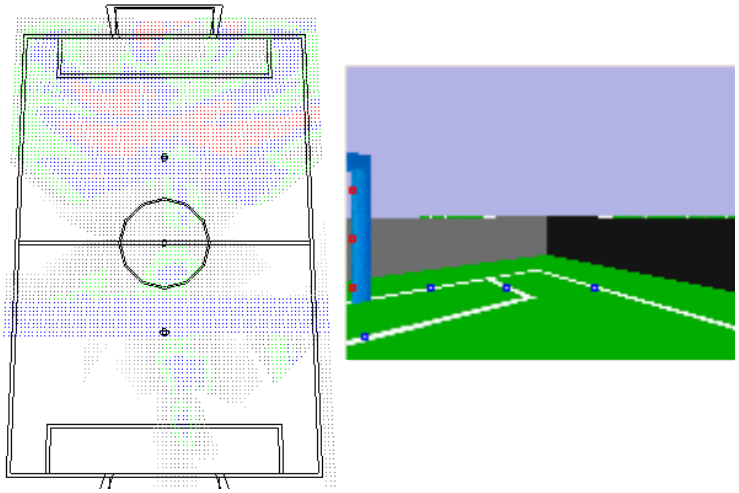
4.4.4. Falsos positivos

Un aspecto que afecta en gran medida al resultado es el de aceptar como válidos puntos que no lo son, teniendo así falsos positivos. Estos falsos positivos podrían hacer que cambiase totalmente el resultado final, sobre todo en el caso de que el falso positivo se produjese con las porterías.

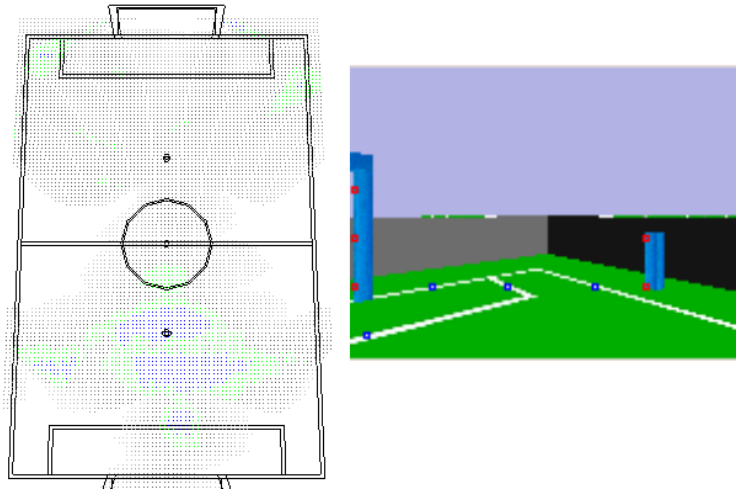
En la figura 4.21 podemos ver un ejemplo de lo anterior. En la primera imagen vemos cómo de nuevo las zonas con mayor probabilidad son las que están a 1-2 metros de la portería y en la parte de detrás del centro del campo. Al añadir un falso poste en la segunda imagen, se produce un falso positivo que hace que el resultado cambie totalmente, dando a las zonas cercanas a la portería muy poca probabilidad, lo que haría que fallasen los algoritmos de localización.

Otro caso en el que los falsos positivos afectarían mucho a los algoritmos, es en el caso de que en una observación no se fuese a obtener ningún punto característico en la imagen (por ejemplo, cuando el robot mira hacia el suelo y sólo ve el color verde del campo). En esa situación el análisis de la imagen desarrollado no encontraría ningún punto y los algoritmos no tendrían en cuenta la observación actual. Si en esta situación, se produce un falso positivo, ya sea con las porterías o con las líneas, el análisis de la imagen sí que encontraría puntos característicos, haciendo que las probabilidades calculadas tuviesen valores impredecibles (incluso 0 para todas las posiciones en caso de que la información fuese incongruente). Esto haría que las posiciones se evaluasen con valores totalmente erróneos, perjudicando a los algoritmos de localización.

Es por ello que se ha hecho especial hincapié en este aspecto, realizando numerosos filtros para validar cada uno de los puntos característicos que percibimos en la imagen, como se ha visto a lo largo de este capítulo.



(a)



(b)

Figura 4.21: Falsos positivos.

Capítulo 5

Algoritmos de localización

A la hora de calcular la localización del robot, una sola observación no suele ser concluyente, ya que puede que la observación no contenga la suficiente información como para localizarse o que se hayan producido errores en el análisis de ésta debido a oclusiones, falsos positivos, etc. Por ello, para obtener una localización fiable es necesario contar con algoritmos que sean capaces de subsanar estos inconvenientes mediante la acumulación temporal de muchas observaciones.

Existen diversas técnicas a la hora de localizar al robot en un mundo conocido, como las que ya hemos visto en el capítulo 1.4, que pueden dividirse en dos bloques principales: las constructivas, que pretenden obtener la solución final de forma analítica a partir de la información recibida, y las abductivas, que calculan la posición en la que se encuentra el robot a partir de suposiciones que se validan a posteriori desde la información espacial y descartando las incompatibles.

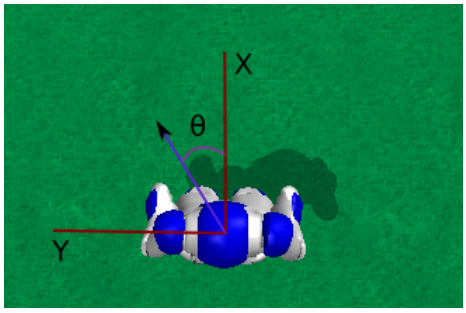
Los algoritmos desarrollados en este proyecto tienen un enfoque abductivo, ya que comparan la imagen real obtenida con las imágenes teóricas que se obtendrían si el robot estuviese en una posición determinada.

Hemos diseñado y programado dos algoritmos de localización, uno utilizando el método de Monte Carlo y otro mediante un algoritmo evolutivos, y que serán explicados a lo largo de este capítulo. Además, estos algoritmos han sido validados experimentalmente y serán comparados entre sí en el capítulo 6.

5.1. Localización por método de Monte Carlo

El método de Monte Carlo utiliza un número N de elementos (llamados partículas), que representan distintas posiciones en las que puede encontrarse el robot en un momento determinado. A través de estas partículas, el algoritmo de Monte Carlo trata de encontrar la posición real del robot de una forma eficiente.

En nuestro caso, las partículas guardan información sobre la posición del robot (X, Y, θ), como se muestra en la figura 5.1, y la probabilidad obtenida con la información de posición que vimos en la sección 4.3.



(a)

Figura 5.1: Ejes de coordenadas en la localización.

Inicialmente las partículas se posicionan en el campo de forma aleatoria, para lo cual los valores (X, Y, θ) de cada partícula son inicializados aleatoriamente dentro de unos límites establecidos. Una vez inicializadas las partículas, el método de Monte Carlo recalcula la posición de cada una de estas partículas en cada iteración utilizando 3 pasos fundamentales:

- Modelo de movimiento: En el caso de que conozcamos el movimiento que ha realizado el robot respecto a la última iteración, debemos modificar todas las partículas para que modifiquen su posición teniendo en cuenta este movimiento incremental. El modelo de movimiento es de mucha utilidad en robots que se mueven con ruedas, puesto que su odometría es bastante fiable, sin embargo, en el caso de los robots con patas, como sucede con el robot humanoide Nao, este movimiento no es tan sencillo de calcular, como veremos en la sección 5.1.1.

- Modelo de observación: El siguiente paso es calcular la probabilidad de cada una de las partículas en el mundo, para ello es necesario comparar la imagen real que está observando el robot con la imagen teórica que tendría el robot de encontrarse

en esa posición. Esto se realiza con el análisis 2D de la imagen y la información de posición (función de coste) que vimos en el capítulo 4.

- Remuestreo: El último paso a realizar es el remuestreo de las partículas, que consiste en renovar las partículas de la siguiente iteración, manteniendo algunas de ellas y generando nuevas en función de su probabilidad, lo que veremos en la sección 5.1.2.

Con estos tres pasos, las partículas más probables generarán más partículas próximas a ellas en la siguiente iteración, de modo que con el paso de las iteraciones se creará una nube de partículas que se desplazará hacia las posiciones más compatibles con las observaciones tomadas, que se corresponderán idealmente con la localización real del robot.

Además de lo anterior, en el caso de que ninguna partícula supere una probabilidad mínima preestablecida, lo que significará que el robot no está bien localizado o que se ha producido un secuestro, se reiniciarán todas las partículas con valores aleatorios al igual que en la inicialización.

5.1.1. Modelo de movimiento

Como ya hemos dicho, obtener el movimiento realizado por el robot cuando se utilizan robots con patas, como es nuestro caso, es una tarea complicada y los errores producidos en la medición pueden ser muy altos. A pesar de ello, hemos intentado desarrollar un modelo de movimiento que se ajustase lo más posible a la realidad, teniendo siempre en cuenta que pueden existir errores odométricos.

El objetivo de este modelo de movimiento es el de calcular el movimiento incremental en (X, Y, θ) del robot desde la última vez que se midió, y aplicar este desplazamiento y giro a cada una de las partículas que existan en ese momento en el algoritmo.

Para calcular este movimiento, hemos hecho uso de la función de Naoqi llamada *getPosition*, que devuelve el movimiento realizado por el robot en (X, Y, θ) desde que se encendió el robot.

Los pasos realizados para calcular el movimiento respecto a la última iteración son:

- Guardamos en cada iteración el valor actual devuelto por la función *getPosition* de Naoqi, para poder comparar este valor con el de la siguiente iteración.

- Comprobamos si el robot se ha movido, comparando los nuevos valores de *getPosition* con los guardados previamente. En el caso de que no haya un movimiento significativo

consideramos que el robot está parado, por lo que no es necesario actualizar la posición de las partículas.

- Calculamos la diferencia en distancia y orientación entre los valores actuales y los almacenados como se muestra en la ecuación 5.1, donde P_{new} son los valores de la iteración actual (X, Y, θ), P_{last} son los valores guardados en la última iteración, y ϵ es un factor de corrección al error sistemático que se produce en la odometría del robot. Este factor varía en el robot real y en los simuladores, por lo que debe ser obtenido a través de un archivo de configuración.

$$P_{diff} = (P_{new} - P_{last}) * \epsilon \tag{5.1}$$

Así, el giro en relativas (P_{rel_θ}) es directamente el valor P_{diff_θ} calculado, mientras que para calcular el desplazamiento relativo entre cada iteración (P_{rel_X}, P_{rel_Y}) es necesario aplicar una matriz de rotación teniendo en cuenta el ángulo θ en el que se encontraba el robot en la última iteración, mediante la ecuación 5.2:

$$\begin{pmatrix} P_{rel_X} \\ P_{rel_Y} \end{pmatrix} = \begin{pmatrix} cos(-P_{last_\theta}) & -sen(-P_{last_\theta}) \\ sen(-P_{last_\theta}) & cos(-P_{last_\theta}) \end{pmatrix} * \begin{pmatrix} P_{diff_X} \\ P_{diff_Y} \end{pmatrix} \tag{5.2}$$

Una vez obtenido el movimiento realizado desde la última iteración debemos aplicar este movimiento a cada partícula. Para ello, el ángulo final (P_{final_θ}) se calcula sumando el ángulo en relativas calculado (P_{rel_θ}) y el ángulo de la partícula (P_{part_θ}), mientras que para obtener los valores $(P_{final_X}, P_{final_Y})$ finales debemos aplicar una matriz RT utilizando el movimiento relativo calculado y la posición de la partícula (P_{part_X}, P_{part_Y}), como se ve en la ecuación 5.3:

$$\begin{pmatrix} P_{final_X} \\ P_{final_Y} \\ 1 \end{pmatrix} = \begin{pmatrix} cos(P_{part_\theta}) & -sen(P_{part_\theta}) & P_{part_X} \\ sen(P_{part_\theta}) & cos(P_{part_\theta}) & P_{part_Y} \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} Prel_x \\ Prel_y \\ 1 \end{pmatrix} \tag{5.3}$$

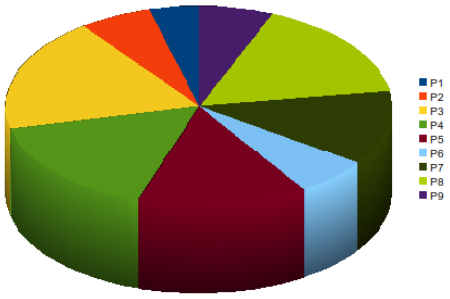
5.1.2. Remuestreo

Tras calcular la probabilidad de cada partícula mediante la función de coste, el último paso a realizar en el algoritmo de Monte Carlo es el remuestreo de partículas. El remuestreo utiliza estas probabilidades para calcular las partículas que se utilizarán en la siguiente iteración.

Esta selección de las partículas se realiza mediante dos elementos de selección, el elitismo y la ruleta.

- Elitismo: El elitismo consiste únicamente en seleccionar las N mejores partículas, determinadas por su probabilidad, con el fin de preservar estas partículas sin modificaciones en la siguiente iteración. Esto nos permite mantener a las mejores partículas para asegurar que la posición del robot no empeore al utilizar la aleatoriedad de la ruleta.

- Selección por ruleta: Se genera una ruleta, como la que puede verse en la figura 5.2, en la que el peso de cada partícula viene dado por su probabilidad, de modo que las partículas con mayor probabilidad tendrán una porción más grande de la ruleta y por tanto tendrán más posibilidades de ser seleccionadas.



(a)

Figura 5.2: Selección por ruleta.

Para obtener una nueva partícula se genera un número aleatorio que permite seleccionar una partícula de forma aleatoria dentro de la ruleta, y se guarda en la siguiente iteración aplicando un ruido térmico a su posición para evitar tener dos partículas en el mismo lugar.

Al realizar lo anterior para las N partículas que buscamos, obtendremos una nueva generación de partículas en las que previsiblemente las partículas converjan hacia la posición en la que se encuentra el robot.

La localización final en (X, Y, θ) para la iteración en la que nos encontremos, vendrá dada por la media de las partículas elitistas de esa iteración, permitiendo así que la localización varíe de una iteración a otra de forma suave y evitando los saltos bruscos.

5.1.3. Fiabilidad de la localización

Como ya vimos en la sección 4.3.3, el algoritmo de localización debe proporcionar un parámetro que indica la fiabilidad de la localización al resto de componentes de BICA. En el caso del algoritmo de Monte Carlo, esta fiabilidad se ha calculado a partir de la calidad de la observación instantánea que ya hemos descrito, y en la probabilidad media de las partículas elitistas.

Esta fiabilidad se calcula de forma progresiva a lo largo de las iteraciones del algoritmo, por lo que tras diversas pruebas se ha conseguido generar una dinámica heurística que permite aumentar y disminuir la fiabilidad suavemente y soportar malas observaciones sin que afecte demasiado a la fiabilidad total.

Así, el aumento o disminución máximo de la fiabilidad en una sola iteración se ha establecido en 0.15, con lo que teniendo en cuenta los dos valores utilizados para el cálculo de la fiabilidad, se han diferenciado 5 casos posibles:

- Probabilidad alta (> 0.8) y calidad alta (> 0.6): La fiabilidad aumentará en un máximo de 0.15, como se puede ver en la ecuación 5.4:

$$fiabilidad_t = fiabilidad_{t-1} + 0{,}15 * \frac{0{,}2 - (1{,}0 - prob)}{0{,}2} \tag{5.4}$$

- Probabilidad alta (> 0.8) y calidad baja (entre 0.6 y 0.3): La fiabilidad aumentará en un máximo de 0.03, como se puede ver en la ecuación 5.5:

$$fiabilidad_t = fiabilidad_{t-1} + 0{,}03 * \frac{0{,}2 - (1{,}0 - prob)}{0{,}2} \tag{5.5}$$

- Probabilidad baja (< 0.7) y calidad alta (> 0.6): La fiabilidad disminuirá en un máximo de 0.15, como se puede ver en la ecuación 5.6:

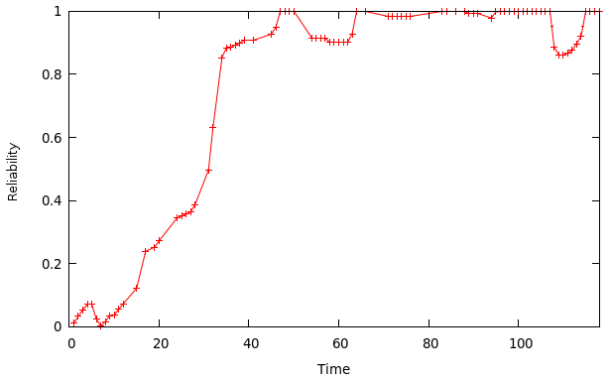
$$fiabilidad_t = fiabilidad_{t-1} - 0{,}15 * \frac{0{,}7 - prob}{0{,}7} \tag{5.6}$$

- Probabilidad baja (< 0.7) y calidad baja (entre 0.6 y 0.3): La fiabilidad disminuirá en un máximo de 0.03, como se puede ver en la ecuación 5.7:

$$fiabilidad_t = fiabilidad_{t-1} - 0{,}03 * \frac{0{,}7 - prob}{0{,}7} \tag{5.7}$$

- Cualquier otro caso: La fiabilidad no cambia respecto al valor anterior.

Con este cálculo, hemos conseguido obtener una fiabilidad que cambie de forma progresiva a lo largo del tiempo en función de las imágenes observadas y la probabilidad de la localización. En la figura 5.3 se puede ver un ejemplo del cambio en la fiabilidad a lo largo del tiempo.



(a)

Figura 5.3: Fiabilidad calculada a lo largo del tiempo.

5.1.4. Experimentos

Para validar el algoritmo desarrollado hemos realizado numerosas pruebas tanto en los simuladores como en el robot real. Dentro de los simuladores, para obtener la precisión del algoritmo se han utilizado los valores de posición real del simulador Webots, ya que es el que más se asemeja a la realidad.

Tanto en Webots como en Gazebo, contamos con sistemas para conocer en todo momento la posición real del robot, haciendo que la comparativa entre la posición real y la calculada sea muy precisa en cada instante. Por el contrario, al realizar las pruebas en el robot real, esta comparativa no es posible, al no contar con un sistema que nos permitiese conocer la posición real del robot, por lo que los valores de precisión se han obtenido calculando el error sólo en puntos concretos.

Los valores configurables en el algoritmo son solamente el número de partículas utilizadas y el porcentaje de partículas elitistas en cada iteración. Tras probar diversas soluciones, hemos decidido utilizar 200 partículas y que un 10 por ciento de ellas sean elitistas, ya que son los valores que mejor resultado nos han dado en la relación coste computacional y precisión.

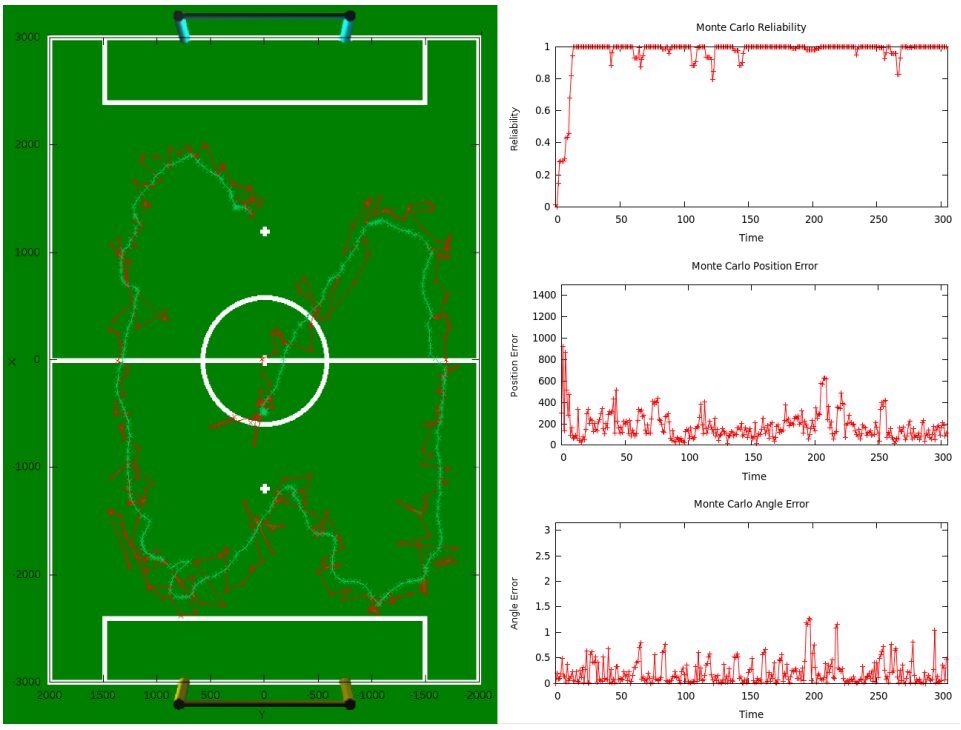
Con estos valores, el coste computacional medio del algoritmo en el robot real es de 27.7 ms, siendo el tiempo siempre bastante estable, no sobrepasando nunca los 40 ms.

A continuación, vamos a mostrar algunos de los experimentos realizados para comprobar las ventajas e inconvenientes del algoritmo.

Error partiendo de posición conocida en simulación

El siguiente experimento está realizado en el simulador Webots, y muestra 4 gráficas distintas: a la izquierda, puede verse una representación del campo en la que se muestra el movimiento real del robot (en verde) y el calculado por el algoritmo (en rojo), mientras que a la derecha vemos las gráficas de la fiabilidad, del error en posición (X, Y) y del error en orientación (θ), en función del tiempo.

El experimento consiste en iniciar el algoritmo hasta que el robot se localice y, una vez localizado, se mueve el robot por todo el campo y se comprueba cómo el algoritmo es capaz de seguir la trayectoria correcta del robot (figura 5.4).



(a)

Figura 5.4: Movimiento con algoritmo de Monte Carlo en Webots.

Como puede verse, el algoritmo mantiene la estimación de posición a medida que el robot se mueve sin perderlo, siendo el error medio en posición de 173.88 mm y en ángulo de

12.03 grados, valores suficientemente buenos como para montar comportamientos basados en posición apoyándose en esto.

Destaca el hecho de que el cambio en la localización entre una medición y otra (que están realizadas cada segundo) es muy brusco, oscilando siempre sobre la trayectoria real.

Robustez ante secuestros en simulación

El segundo experimento ha consistido en comprobar la robustez del algoritmo frente a secuestros. El experimento consiste en situar al robot en un lugar del campo, esperar unos segundos a que se localice, y después trasladarlo a otra parte sin que su odometría lo perciba para ver la evolución en la estimación calculada.

Con esto, comprobamos la capacidad que tiene el algoritmo para recuperarse ante situaciones similares a las que suceden en la competición real, cuando un robot es penalizado o el árbitro le mueve.

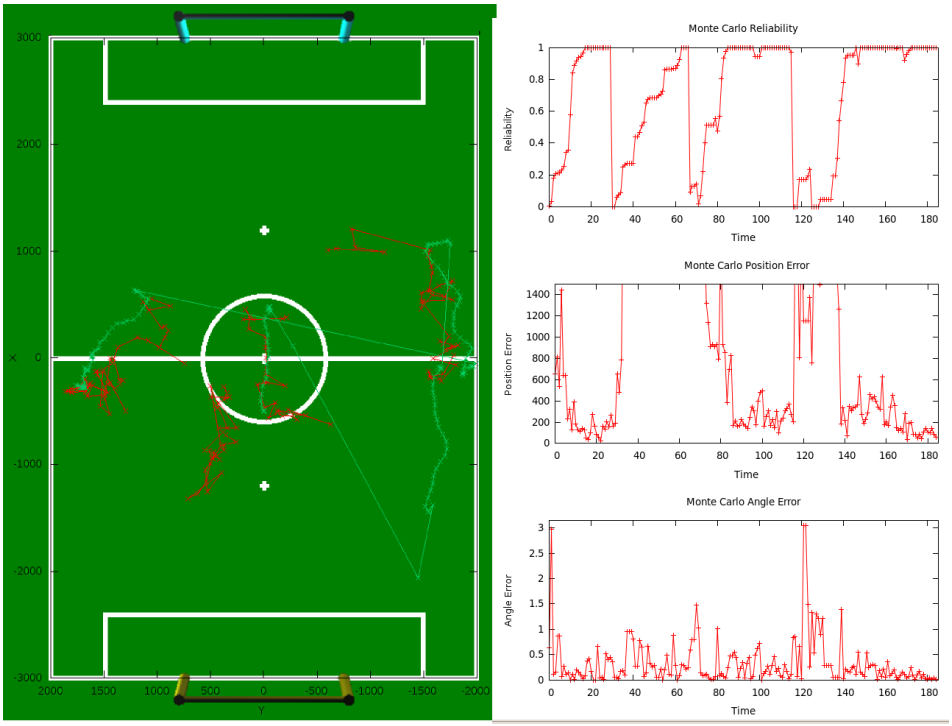
Como se puede ver en la figura 5.5, se han realizado cuatro secuestros distintos, obteniendo en 3 de ellos una relocalización similar a la real, con un error medio en posición de 222.13 mm y en ángulo de 11.45 grados.

Sin embargo, en el segundo secuestro la localización seleccionada tiene un error muy grande. Esto se produce debido a las simetrías con las que cuenta el campo.

Si nos fijamos, la información que obtendríamos desde el modelo de observación no sería muy distinta entre la posición real en la que se encontraba el robot y la que ha seleccionado de forma incorrecta. En ambas posiciones tendríamos una serie de puntos blancos a aproximadamente un metro de distancia (correspondientes a la línea central y al círculo central) y a aproximadamente 4 metros, una serie de puntos azules correspondientes a uno de los postes de la portería. Si no recibimos información que excluya alguna de las dos soluciones, nos encontraríamos ante una simetría en el campo.

Debido a la naturaleza del algoritmo, el método de Monte Carlo sólo puede trabajar con una única solución de modo permanente, lo que lleva a que en el caso de simetrías debe seleccionar una de las posibles soluciones, que puede no ser la correcta, como en este caso.

También es destacable en este caso, el hecho de cómo cambia la fiabilidad de la localización cuando se producen secuestros. En el momento del secuestro, la fiabilidad pasa a valer cero, debido a que el componente FSR habrá informado al algoritmo de que el robot ya no está tocando el suelo, y por lo tanto le han secuestrado o se ha caído.



(a)

Figura 5.5: Secuestros con algoritmo de Monte Carlo en Webots.

El algoritmo en ese momento se reinicia, volviendo a lanzar todas las partículas, y vuelve a aumentar la fiabilidad de forma progresiva según se va mejorando en la localización. También sorprende el hecho de que en el segundo secuestro (el del error por las simetrías que hemos visto), la fiabilidad ha subido mucho más lento que en el resto de casos, debido a que la probabilidad de las partículas no ha sido tan alta.

Por último, el tiempo medio de recuperación entre el secuestro y la selección de una nueva localización fiable (hasta llegar a una fiabilidad mayor de 0.6) ha sido de 16.5 segundos.

Error partiendo de posición conocida en el robot real

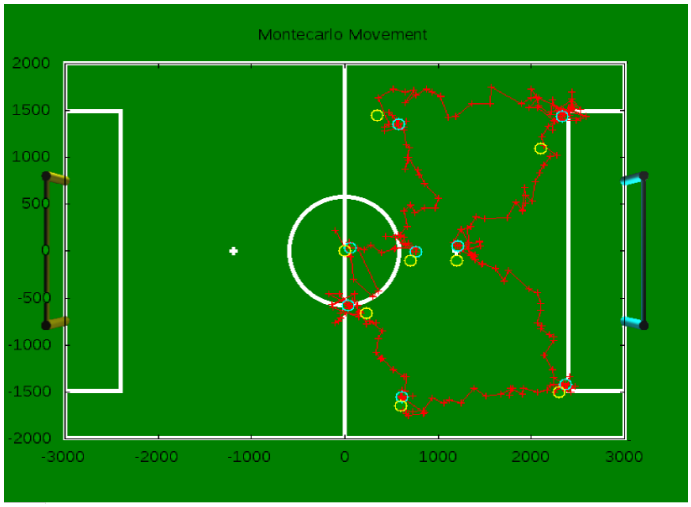
A la hora de realizar las pruebas en el robot real, hay que tener en cuenta dos factores que hacen de él un escenario más exigente que el simulador:

- Falsos positivos: Pueden existir falsos positivos al analizar las imágenes, ya que a pesar de que se han intentado minimizar con los filtros que vimos en el capítulo 4, siempre pueden encontrarse nuevas situaciones no contempladas o directamente ruido en la imagen obtenida del robot, como ya mostramos en la sección 4.4.4.

- Errores odométricos: Como dijimos, los errores odométricos en robots con patas suelen ser muy altos. En los simuladores estos errores pueden minimizarse ya que suelen tratarse de errores sistemáticos que pueden corregirse. Sin embargo, en el robot real estos errores son más impredecibles, por lo que siempre pueden tener error.

Como consecuencia de estos dos aspectos, los resultados en el robot real son mucho más propensos a errores.

En el robot real hemos intentado recrear los experimentos realizados en el simulador, aunque como ya hemos dicho, no podemos medir el error en todo momento, así que sólo tenemos en cuenta algunos puntos de referencia. El primer experimento es similar al que hemos visto en 5.4, calculando el error que se produce al mover el robot por el campo. En este caso, en la imagen 5.6 mostramos en rojo la trayectoria calculada por el robot, en azul los puntos seleccionados de la trayectoria y en amarillo la posición real del robot.



(a)

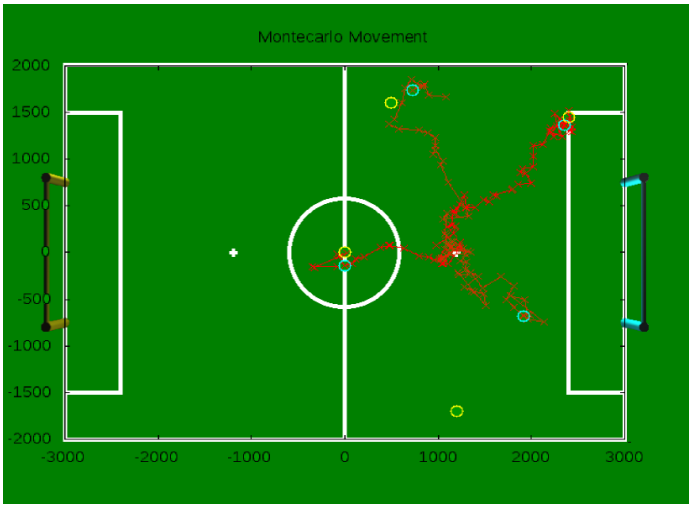
Figura 5.6: Movimiento con algoritmo de Monte Carlo en el robot real.

Al igual que en el simulador, la trayectoria seguida por el robot sufre cambios bruscos, siendo el error medio en los 8 puntos seleccionados de 176.22 mm.

Robustez ante secuestros en el robot real

También hemos realizado en el robot real experimentos con secuestros, como el que podemos ver en la figura 5.7. En este experimento se ha secuestrado al robot en varias ocasiones haciendo que fuese al punto de penalty una vez se localizase. De los 4 secuestros

realizados, en 3 de ellos el robot se ha recuperado como acierto. Sin embargo, en uno de ellos ha seleccionado otra posición del campo muy alejada de la realidad debido a las simetrías.



(a)

Figura 5.7: Secuestros con algoritmo de Monte Carlo en el robot real.

El error medio de las 3 posiciones bien localizadas es de 171.15 mm, mientras que si tenemos en cuenta también la localización errónea, el error medio producido es de 440.13 mm.

En general, el algoritmo cuenta con una gran precisión, estando por debajo de la precisión mínima que nos fijamos como objetivo en el capítulo 2. Por otra parte, el punto débil del algoritmo son las simetrías, que hacen que en ocasiones la localización seleccionada esté muy alejada de la posición real.

El tiempo de ejecución en el robot real también está dentro de los límites que nos fijamos, siendo el tiempo medio de ejecución de 27.7 ms, destacando además el hecho de que el tiempo es muy estable en todas las iteraciones, siendo siempre cercano al tiempo medio.

5.2. Localización por algoritmo evolutivo

Como ya hemos visto en los experimentos anteriores, el método de Monte Carlo puede tener un mal funcionamiento en el caso de que las imágenes observadas contengan elementos simétricos en otras partes del campo. Esto se podría subsanar si tuviésemos un método capaz de soportar varias hipótesis de localización simultáneamente. Con este objetivo, hemos diseñado y desarrollado un algoritmo evolutivo capaz de soportar varias localizaciones y que describiremos a continuación.

Los algoritmos evolutivos son métodos de optimización metaheurísticos que se inspiran en la evolución biológica para buscar la solución a un problema. Las soluciones candidatas son conocidas como individuos, de forma similar a las partículas en el algoritmo de Monte Carlo. Estos individuos son parte de una población que evoluciona a lo largo del tiempo utilizando para ello operadores genéticos, como por ejemplo la mutación o el cruce, que explicaremos más adelante.

Además, cada individuo es evaluado a partir de una función de salud que nos permite conocer su "vida", es decir, conocer en qué medida se acerca a la solución óptima, similar a la probabilidad de cada partícula que vimos en Monte Carlo.

En nuestra implementación, el algoritmo evolutivo cuenta con 4 elementos básicos:

- Individuos: Los individuos son análogos a las partículas en el algoritmo de Monte Carlo. En este caso, guardan la posición del robot (X, Y, θ), la probabilidad obtenida de la función salud, y un parámetro para indicar si es elitista, al igual que sucedía con las partículas.

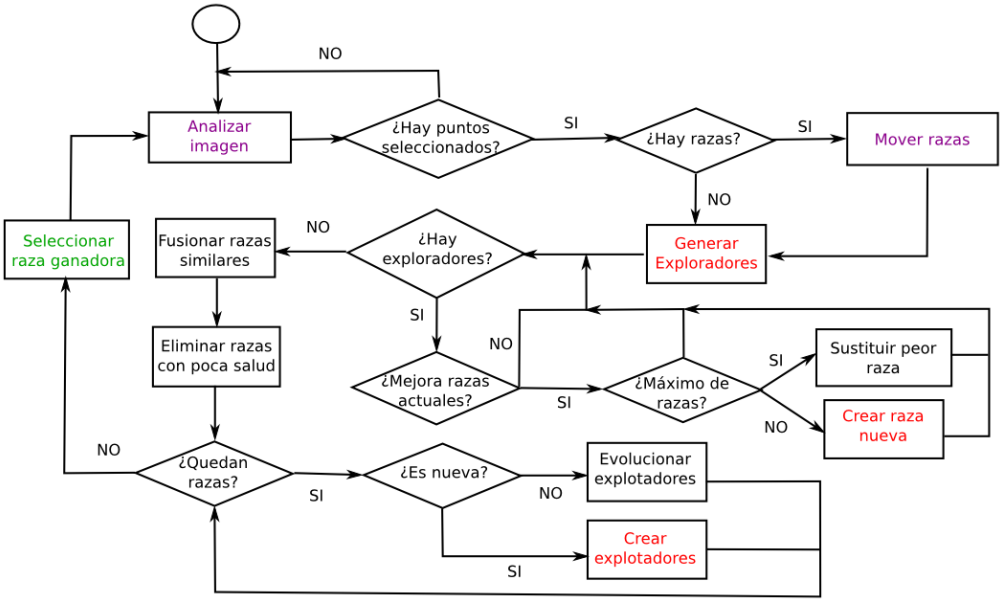
- Razas: Cada raza es una población de individuos que representan una posible solución a la localización del robot. En el algoritmo existen varias razas que compiten entre sí para ser la ganadora en una iteración determinada, donde la raza ganadora es considerada la posición actual del robot.

  Los parámetros almacenados en la raza son la posición del robot (X, Y, θ), la salud de la raza, el número de victorias y su vida (número de iteraciones que deben transcurrir antes de poder ser eliminada).

- Exploradores: Individuos independientes encargados de buscar posiciones en las que generar nuevas razas.

- Explotadores: Cada uno de los individuos que forman parte de una raza, encargados de analizar en profundidad una posible posición en busca de la mejor solución.

En cada iteración del algoritmo se realizan una serie de pasos para conocer la localización del robot en cada momento (figura 5.8).



(a)

Figura 5.8: Esquema del algoritmo evolutivo.

A continuación vamos a describir algunos de los pasos más importantes realizados en cada iteración:

- Cálculo de salud: Analizamos la imagen y obtenemos la información de posición como vimos en el capítulo 4.
- Creación de exploradores: Creamos y dejamos evolucionar nuevos exploradores para buscar posiciones dentro del campo en las que crear nuevas razas. Este proceso lo veremos con detalle en la sección 5.2.1.
- Gestión de razas: Creamos, fusionamos o eliminamos razas en función del estado de éstas y de la salud de los exploradores creados, como veremos en 5.2.2.
- Evolución de razas: Se crean los explotadores de la raza o, en caso de ya existir, se evolucionan mediante operadores genéticos, como explicaremos en 5.2.3.

  En el caso de que el robot se mueva, también aplicamos un operador de movimiento a cada una de las razas al inicio de cada iteración. Este operador de movimiento es el mismo que vimos en la sección 5.1.1.

- Selección de la raza ganadora: Después de calcular la salud de cada raza, seleccionamos entre ellas la ganadora, que determinará la posición en la que se encuentra el robot, lo que veremos en 5.2.4.

Como ya hemos visto, tanto el análisis de observaciones como el operador de movimiento son iguales a los explicados en el algoritmo de Monte Carlo. Por ello vamos a explicar de forma detallada en las siguientes secciones el resto de pasos del algoritmo.

5.2.1. Creación de exploradores

Los exploradores son individuos que no pertenecen a ninguna raza y que intentan buscar posiciones en las que la salud del individuo es mejor que la salud de las razas que ya tenemos.

En un principio, estos exploradores eran distribuidos de forma aleatoria en cada iteración. Sin embargo, tras realizar varias pruebas, nos dimos cuenta de que dado el tamaño del campo de fútbol de la RoboCup, era mejor inicializar siempre los exploradores en posiciones determinadas que barran casi todo el campo y realizar búsquedas locales que nos permitan alcanzar la mejor posición.

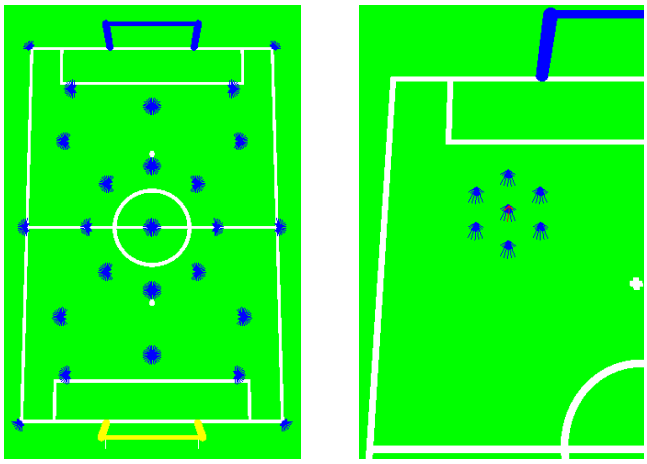
Por ello, la creación de nuevas razas a partir de exploradores se realiza en 4 iteraciones distintas, con el fin de repartir el coste computacional y evitar iteraciones puntuales muy costosas.

En las dos primeras iteraciones se calcula la salud de las posiciones fijadas, guardándolas ordenadamente en una lista enlazada según la salud obtenida. En total se evalúan 490 posibles posiciones y ángulos, en las posiciones que podemos ver en 5.9(a).

En la tercera y cuarta iteración, seleccionamos las 6 mejores posiciones guardadas en la lista enlazada y las hacemos evolucionar como puede verse en la figura 5.9(b). A partir del explorador seleccionado (en rojo), se generan 4 nuevos exploradores en la misma posición pero con una pequeña variación en el ángulo, y otros 30 exploradores a su alrededor en un círculo de 30 cm de radio.

Al evolucionar los 6 primeros exploradores, calculamos la salud en total de 210 exploradores, que son almacenados en orden en la lista al igual que en las dos primeras iteraciones. Al terminar las 4 iteraciones, seleccionaremos los 6 mejores exploradores, que serán candidatos para generar una nueva raza, como se verá en la sección 5.2.2.

Una vez implementada esta solución, hemos comprobado cómo al calcular nuevos exploradores con observaciones con poca información (por ejemplo, donde sólo se ven



(a) (b)

Figura 5.9: Posiciones fijas de los exploradores (a) y evolución de éstos (b).

líneas), los candidatos a raza generados no son demasiado buenos. Por ello, y con el objetivo de disminuir el tiempo de ejecución del algoritmo, que es uno de los puntos más importantes a tener en cuenta en la ejecución en el robot real, hemos decidido realizar la creación de exploradores únicamente cuando veamos alguna de las porterías.

5.2.2. Gestión de razas

Al disponer de varias razas, tenemos que desarrollar un mecanismo capaz de gestionar su creación, eliminación o fusión. Antes de detallar estos mecanismos vamos a explicar dos de los parámetros de los que dispone cada raza, el número de victorias y la vida:

- Número de victorias: Se considera que una raza ha tenido una victoria en una iteración cuando su probabilidad es mayor que la del resto de razas. En un principio todas las razas recién creadas comienzan con 0 victorias, y este valor se modifica en función de cómo se produzcan las victorias, como se verá en la sección 5.2.4. Este parámetro servirá finalmente para seleccionar la raza que establecerá la localización del robot.

- Vida: Este parámetro se ha creado con dos objetivos, el primer objetivo consiste en dar prioridad a las razas ya creadas frente a nuevos candidatos, evitando así la generación continua de razas que se crean y mueren en la siguiente iteración. Para ello, al crearse una raza se le establece una vida mínima de 3 iteraciones, en las cuales no puede ser borrada ni sustituida (aunque sí fusionada).

El segundo objetivo consiste en evitar que una raza pueda ser eliminada por una mala observación que haga que su salud sea baja, cuando previsiblemente era un buen candidato a tener en cuenta para la solución final. Esto se consigue haciendo que la vida mínima de una raza sea de 9 iteraciones cuando ha sido la de mayor probabilidad en una iteración.

De esta forma se pretende que las razas tengan cierta estabilidad y que no se estén creando y borrando de forma continua.

Creación de nuevas razas

Cuando dispongamos de nuevos candidatos a ser raza, a través de la creación de exploradores, debemos decidir si creamos nuevas razas o si sustituimos las ya existentes. El algoritmo cuenta con un máximo de razas a crear, que actualmente está fijado en 4, que evita el aumento exponencial en el tiempo de ejecución con el número de razas.

El primer paso a realizar es el de comprobar si pueden crearse nuevas razas. Para ello debemos comprobar que no se ha alcanzado el número máximo de razas, y que alguna de las razas tenga tanto su vida como su número de victorias a cero. En caso de que esto no sea así, los exploradores no se tendrán en cuenta.

Si se pueden crear nuevas razas, el siguiente paso es el de comprobar la novedad de los nuevos candidatos, es decir, para cada candidato debemos ver si alguna de las razas ya existentes está situada en la misma posición (X, Y, θ) que él. De ser así, no se intentará crear una nueva raza, sino que se supondrá que la raza existente ya es la representativa de ese candidato, y se aumentará su vida.

Por ultimo, si el candidato supone una novedad se creará una nueva raza en dos casos: cuando no se haya llegado aún al máximo de razas permitidas, o cuando su salud sea mayor que la de alguna de las razas ya existentes que puedan ser eliminadas.

Fusión de razas

En el caso de que dos razas evolucionen hacia dos localizaciones similares, tanto en posición como en ángulo, se considera que ambas razas llevan hacia la misma solución final y se produce una fusión entre ellas. Esta fusión consiste en eliminar la raza que menos victorias tenga, y en caso de empate, se eliminará la que tenga una salud más baja.

Eliminación de razas

En el caso de que alguna de las razas tenga tanto su número de victorias como su vida a 0, se debe comprobar si su salud es demasiado baja, lo que haría que la eliminásemos. Este valor mínimo se ha establecido en 0.6, ya que este valor es lo suficientemente bajo como para considerar que la localización de la raza no es la correcta.

5.2.3. Evolución de razas

Cuando se crea una raza desde un explorador, todos los explotadores de la raza son creados de forma aleatoria mediante un ruido térmico que se aplica sobre el explorador que creó la raza. A partir de ese momento, en el resto de iteraciones los explotadores de la raza evolucionan utilizando tres operadores genéticos:

- Elitismo: Como ya vimos en el método de Monte Carlo, el elitismo consiste en seleccionar los N mejores explotadores de cada raza, que se clasificarán por su salud. Los explotadores elitistas se guardarán sin modificación para la siguiente población.
- Cruce: Se seleccionan al azar dos explotadores de la raza y se cruzan, haciendo la media entre ellos en (X, Y, θ).
- Mutación: A partir del valor del explotador, se aplica un ruido térmico aleatorio, que modifica su posición y su ángulo.

Una vez evolucionados los explotadores, se calcula el valor final de la raza como la media de todos sus elitistas, haciendo que no se produzcan cambios bruscos en la localización.

Además, como ya dijimos, en el caso de que el robot se haya desplazado desde la última iteración se aplica un operador de movimiento al inicio de la iteración tanto a las razas como a sus explotadores, que se corresponde con el modelo de movimiento que vimos en Monte Carlo (5.1.1).

5.2.4. Selección de la raza ganadora

Tras evaluar todas las razas existentes debemos seleccionar una de ellas para que sea la localización final del robot. La raza seleccionada será la que cuente con mayor número de victorias tras completar la iteración, por lo que en cada iteración debemos aumentar o disminuir este valor en cada una de las razas.

El primer paso a realizar es seleccionar la raza con mayor salud en esta iteración, para aumentar su vida como ya comentamos en 5.2.2. En el caso de que la raza seleccionada ya fuese la raza ganadora en la anterior iteración, se aumentará el número de victorias de ésta y se disminuirá el del resto. Sin embargo, cuando la raza con mayor salud sea distinta de la última ganadora, sólo cambiaremos los valores del número de victorias cuando la diferencia entre su salud y la salud de la última ganadora sea suficientemente grande.

Esto es así puesto que nos interesa que sea difícil cambiar la raza ganadora cuando la fiabilidad de la localización actual es muy alta o cuando la imagen observada no es muy buena (no se vea ninguna portería). De esta forma, intentamos favorecer a las razas que ganen durante mucho tiempo y sólo cambiamos cuando la diferencia es muy grande (lo que indicaría que la localización actual no es correcta).

En el caso de que la diferencia sea suficientemente grande se procede igual que antes, a la raza ganadora se le sumaría una victoria y al resto de razas se les restaría.

Así, la localización final que devolverá el algoritmo en esta iteración será la que más victorias tenga tras los cálculos que acabamos de comentar.

5.2.5. Fiabilidad de la localización

Al igual que en el algoritmo de Monte Carlo, es necesario conocer la fiabilidad de la localización en todo momento para que lo conozcan el resto de componentes de BICA.

Los cálculos realizados para obtener la fiabilidad son los mismos que los que vimos en el capítulo 5.1.3, teniendo en cuenta la fiabilidad de la observación actual y la salud de la raza ganadora.

Sin embargo, a este cálculo se ha añadido una pequeña variación para tener en cuenta los saltos entre razas, algo que no era necesario en el anterior algoritmo. Tras calcular la fiabilidad como ya vimos en Monte Carlo, se comprueba si se ha producido un cambio en la raza ganadora respecto a la última iteración.

En ese caso, se calcula la distancia en (X, Y) entre la nueva raza ganadora y la última ganadora con la ecuación 5.8. Si la distancia es mayor de 2 metros, la fiabilidad se pone a 0, y en caso de ser menor, se actualiza su valor teniendo en cuenta esta distancia como se ve en la ecuación 5.9

$$dist = \sqrt{(New_X - Last_X)^2 + (New_Y - Last_Y)^2} \tag{5.8}$$

$$fiabilidad_t = fiabilidad_t * (1 - \frac{dist}{2000}) \tag{5.9}$$

Esto hace que la fiabilidad disminuya cuando se produce un cambio de raza, haciendo que sea cero cuando la distancia entre ellas es muy grande, como si se hubiese reiniciado el algoritmo, pero manteniendo la fiabilidad si la distancia es muy pequeña, ya que entendemos que este salto se produce para ganar en precisión.

5.2.6. Experimentos

Al igual que con el algoritmo de Monte Carlo, se han realizado numerosas pruebas tanto en el simulador como en el robot real.

El algoritmo evolutivo cuenta con más parámetros configurables que el método de Monte Carlo. Estos parámetros son: el número máximo de razas, los explotadores en cada raza, el porcentaje de elitistas en cada raza y el porcentaje de individuos que evolucionan mediante cruce.

Para cada uno de estos valores se han realizado distintas pruebas y hemos seleccionado los valores que mejor se comportaban y tenían unos tiempos de ejecución dentro de unos márgenes. Así, el número de razas se ha establecido en 4, el número de explotadores en 30 para cada raza, el porcentaje de elitistas en el 20 %, y el porcentaje de cruce también en el 20 %.

El número máximo de razas ha sido un factor fundamental, puesto que a mayor número de razas, mejores serían los resultados en cuanto a precisión espacial pero aumentaría el tiempo de ejecución de forma lineal.

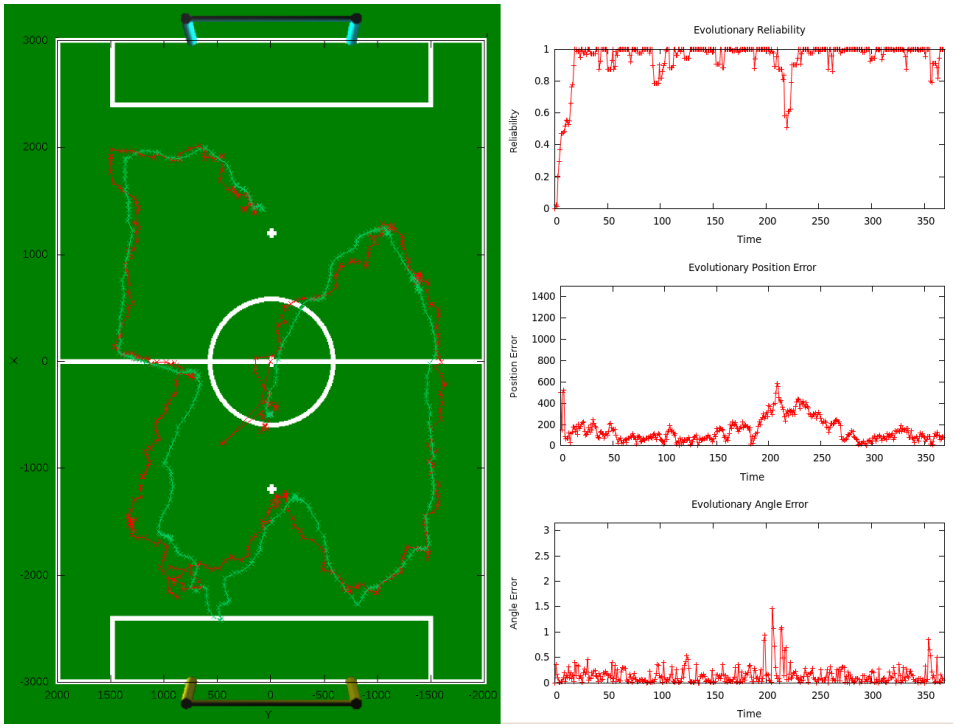
Vimos experimentalmente como con 2 razas el algoritmo es muy rápido, pero había casos en los que no se creaban razas en los lugares correctos debido a las características del campo (por ejemplo, en el caso de las esquinas del campo, existen 4 zonas simétricas). También comprobamos como con 6 razas se contemplaban la mayoría de los casos, pero su tiempo de cómputo era demasiado alto. Debido a esto, decidimos quedarnos en un término medio en el que la eficiencia y la precisión estuviesen compensadas, con 4 razas.

Con estos valores, el coste computacional medio del algoritmo en el robot real ha sido de 37.6 milisegundos por iteración, aunque este tiempo cambia en función del número de razas y de si lanzamos exploradores en esa iteración o no, pudiendo ir desde 10 ms hasta 80 ms.

A continuación mostramos algunos de los experimentos realizados.

Error partiendo de posición conocida en simulación

Al igual que con el algoritmo anterior, los primeros experimentos que vamos a mostrar son los realizados en el simulador Webots. En la figura 5.10 podemos ver una ejecución típica del primer experimento, que consiste en situar al robot en el centro del campo, esperar a que se localice y, una vez localizado, mover al robot por el campo para ver la evolución de la localización.



(a)

Figura 5.10: Movimiento con algoritmo evolutivo en Webots.

Al igual que sucedía en Monte Carlo, el algoritmo no pierde la posición del robot, siendo el error medio en posición de 142.42 mm y en ángulo de 8.42 grados. La precisión conseguida es bastante buena y es suficiente para poder utilizarse en nuestras aplicaciones.

En este caso, el movimiento que se produce en la localización entre cada medición es más suave que en el caso del algoritmo de Monte Carlo, esto se produce debido al funcionamiento de los explotadores de cada raza, que son capaces de acercarse al máximo local de la posición calculada.

También podemos ver, como cuando el robot estaba alejado de la portería ha aumentado el error en la localización, aunque este error se ha corregido cuando nos hemos acercado a la línea del centro del campo, al tener una referencia cercana más fiable. En el caso de

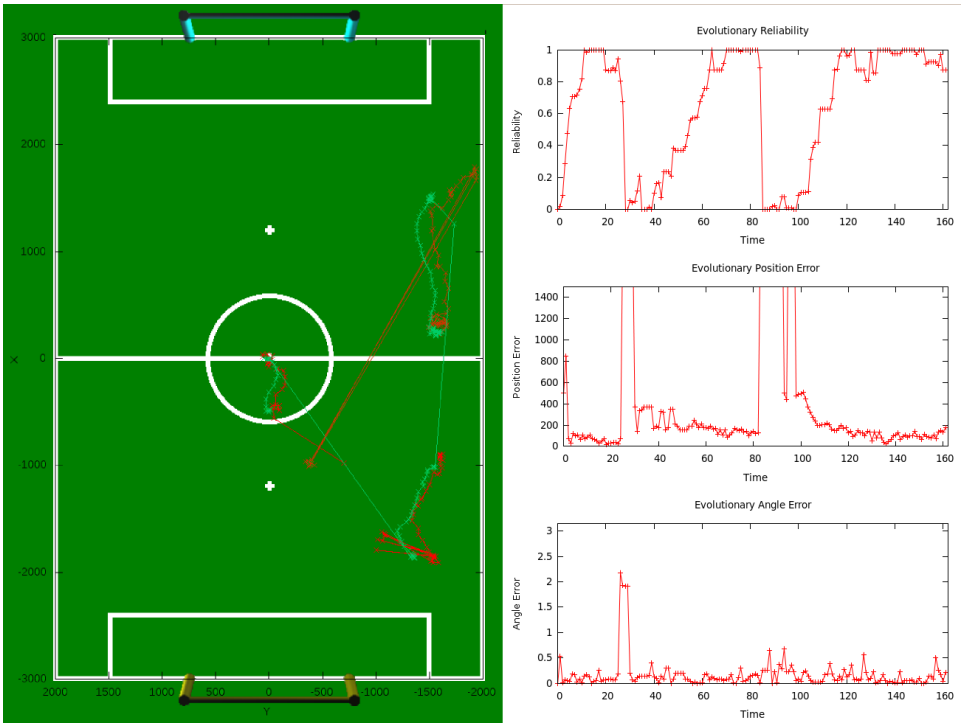
que este error hubiese ido en aumento y el robot se perdiese, hubiese llegado un punto en el que se crearía una raza en la posición correcta y la localización habría cambiado.

Robustez ante secuestros en simulación

Una ejecución representativa del experimento de robustez frente a secuestros es el que se muestra en la figura 5.11. En este caso, los 3 secuestros realizados han sido satisfactorios, no cayendo en posiciones simétricas, siendo el error medio en posición de 148.99 mm y en ángulo de 6.36 grados.

Como puede verse, en los 3 secuestros la posición inicialmente seleccionada no ha sido totalmente correcta, debido a las simetrías. Sin embargo, al contar con múltiples hipótesis, al transcurrir varias iteraciones y contar con más información, la raza seleccionada ha sido finalmente la correcta.

En el caso de haber utilizado un algoritmo que no soportase varias hipótesis, como el de Monte Carlo, es posible que la localización hubiese seleccionado la posición incorrecta, como sucedía en uno de los secuestros que vimos en la figura 5.5.



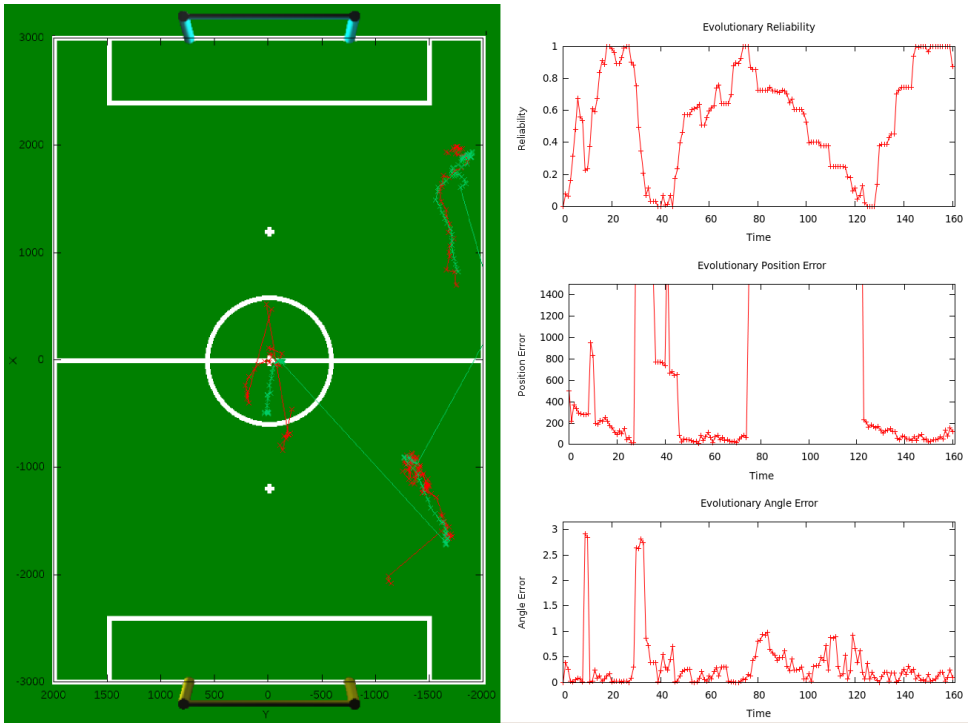
(a)

Figura 5.11: Secuestros con algoritmo evolutivo en Webots.

En este caso, también ha sido fundamental la ayuda de los FSR para realizar una rápida recuperación, llegando de nuevo a una posición fiable (con fiabilidad mayor que 0,6) en 20.67 segundos de media. El tiempo de recuperación es mayor que en el algoritmo anterior, debido a las indecisiones en la selección de la raza al comienzo de cada secuestro, pero a cambio el porcentaje de recuperaciones acertadas es mayor.

Para ver la influencia del uso de los FSR en la recuperación, hemos realizado un nuevo experimento que consiste en secuestrar al robot sin levantarlo del suelo, que puede verse en la figura 5.12, donde se puede comprobar cómo los mecanismos de estabilidad del algoritmo ralentizan su recuperación.

Como puede verse, tras secuestrar al robot, la fiabilidad disminuye poco a poco, por lo que tarda mucho más en cambiarse de raza. Así, el tiempo de reacción medio medido en esta situación ha sido de 30 segundos, casi un 50 % más que utilizando los FSR.



(a)

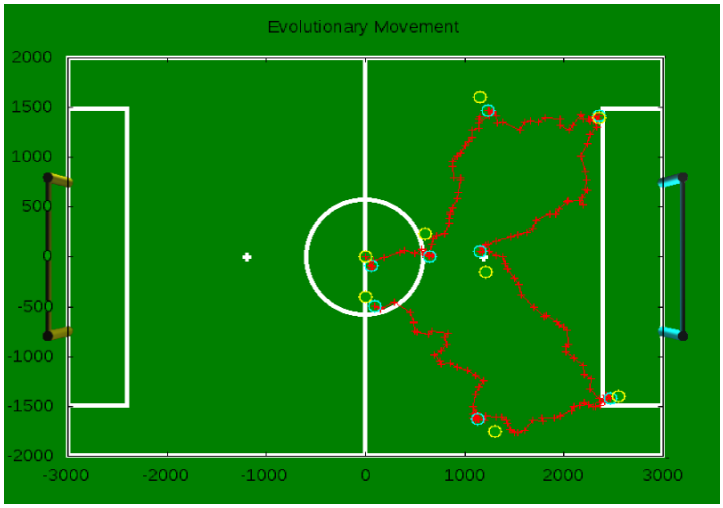
Figura 5.12: Secuestros con algoritmo evolutivo en Webots sin FSR.

Error partiendo de posición conocida en el robot real

Como ya vimos con el algoritmo de Monte Carlo, las pruebas en el robot real suelen tener mayor error debido sobre todo a los errores en la odometría y a los falsos positivos.

De nuevo hemos realizado muchas pruebas en el robot real para comprobar la capacidad del algoritmo para seguir la trayectoria del robot una vez localizado. Como ya dijimos, no es posible conocer el error de manera continua, por lo que se ha medido el error en varios puntos de referencia. Así, en la figura 5.13 vemos una ejecución representativa donde mostramos en rojo la trayectoria calculada por el robot, en azul los puntos seleccionados de la trayectoria, y en amarillo su posición real.

El error medio medido en los 8 puntos seleccionados ha sido de 143.68 mm, muy similar al resultado que obtuvimos en en el simulador.



(a)

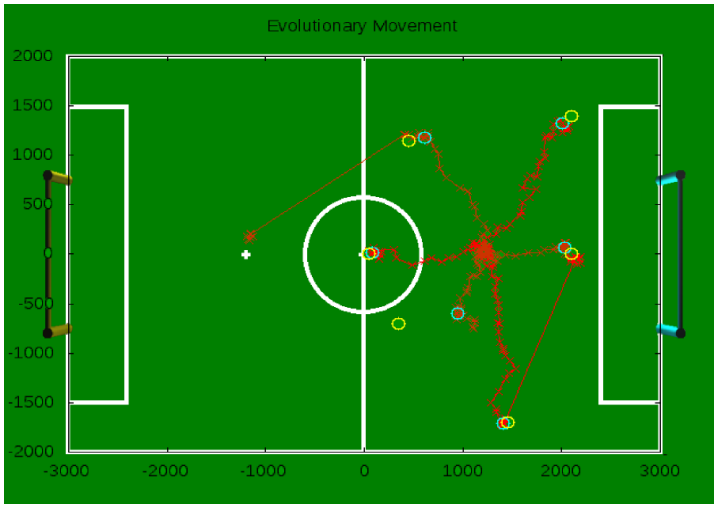
Figura 5.13: Movimiento con algoritmo evolutivo en el robot real.

Robustez ante secuestros en el robot real

En cuanto a los secuestros, podemos ver en la figura 5.14 uno de los experimentos realizados, en el que se secuestraba al robot para que una vez localizado fuese al punto de penalty. En este caso, de 6 secuestros realizados, 5 se han realizado con una precisión muy alta mientras que en uno de ellos, el error ha sido de 600 mm, siendo el error medio de 175.74 mm.

En el caso de producirse errores en la localización tan altos como el que hemos visto, debidos a simetrías o falsos positivos, y que se producen por seleccionarse la raza ganadora

de forma incorrecta, es previsible que de seguir moviéndose el robot, se llegase a un punto donde la simetría no fuese tan clara y se cambiase a la raza correcta.



(a)

Figura 5.14: Secuestros con algoritmo evolutivo en el robot real.

El resultado final del algoritmo ha sido muy satisfactorio, ya que cuenta con una gran precisión, estando por debajo de la precisión mínima que nos pusimos como objetivo, y además es capaz de manejar simetrías, puesto que está especialmente diseñado para ello.

Además, el tiempo medio de ejecución también en bastante bajo, siendo de 37.6 ms, aunque teniendo como punto negativo que es bastante variable, oscilando entre los 10 hasta los 80 ms.

Podremos ver una comparativa de los dos algoritmos que hemos descrito en el próximo capítulo, donde analizaremos los pros y los contras de ambos para finalmente quedarnos con uno de ellos para ser utilizado en el contexto de nuestro trabajo.

5.3. Diseño General

Para desarrollar los algoritmos que acabamos de describir, hemos tenido que adaptar su estructura para que fuese compatible con las dos arquitecturas de desarrollo utilizadas, JdeRobot y BICA. A continuación vamos a detallar de qué forma se comunican nuestros algoritmos con estas plataformas y se mostrarán las interfaces gráficas desarrolladas para cada uno de ellos.

5.3.1. JdeRobot

En JdeRobot hemos diseñado un nuevo esquema que se comunica con los simuladores y el robot real a través de dos drivers, el driver *Gazebo* y el driver *Naobody*.

Para conectarnos con el simulador Gazebo se sigue el diseño que se muestra en la figura 5.15(a). Nuestro esquema toma las imágenes y los valores odométricos del driver *Gazebo*, que se encarga de la comunicación con el simulador, y además permite controlar la navegación del robot por el campo.

Por otra parte, en el caso del simulador Webots o del robot real, es necesario utilizar el driver *Naobody* para obtener los valores de los sensores, mientras que para controlar al robot, utilizamos otro esquema específicamente diseñado para ello llamado *Naooperator* (figura 5.15(b)).

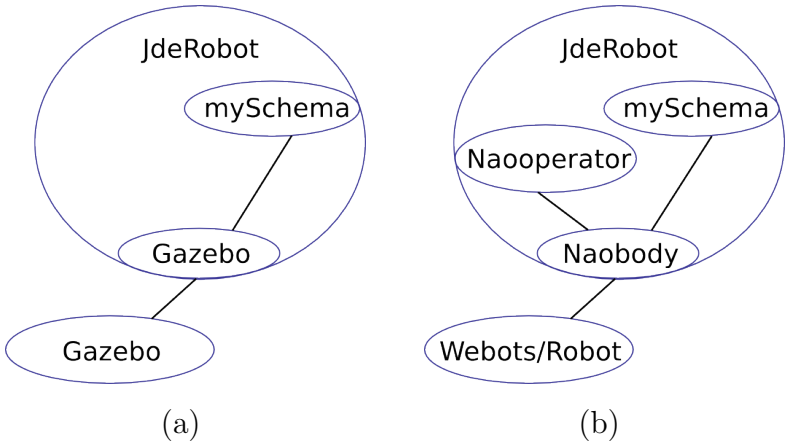


Figura 5.15: Conexión de JdeRobot con los simuladores y el robot real.

Una vez conectados al driver correcto, la comunicación con el robot y los simuladores es totalmente transparente para los algoritmos, sin necesidad de reimplementar partes del código para cada caso.

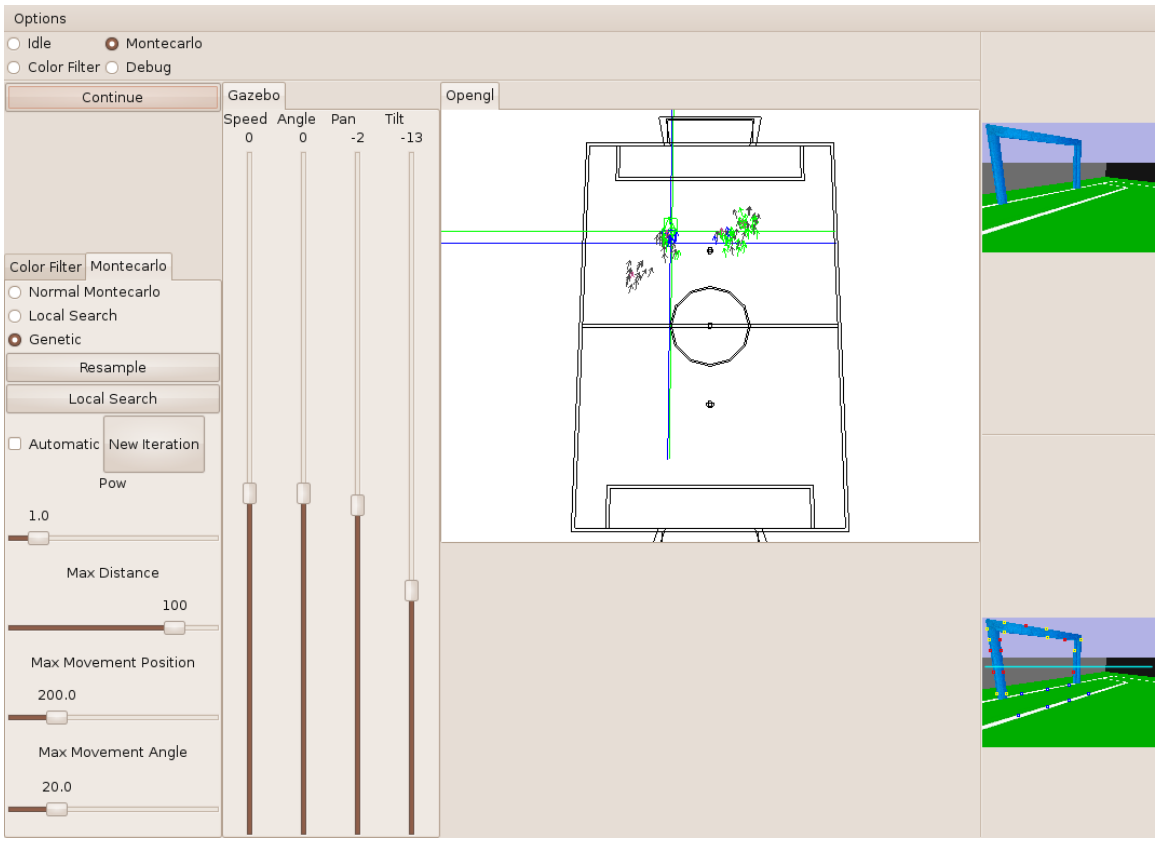
Para depurar los algoritmos en esta plataforma hemos desarrollado la interfaz gráfica en GTK que se muestra en la figura 5.16. La interfaz cuenta con 3 partes diferenciadas:

- En la parte izquierda podemos seleccionar el tipo de algoritmo que queremos utilizar. Así, podemos ejecutar normalmente el algoritmo de Monte Carlo o el evolutivo, o bien podemos seleccionar tareas específicas de depuración, pudiendo además configurar algunos de los parámetros de cada uno.

Además, si utilizamos el simulador Gazebo, manejamos tanto el movimiento del cuerpo del robot como su cuello desde esta interfaz. En el caso de utilizar Webots o el robot real se utilizará el esquema *Naooperator* como ya hemos comentado.

- En la parte central tenemos un cuadro de OpenGL, en el que se visualiza el campo de la RoboCup. En este campo se muestra la localización calculada (cruz azul), la real (cruz verde) y las partículas o individuos que existan en ese momento del algoritmo. Además, este cuadro de OpenGL también ha servido para mostrar información de depuración como la que vimos en la figura 4.17.

- A la derecha, vemos la imagen actual obtenida del robot y el resultado final tras analizar esa observación con los puntos seleccionados.

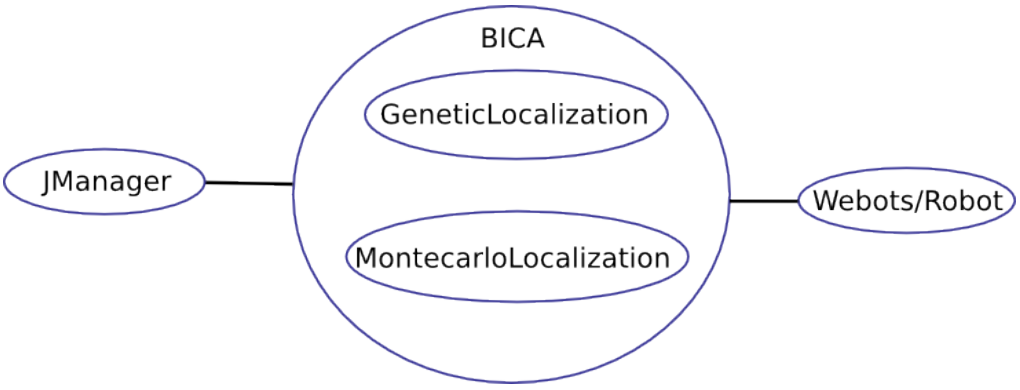


(a)

Figura 5.16: Interfaz gráfica en JdeRobot usando el simulador Gazebo.

5.3.2. BICA

BICA, al ejecutarse como un módulo de Naoqi, puede enviar o recibir datos del simulador o del robot real llamando directamente al resto de módulos de Naoqi. Así, se han creado dos componentes dentro de la arquitectura BICA, llamados *GeneticLocalization* y *MontecarloLocalization* que pueden comunicarse directamente con el simulador Webots o el robot real, como se muestra en la figura 5.17.



(a)

Figura 5.17: Conexión de BICA con los simuladores y el robot real.

Para obtener de forma visual información de estos componentes se utiliza la herramienta JManager, desarrollada en Java, que permite la depuración y el análisis de los datos de forma sencilla, ya que se comunica directamente con los componentes que se ejecutan en el robot. Para ambos componentes hemos generado una única pestaña que se encarga de obtener la información del que se esté ejecutando en ese momento.

En la figura 5.18 podemos ver la pestaña generada, que cuenta con dos elementos principales:

- En la parte izquierda podemos ver el campo de la RoboCup en OpenGL. En este campo se muestra la localización calculada (flecha roja con círculo), la localización real (flecha morada con círculo) y las partículas o individuos que tenga el algoritmo en ese instante.

  El tamaño del círculo de la localización calculada dependerá de la fiabilidad de la localización, haciéndose el círculo más pequeño cuanto mayor sea la fiabilidad.

  Este campo en OpenGL puede manipularse mediante eventos de ratón, pudiendo hacer zoom en una zona concreta o desplazar el campo para verlo desde otra perspectiva.

- A la derecha se muestra la observación actual del robot y el resultado final tras su análisis. Debajo de estas dos imágenes, también podemos ver la fiabilidad de la localización actual de forma numérica.

(a)

Figura 5.18: Interfaz gráfica en JManager usando el simulador Webots.

Capítulo 6

Conclusiones y Trabajos futuros

En los capítulos anteriores hemos explicado los algoritmos diseñados para resolver el problema de la localización de un robot en el campo de la RoboCup. En este capítulo vamos a comparar los dos algoritmos realizados con el fin de obtener conclusiones sobre su posible utilización. Además, repasaremos los objetivos planteados al inicio para saber en qué grado se han satisfecho. Por último, indicaremos cuáles pueden ser los trabajos futuros abiertos en relación con el proyecto realizado.

6.1. Comparativa de algoritmos de localización

En el capítulo 5, vimos los dos algoritmos desarrollados para localizar al robot, el algoritmo de Monte Carlo y el algoritmo evolutivo. Ambos algoritmos han sido altamente probados tanto en el simulador como en el robot real, cumpliendo además con los objetivos planteados. Sin embargo, existen algunas diferencias entre ellos, que vienen resumidas en el cuadro 6.1.

Como hemos visto, el principal problema del algoritmo de Monte Carlo es que no puede manejar las simetrías correctamente, ya que siempre se queda con una única solución, algo que el algoritmo evolutivo es capaz de solventar al contar con múltiples razas.

Por otra parte, el algoritmo evolutivo tiene una complejidad de implementación mayor, puesto que es necesario encontrar los valores adecuados de número de razas, número de explotadores, etc, que hagan que el algoritmo tenga un buen funcionamiento sin penalizar el coste computacional. Por el contrario, con el método de Monte Carlo, implementamos un algoritmo clásico de condensación muy utilizado y no demasiado complejo.

	Monte Carlo	Evolutivo
Multimodal	No	Sí
Tolerancia Simetrías	Mala	Buena
Complejidad	Media	Alta
Tiempo medio	27.7 ms	37.6 ms
Precisión Movimiento	173.88 mm, 12.03 grados	142.42 mm, 8.42 grados
Precisión Secuestros	222.13 mm, 11.45 grados	148.99 mm, 6.36 grados
Tiempo Recuperación	16.5 segundos	20.67 segundos

Cuadro 6.1: Comparativa entre algoritmos de localización.

El tiempo de ejecución en el robot real es menor en el caso del algoritmo de Monte Carlo, aunque ambos tiempos son bastante bajos, por lo que la media no es muy relevante. Sí es importante el hecho de que el algoritmo de Monte Carlo tenga un tiempo de ejecución bastante estable, al utilizar siempre el mismo número de partículas, mientras que el evolutivo cambia su tiempo en función de la situación, pudiendo llegar hasta los 80 ms.

Para la precisión del algoritmo se han utilizado los valores de Webots, ya que son más fiables que las medidas en puntos concretos en el robot real, aunque como hemos visto los errores eran similares. Ambos algoritmos cuentan con una gran precisión, estando ambos en torno a los 15-20 cm de error en posición y de unos 10 grados en ángulo. Sin embargo, el algoritmo evolutivo es ligeramente superior, gracias a el papel que juegan los explotadores de cada raza.

El tiempo de recuperación ante secuestros es mejor en el algoritmo de Monte Carlo, debido a que converge rápidamente a un solo punto, mientras que el algoritmo evolutivo puede estar cambiando de raza durante unos segundos hasta que concreta su posición.

Teniendo en cuenta estos factores y el contexto en el que se van a utilizar los algoritmos, consideramos clave el comportamiento ante las simetrías, puesto que pueden llevar a tener una localización con gran error. Por ello, estimamos que el algoritmo que mejor se adecúa al problema que queremos solucionar es el algoritmo evolutivo.

6.2. Conclusiones

Tras un desarrollo de cerca de 4900 líneas de código para la arquitectura JdeRobot y de 8800 para BICA, se han alcanzado los objetivos planteados en el capítulo 2, cuyo objetivo principal era el de conseguir autolocalizar al robot Nao en el campo de la RoboCup utilizando la visión artificial.

A continuación vamos a repasar los distintos subobjetivos que se plantearon para conocer la solución adoptada en cada uno de ellos:

1. El primero de los subobjetivos a alcanzar era la detección de las porterías y las líneas del campo en la imagen, que se ha resuelto mediante los pasos y filtros que estudiamos en el capítulo 4. Como ya vimos, una vez detectados los puntos característicos utilizando para ello filtros de calor y bordes, se ha tenido especial cuidado con los falsos positivos, para lo cual se han creado varios filtros que cada punto debía pasar para ser validado.

2. El siguiente subobjetivo era extraer información espacial 3D a partir del análisis 2D realizado, esto se ha realizado mediante la extracción de información de posición que hemos descrito en la sección 4.3. Esto ha servido como ayuda para los algoritmos de localización, sirviendo de función de coste en Monte Carlo y como función salud en el algoritmo evolutivo. Para lograr extraer esta información 3D, se ha hecho uso intensivo de la geometría proyectiva mediante el componente *Kinematics*, que fue descrito en la sección 3.8.

   Además se ha utilizado información precalculada de puntos cercanos en 3D para mejorar la eficiencia del algoritmo, como vimos en la sección 4.3.1.

3. Tras extraer la información de posición en una única observación, el siguiente paso era lograr que nuestro robot se localizase en 3D mediante la acumulación de observaciones. Esto se ha cumplido con el desarrollo de los dos algoritmos que vimos en el capítulo 5: el método de Monte Carlo, y el algoritmo evolutivo, que ha sido específicamente diseñado y desarrollado por nosotros.

   Además, los algoritmos son capaces de recuperarse ante secuestros mediante los mecanismos propios de cada uno de los algoritmos o con la ayuda del componente FSR, que describimos en la sección 3.10, capaz de informar a los algoritmos cuando el robot se cae o es levantado. Este componente es especialmente importante puesto que mejora el tiempo de respuesta ante secuestros de los algoritmos y además, fue desarrollado a pesar de no formar parte de los objetivos iniciales del proyecto.

4. Los algoritmos desarrollados han sido validados experimentalmente tanto en los simuladores (Webots y Gazebo), como con el robot Nao real, como se especificaba en los objetivos. El código desarrollado se ha integrado totalmente en el código del equipo de fútbol robótico de la URJC, donde cabe destacar que el algoritmo evolutivo fue probado y utilizado durante una competición real, el Mediterranean Open de marzo de 2010.

   Estos experimentos han demostrado que es viable aplicar las técnicas desarrolladas en el contexto real de la RoboCup, sirviendo como alternativa a otros algoritmos clásicos utilizados normalmente, como el propio método de Monte Carlo, o los filtros extendidos de Kalman.

Requisitos

Los requisitos que han tenido que cumplir nuestras aplicaciones han estado marcados por lo especificado en el capítulo 2.2. Como ya establecimos allí, se ha utilizado la arquitectura BICA para integrar nuestros algoritmos con el resto del código disponible hasta ahora en esta arquitectura, desarrollando nuestras aplicaciones en C++.

Esta arquitectura ha sido enriquecida gracias a nuestro proyecto, puesto que además de los componentes creados para cada uno de los algoritmos desarrollados, también se han creado otros componentes que ya son utilizados por el resto de miembros del equipo, estos son: *Kinematics*, para cálculos con geometría proyectiva, *FSR*, que interactúa con los sensores FSR del robot, *WorldModel*, que describe el modelo del mundo y facilita su modificación, así como otras muchas mejoras en la percepción, como el cálculo del fin del campo o la detección de obstáculos.

Además, como ya hemos visto, también se ha utilizado inicialmente la arquitectura JdeRobot, puesto que nos facilitaba el desarrollo al contar con un gran número de librerías y drivers ya desarrollados, y que nos permitían la comunicación de forma sencilla con los simuladores y el robot real.

En cuanto a los algoritmos realizados, se ha utilizado ampliamente la visión artificial para generar nuestro modelo de observación. Para ello, se ha utilizado la librería de visión OpenCV, que puede encontrarse en numerosas plataformas. Esto, junto al hecho de que sólo hemos utilizado una cámara para desarrollar nuestro modelo de observación, hace que nuestros algoritmos puedan ser aplicados en otros robots que cuenten con este sensor, independientemente de sus características.

La eficiencia de los algoritmos ha primado durante el desarrollo de los algoritmos, intentando siempre realizar métodos con la mayor eficiencia temporal posible. Esto ha hecho que nuestro trabajo haya podido ser utilizado finalmente en las competiciones reales sin un coste computacional muy alto.

La robustez de los algoritmos es muy alta, gracias a que en el modelo de observación se han intentado minimizar los falsos positivos. Esto hace que los algoritmos tengan un buen comportamiento incluso cuando se producen oclusiones debido a la presencia de otros robots.

En cuanto a la precisión, como hemos visto en la sección anterior, ambos algoritmos cuentan con un error medio inferior al que establecimos en los requisitos, siendo el error medio en el robot real del método de Monte Carlo de 176.22 mm y el del algoritmo evolutivo de 143.68 mm.

6.3. Trabajos futuros

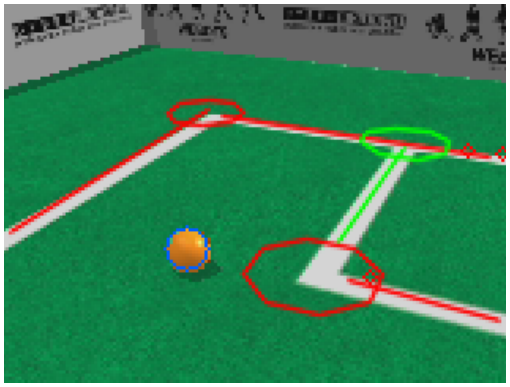
El trabajo realizado en el proyecto puede ser continuado más adelante por otras vías que describiremos a continuación.

Mejoras en el modelo de observación

Una mejora muy importante que podría hacer que el comportamiento y la robustez de los algoritmos aumentase, sería la de desarrollar un modelo de observación con información más rica que los puntos. En nuestro modelo, la evaluación de la posición del robot se realiza mediante la detección de puntos independientes. Sin embargo, podrían detectarse en la imagen objetos que nos proporcionasen más información, como por ejemplo reconociendo las líneas del campo como rectas y calculando las intersecciones entre ellas.

En este sentido, actualmente se está trabajando en un modelo de observación nuevo que tenga en cuenta este tipo de elementos, como por ejemplo, la detección del círculo central, el cálculo de la geometría de la portería, las intersecciones, etc. En la figura 6.1 podemos ver un ejemplo del estado actual de este nuevo modelo, donde se detectan en la imagen las líneas y sus intersecciones.

Se prevé que en un futuro, este nuevo modelo de observación se incorpore a nuestro algoritmo, lo que hará que mejoren los resultados.



(a)

Figura 6.1: Nuevo modelo de observación.

Localización compartida

Otra de las mejoras que podrían realizarse, es la de permitir la comunicación entre varios robots para que ellos actualicen su posición en función de su propia localización y la información recibida a través de los otros robots. Esto haría que mejorase la precisión y la fiabilidad de la localización de cada uno de los robots; sin embargo, sería necesario realizar algoritmos de detección de otros robots fiables, que permitiesen identificar qué robot estamos viendo en un momento determinado.

Ampliación a otros campos

A partir de este proyecto, puede ampliarse la utilización de estos algoritmos a otros campos no tan controlados como el terreno de juego de la RoboCup. Por ejemplo, podría usarse el algoritmo evolutivo para localizar a un robot en un edificio, como el departamental de la universidad.

Esto haría que entrasen en juego nuevos retos, sobre todo con el modelo de observación, puesto que factores como la iluminación o el no tener un medio tan controlado harían que la percepción de elementos con los que localizarse de forma fiable fuese más complicado.

Algoritmos mixtos

Los métodos que hemos implementado están pensados para poder tener una localización fiable mediante la acumulación de observaciones, lo que hace que no se vean afectados por falsos positivos en observaciones independientes o por oclusiones. Sin embargo, la precisión

de este tipo de algoritmos es mejorable. Por ejemplo, existen algoritmos conocidos como MonoSLAM [Smith *et al.*, 2006], que están basados en localizarse a partir del movimiento entre observaciones y que cuentan con precisiones mucho mayores; no obstante, estos métodos fallan en caso de observaciones independientes, errores u oclusiones, por lo que son muy frágiles.

Por ello, podría estudiarse el uso de métodos mixtos entre estos dos tipos de localización, que permitiesen una localización más robusta y precisa.

Bibliografía

[Agüero, 2010] Carlos Agüero. Técnicas de percepción compartida aplicadas a la asignación dinámica de roles en equipos de robots móviles. *Tesis doctoral - Universidad Rey Juan Carlos*, 2010.

[Barrera, 2008] Pablo Barrera. Aplicación de los métodos secuenciales de monte carlo al seguimiento visual 3d de múltiples objetos. *Tesis doctoral - Universidad Rey Juan Carlos*, 2008.

[Crespo, 2003] María Ángeles Crespo. Localización probabilística en un robot con visión local. *Proyecto Fin de Carrera. ITIS - Universidad Rey Juan Carlos*, 2003.

[Domínguez, 2009] José Manuel Domínguez. Autolocalización probabilística visual de un robot en el simulador gazebo. *Proyecto Fin de Carrera. Ing. Informática - Universidad Rey Juan Carlos*, 2009.

[García, 2007] Iván García. Reconstrucción 3d visual con algoritmos evolutivos. *Proyecto Fin de Carrera. ITIS - Universidad Rey Juan Carlos*, 2007.

[Hidalgo Blázquez, 2006] Víctor Manuel Hidalgo Blázquez. Comportamiento de persecución de un congénere con el robot pioneer. *Proyecto Fin de Carrera. ITIS - Universidad Rey Juan Carlos*, 2006.

[Kachach, 2005] Redouane Kachach. Localización del robot pioneer basada en láser. *Proyecto Fin de Carrera. ITIS - Universidad Rey Juan Carlos*, 2005.

[Kachach, 2008] Redouane Kachach. Calibración automática de cámaras en la plataforma jdec. *Proyecto Fin de Carrera. Ing. Informática - Universidad Rey Juan Carlos*, 2008.

[López, 2005] Alberto López. Localización visual del robot pioneer. *Proyecto Fin de Carrera. ITIS - Universidad Rey Juan Carlos*, 2005.

[López, 2010] Luis Miguel López. Autolocalización en tiempo real mediante seguimiento visual monocular. *Proyecto Fin de Carrera. Ing. de Telecomunicación - Universidad Rey Juan Carlos*, 2010.

[Martín *et al.*, 2010] Francisco Martín, Carlos Agüero, José M. Cañas, and Eduardo Perdices. Humanoid soccer player design. *Robot Soccer, ed. Vladan Papic. IN-TECH, pp 67-100, 2010. ISBN: 978-953-307-036-0*, 2010.

[Martín, 2008] Francisco Martín. Aportaciones a la auto-localización visual de robots autónomos con patas. *Tesis doctoral - Universidad Rey Juan Carlos*, 2008.

[Martínez, 2003] Juan José Martínez. Equipo de fútbol con jde para la liga simulada robocup. *Proyecto Fin de Carrera. ITIS - Universidad Rey Juan Carlos*, 2003.

[Marugán, 2010] Sara Marugán. Seguimiento visual de personas mediante evolución de primitivas volumétricas. *Proyecto Fin de Carrera. Ing. Informática - Universidad Rey Juan Carlos*, 2010.

[Perdices, 2009] Eduardo Perdices. Autolocalización visual en la robocup basada en detección de porterías 3d. *Proyecto Fin de Carrera. Ing. Informática - Universidad Rey Juan Carlos*, 2009.

[RoboCup, 2009] RoboCup. Robocup standard platform league (nao) rule book. `http://www.tzi.de/spl/pub/Website/Downloads/Rules2009.pdf`, Mayo 2009.

[Smith *et al.*, 2006] P. Smith, I. Reid, and Andrew J. Davison. Real-time monocular slam with straight lines. *British Machine Vision Conference*, Septiembre 2006.

[Vega, 2008] Julio Vega. Navegación y autolocalización de un robot guía de visitantes. *Proyecto Fin de Carrera. Ing. Informática - Universidad Rey Juan Carlos*, 2008.

[Álvarez, 2001] José María Álvarez. Implementación basada en lógica borrosa de jugadores para la robocup. *Proyecto Fin de Carrera. ITIS - Universidad Rey Juan Carlos*, 2001.