INGENIERÍA TÉCNICA EN INFORMÁTICA DE SISTEMAS

Escuela Técnica Superior de Ingeniería Informática

Curso académico 2009-2010

Proyecto Fin de Carrera

Caminata basada en ondas acopladas para el robot humanoide Nao

Autor: Francisco Miguel Rivas Montero
Tutor: José María Cañas Plaza

¿Por qué estudiar en vez de aprender?,
¿acaso no es eso lo realmente importante?

Si no conozco una cosa, la investigaré. (Louis Pasteur)

Agradecimientos

Quisiera dar la gracias a todas aquellas personas que me han apoyado, ayudado y animado durante todo este proyecto. Gracias a ellos todo este trabajo ha sido un poco menos duro.

Antes de nada quisiera mencionar a José María Cañas Plaza, tutor de este proyecto durante estos dos largos años. Gracias por todo su apoyo y ayuda en este trabajo pero sobre todo por sus ánimos en los malos momentos y toda esa confianza y sabiduría que nos inculca en cada tutoría. Espero que este proyecto solo sea un comienzo.

A toda la gente del grupo de robótica por su ayuda incondicional y por el buen ambiente generando en el laboratorio. Sobre todo destacar a Eduardo Perdices por su ayuda, sobre todo al comienzo del proyecto, y sus conocimientos. También a toda la gente que ha compartido la carrera conmigo en especial al súper equipo Ïwatuxixi"que sois los mejores.

Además quisiera dar las gracias a toda la gente de la Universidad Rey Juan Carlos del campus de Alcorcón, por ese buen ambiente que se respira, así da gusto ir a trabajar, en especial a Lola, Alberto, María y Paco.

Quisiera también agradecer el apoyo y compresión de mi novia Noelia que ha sabido darme ánimos en los momentos más duros.

Y cómo no, a mi familia, en especial a mis padres y a mis tíos que sin ellos todo esto no habría sido posible. Muchísimas gracias por ser como sois y sobre todo por haberme hecho ser como soy ahora.

¡Gracias a todos!

Resumen

Con el avance en el mundo de la robótica empiezan a aparecer nuevos robots cada día, éstos son cada vez más sofisticados y con diferentes fines. Uno de estos nuevos tipos de robots son los humanoides y es en este grupo donde nos vamos a centrar, en concreto en el robot Nao de Aldebaran.

En este proyecto hemos tratado de crear una forma de andar propia para el robot que sea totalmente parametrizable con el fin de encontrar la marcha más eficiente posible. Para lograr este fin hemos diseñado una herramienta con la cual podemos controlar cualquier componente del robot y gracias a ella podemos reproducir cualquier movimiento del mismo.

Basándonos en movimientos periódicos hemos parametrizado por completo la forma de andar del robot y variando estos parámetros conseguimos que el robot se mueva de diferentes maneras. Una vez conseguido ésto, lanzamos un algoritmo de búsqueda formado por esos n parámetros para encontrar el modo de andar óptimo.

Aunque en este proyecto hemos utilizado el tamibén el robot real, las pruebas sobre formas de andar se han basado en el simulador debido a la posibilidad de automatizarlas así como la obtención de más información sobre odometría, tiempos, estabilidad...

Para el desarrollo de este proyecto se han usando varios lenguajes de programación como C y C++ bajo la plataforma JDErobot, que nos ha servido para facilitar la interconexión entre los diferentes componentes robóticos así como la posibilidad de poder usar sus componentes propios y Objective C para el componente del iPhone.

Índice general

Índice de figuras

Capítulo 1
Introducción

En este primer capítulo vamos a explicar, de forma general, el contexto de este proyecto. Desarrollaremos los conceptos básicos de la robótica y presentaremos las dificultades encontradas a la hora de generar movimientos con robots bípedos, que es el tema principal que abordaremos en este proyecto.

1.1. Robótica

La robótica se define como la ciencia y la tecnología de los robots que engloba tanto el diseño, la manufacturación y el software que se va a ejecutar en ellos. Combina disciplinas como son la mecánica, la electrónica, la informática, la inteligencia artificial, la ingeniería del control.

Robot Institute of America dio en 1979 la siguiente definición de robot: *"Dispositivo multifuncional reprogramable diseñado para manipular y/o transportar material a través de movimientos programados para la realización de tareas variadas."*

Los componentes básicos de un robot moderno son los sensores, encargados de captar la información del entorno del robot; los actuadores, encargados de interactuar con el entorno; la unidad de proceso, que analiza toda la información captada por los sensores y genera las señales correspondientes para cada uno de los actuadores; y el software que se ejecuta en la unidad de proceso. Aunque este último componente no es un componente hardware es una de las partes más importantes en un robot y es donde, precisamente, se centra este proyecto.

Es en el apartado de software donde más está avanzando la robótica en la actualidad ya que disponemos de numeroso hardware avanzado para poder crear los robots, como receptores GPS, sensores de alta calidad, potentes actuadores... pero necesitamos un buen software que gestione todos estos componentes. El software que se ejecuta dentro del robot es lo que diferencia la funcionalidad del mismo, puede hacer que el robot sea un robot de compañía capaz de interactuar con personas, aprender, etc. o ser el mejor jugador de fútbol con fines competitivos como los usados en la RoboCup.

El primer robot completamente programable y dirigido de forma digital data de

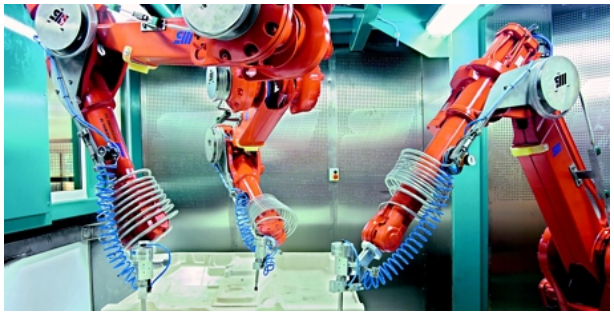


Figura 1.1: Robot*Kiva*

1961, fue diseñado por Unimate [1] y desarrollado para levantar piezas calientes de metal de una máquina de tinte y colocarlas correctamente.

A partir de ese momento la robótica se empieza a encaminar fundamentalmente al entorno industrial, aparecen fábricas con algunos robots para realizar tareas específicas o para ayudar al personal humano de las fábricas, como por ejemplo las células robotizadas Waterjet 3D (figura 1.1) del fabricante Idasa capaz de realizar cortes sobre metales o cerámica en 3 dimensiones. Hoy en día existen fábricas capaces de trabajar de forma totalmente automática únicamente con robots, un claro ejemplo es una fábrica de Mini Cooper (figura 1.2(a)), que es capaz de realizar el ensamblaje de todas las piezas de un coche de forma totalmente mecanizada por robots, aunque Honda (figura 1.2(b)) y Hyundai (figura 1.2(c)) también poseen fabricas de este tipo.

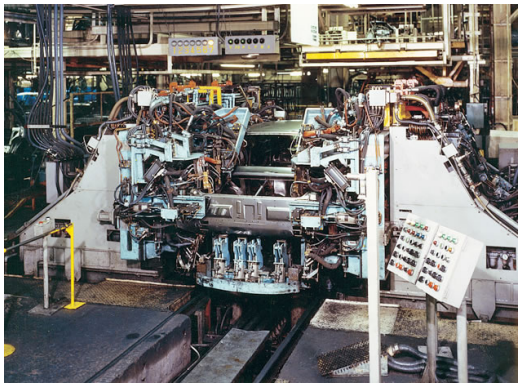
En otro tipo de fábricas donde también son muy frecuentes los robots es en las factorías de alimentación donde, el trabajo de envasado y empaquetado se hace de automática (figuras 1.3(a) y 1.3(b).

Otro tipo de robots con grandes capacidades móviles y que se están expandiendo en la actualidad son los robots de *Kiva* 1.4 del fabricante *Kiva Systems* [2], estos robots trabajan como mozos de almacén y son capaces de gestionar un centro de distribución ellos solos, se pueden desplazar a gran velocidad, cargar con grandes pesos, trabajar en condiciones de temperatura extrema y sin necesidad de luz, comunicarse mediante redes inalámbricas y ejecutar procesos logísticos. Todo esto nos aporta grandes ventajas con respecto a los trabajadores humanos.

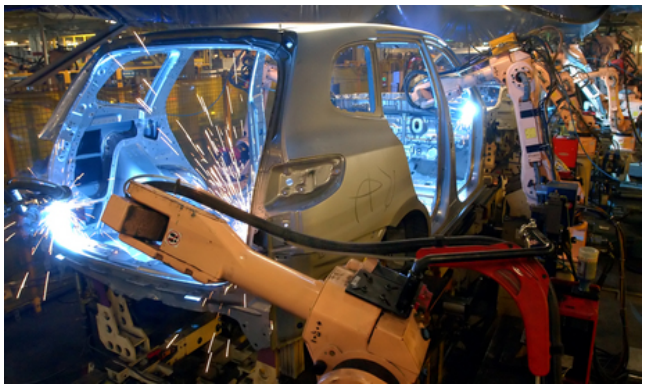
Hoy en día un campo donde tiene gran cabida la robótica es en la medicina, debido a la gran complejidad de algunas operaciones y la precisión necesaria para realizarlas correctamente es necesario el uso de algún tipo de herramienta que ayude a los médicos o cirujanos. Dos ejemplos de estos robots con fijes médicos son el robot cirujano Leonardo Da Vinci(figura 1.5(a)) , empleado como herramienta en intervenciones quirúrgicas, y el robot PnuStep (figura 1.5(b)) , capaz de realizar biopsias durante la realización de una resonancia magnética nuclear.

[1] http://www.unimate.com/
[2] www.kivasystems.com/

(a)



(b)



(c)

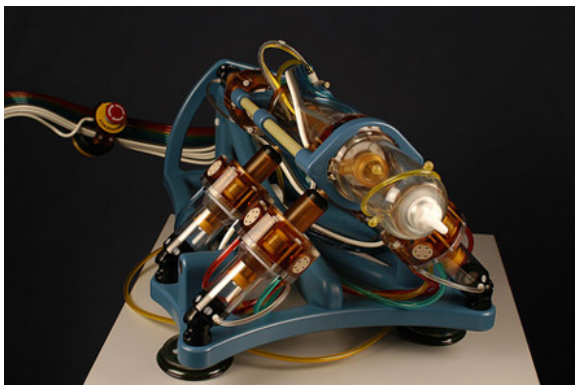
Figura 1.2: Fábrica de Mini (a). Fábrica de Honda (b). Fábrica de Hyundai

(a)

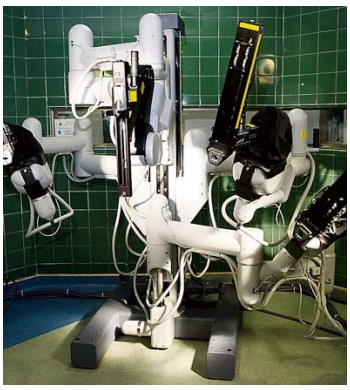
(b)

Figura 1.3: Automatización del proceso de envasado utilizando robots

Figura 1.4: Robot*Kiva*



(a)



(b)

Figura 1.5: Robot PneuStep (a). Robot cirujano *Leonardo Da Vinci* (b).

También empiezan a aparecer robots con fines domésticos como robots para la limpieza y mantenimiento del hogar e incluso robots médicos avanzados capaces de realizar biopsias y RMN 1.5(a) o complicadas operaciones quirúrgicas 1.5(b).

Como era de prever, también la robótica ha hecho su aparición en el sector del entretenimiento como es el caso del robot Aibo de Sony 1.6(a) que han diseñado un robot de compañía o Lego con su NXT 1.6(b).

Uno de los robots que más se ha popularizado en el entorno doméstico es el robot *Roomba* 1.7, un aspirador autónomo con gran movilidad que se desplaza por toda la superficie del lugar en el que se encuentre evitando los obstáculos que pueda haber. Sólo es necesario ponerlo en el suelo, activarlo y del resto se encarga el propio robot sin necesidad alguna de supervisión.

En los últimos años una de las líneas de la robótica que más ha avanzado es la movilidad, se empieza a exigir que los robots sean capaces de desplazarse en distintos

(a) (b)

Figura 1.6: Robot *Aibo* de Sony (a). Robot NXT de Lego (b).

Figura 1.7: *Roomba*

Figura 1.8: *Urban Challange*

terrenos ya sean conocidos o desconocidos, utilizando para ello sensores que captan información del entorno y así localizar su posición o desplazarse hasta su destino.
Un reto que nace a partir de estos nuevos conocimientos es la *Urban Challange* 1.8, donde se realiza una carrera en un circuito urbano en la que participan vehículos autónomos que deben llegar de un punto a otro sin intervención humana obedeciendo a las normas de tráfico humanas y comportándose responsablemente en presencia de tráfico.

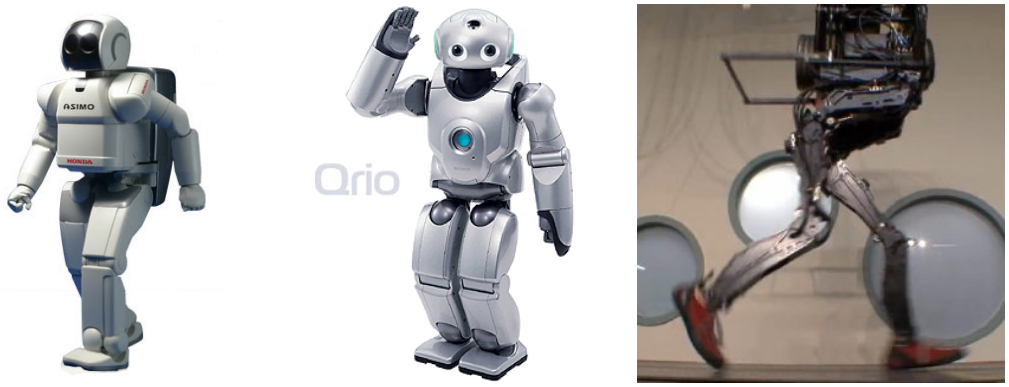
En otro de los campos en los que más destaca la robótica es en las misiones espaciales, las tareas que pueden realizar este tipo de robots son recogida de diferentes medidas o muestras, la exploración en todo tipo de terrenos y creación de cartografía, la construcción de estructuras... Pero lo más interesante en este campo, no es sólo la diversa funcionalidad que nos ofrecen los robots para este fin, sino que no es necesario el desplazamiento ni la presencia de ningún humano en la posición del robot ya que todas estas tareas pueden ser realizadas de forma autónoma o controladas a gran distancia.

1.2. Robots bípedos

La primera referencia a un robot humanoide data de 1206 cuando Al-Jazari construyó el primer robot automatizado. Este robot consistía en un barco con cuatro músicos robotizados y su mecanismo básico era un tambor programable con clavijas que chocaban con pequeñas palancas que accionaban instrumentos de percusión. Desde entonces la robótica ha avanzado en gran medida y actualmente los robots humanoides más destacados son: el robot Nao de Aldebaran (popularizado al ser uno de los robots elegidos en la RoboCup), el robot ASIMO de Honda (figura 1.9(a)), Q-RIO de Sony (figura 1.9(b)) o el robot PETMAN de Boston Dynamics (figura 1.9(c)).

De entre estos humanoides cabe destacar la labor realizada por Honda en la fabricación del ASIMO. Este fabricante japonés muy interesado en la locomoción bípeda

desde los años 80, presentó este modelo en el año 2000 basado en numerosos prototipos desarrollados en años anteriores. Este humanoide es uno de los más avanzados del mundo y capaz de andar, correr, subir y bajar escaleras, girarse... El robot ASIMO mide 130 cm y peso alrededor de 54 gramos, al igual que otros robots modernos tiene un aspecto muy similar al de un humano.



(a) (b) (c)

Figura 1.9: Robot *Asimo* de Honda (a). Robot *Qrio* de Sony (b). Robot *Petman* de Boston Dynamics (c).

Según su movilidad los robots pueden ser clasificados como móviles o fijos. Se puede entender como robot móvil aquel que es capaz de variar su posición de forma que cualquier sistema que se ligue a él sea móvil. Por el contrario, los robots fijos, aunque poseen articulaciones y mecanismos para realizar movimientos, no se consideran móviles dado que poseen un sistema de referencia ligado a él que sí que es fijo. Dentro de los robots móviles se pueden encontrar dos grandes grupos diferenciados por el tipo de locomoción: los robots con ruedas y los robots con patas. En este proyecto nos centraremos en el segundo grupo y más concretamente en los robots humanoides bípedos.

Las principales ventaja de un robot con ruedas son la estabilidad, la velocidad, la sencillez de movimientos y el control que nos ofrece sobre su locomoción al componerse básicamente de dos ruedas motrices y una rueda direccionable con lo que podemos reducir todo su movimiento en dos motores, de forma que variando estas dos señales obtendremos distintas formas de locomoción.

Frente a este tipo de robots, los humanoides tienen una ventaja importante, la movilidad. Un robot bípedo se puede mover por terrenos irregulares, subir y bajar escaleras, saltar... por lo que podemos generar formas de locomoción mucho más variadas y flexibles lo que le permite desplazarse sin problemas en cualquier entorno humano como por ejemplo una casa.

El gran problema de estos robots bípedos es la dificultad a la hora de generar los movimientos, como ya hemos dicho antes, podríamos controlar un robot con ruedas

de forma adecuada con sólo dos valores, pero trabajando con los humanoides esto se complica. En el caso del robot bípedo corriente tenemos tres articulaciones en cada pierna con un total de 6 grados de liberad en cada una de ellas, que tenemos que controlar de manera adecuada para que el robot ande correctamente. Otro factor a tener en cuenta es la estabilidad ya que la de un robot bípedo es significativamente menor que la de un robot con ruedas, un humanoide es fácil que se caiga sobretodo a la hora de andar.

Debido a todo lo explicado anteriormente encontrar una forma de andar óptima para un robot bípedo tiene gran complejidad ya que supone numerosas variables que deben estar sincronizadas entre sí para que el robot se desplace y no se caiga al andar. Dos de los principales factores que determinan una buena forma de andar son la velocidad y la estabilidad.

No existe una forma de andar general para todos los robots bípedos ya que ésta va a estar fuertemente ligada al hardware del robot por lo que tenemos que dar valores a cada uno de los motores que intervienen en el movimiento el vual va a depender del número de motores de los que disponga el robot.

Cada día la robótica se intenta acercar a las personas presentando robots mucho más amigables, como sucedió en 1999 con el Aibo, un robot de entretenimiento con aspecto de perro, y en estos últimos años con el Nao (figura 1.10), un robot bípedo que parece una persona de 58 cm. Son robots muy llamativos con un aspecto muy amigable por lo que facilita su incorporación al ámbito doméstico y con lo que se consigue una interacción social y una mejor aceptación en el mundo de entretenimiento y la educación.

1.3. Robots bípedos en la URJC

En la Universidad Rey Juan Carlos desde el Grupo de Robótica [3], se han desarrollado en los últimos años numerosos estudios y proyectos centrados en la robótica. Los temas principales que aborda este grupo son la lógica borrosa, visión artificial, teoría de control, elaboración de mapas, arquitecturas de control, comportamientos...
Gran parte de estos proyectos están orientados en el entorno de la RoboCup como por ejemplo la tesis doctoral de Martín, 2008, quien aportó soluciones para la localización en el campo de fútbol para robots Aibo o Agüero, 2010, con un nuevo método para la detección de la pelota y un mecanismo de intercambio de roles para los robots.

La RoboCup es un proyecto internacional fundado en 1997 creado para promover, mediante diferentes competiciones para robots autónomos, la investigación y la educación sobre inteligencia artificial. El objetivo último del proyecto es desarrollar

[3] http://www.robotica-urjc.es

Figura 1.10: Robot Nao de Aldebaran

para la RoboCup 2050 un equipo de robots humanoides completamente autónomos que sean capaces de ganar al equipo humano campeón del mundo de fútbol.
Para lograr estos objetivos de robots autónomos que sean capaces de jugar al fútbol es necesario el uso de Inteligencia Artificial y Sistemas Inteligentes para resolver tareas como la colaboración multiagente, la coordinación del robot, etc. Se usó el fútbol como actividad principal de la RoboCup porque es un deporte fácil de entender y que se práctica en todo el mundo, tiene interacción entre diferentes agentes y su entorno, por lo que es necesaria una cooperación entre todos los robots que componen un equipo, se puede aplicar todo tipo de tecnología, es un entorno dinámico que cambia continuamente y es competitivo ya que te enfrentas a un rival directo. La RoboCup no sólo se centra en robots con fines futbolísticos sino que también se realizan conferencias técnicas, tareas educativas... La RoboCup se desarrolla en varias modalidades, entre las que destaca la *RoboCupSoccer* 1.11.

Un proyecto interesante presentado en 2010 y también relacionado con la RoboCup ha sido el de Jorge Bermejo [4] quien ha diseñado un robot Nao para el simulador Gazebo (figura **??**), muy interesante sobre todo a la hora de realizar pruebas en un robot simulado. Con este nuevo modelo de robot y el driver proporcionado podremos extender las herramientas desarrolladas en nuestro proyecto a un nuevo entorno.

Como hemos comentado anteriormente uno de los temas de investigación es la generación de comportamientos. Actualmente la inmensa mayoría de los comportamientos desarrollados están basados en la arquitectura BICA, un ejemplo es el proyecto fin de carrera presentado por Raúl Benítez Mejías, *Componente BICA de coreografías para robot NAO aplicado a terapias en enfermos de Alzhéimer* en 2010.

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

Figura 1.11: Robots Nao en la RoboCup

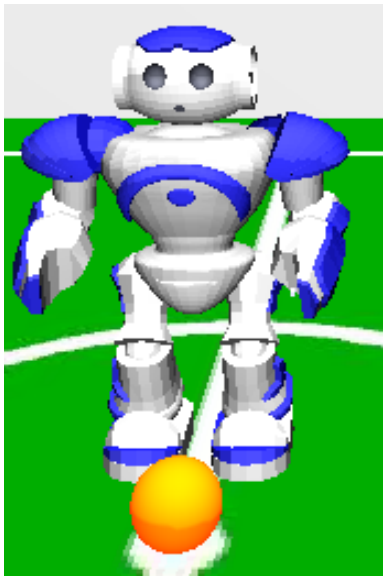


Figura 1.12: Robots Nao simulado en Gazebo

El grupo de robótica de la Universidad Rey Juan Carlos también organiza un proyecto de difusión de la ciencia y la tecnología llamado RoboCampeones[5], este concurso de construcción y programación de robots trata de fomentar las ingenierías entre los alumnos de secundaria presentando la tecnología como algo vistoso y divertido.

Otra de las principales líneas de investigación del Grupo de Robótica de la Universidad Rey Juan Carlos es una arquitectura llamada JDERobot (Jerarquía Dinámica de Esquemas), la cual es muy útil a la hora de trabajar con componentes robóticos, visión computacional, generación de comportamientos... y será la herramienta en la que basaremos todo este proyecto.

En este proyecto desarrollaremos las herramientas necesarias para dar soporte completo al robot Nao de Aldebaran Robotics dentro de la plataorma JDErobot, sobre esta arquitectura modelaremos una forma andar con un número reducido de parámetros con los que buscaremos una forma de andar óptima, es decir, rápida, estable y lineal.

En el siguiente capítulo (2 desarrollaremos los objetivos concretos que pretendemos cumplir, así como los requisitos subyacentes de éstos. A continuación, en el capítulo 3 explicaremos tanto la plataforma de desarrollo como las herramientas utilizadas para poder cumplir los objetivos. Dado que es un proyecto basado en la arquitectura de JDErobot y ésta no tiene soporte para el robot Nao tendremos que desarrollar todo lo necesario para que este soporte sea lo más completo posible y tener aplicaciones base sobre las que asentar el resto del proyecto, todo esto quedará explicado en el capítulo 4. En el capítulo 5 describiremos la forma elegida para modelizar la caminata del humanoide Nao y en el 6 explicaremos las estrategias adoptadas para realizar la búsqueda de la mejor de ellas. Por último presentaremos en el capitulo 7 las conclusiones finales del proyecto así como una perspectiva de futuro.

[5]http://www.robocampeones.es

Capítulo 2
Objetivos

Una vez presentados la motivación y 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 es la creación de una forma de andar para el humanoide Nao que sea rápida, segura y lineal, para ello será necesario desarrollar un soporte completo para el robot Nao dentro de la plataforma de JDErobot. Las formas de andar ofrecidas por el fabricante sobre su middleware son bastante cerradas y escasamente modulables, sobre todo en fiabilidad con velocidad elevada, linealidad y enlazado de movimientos. El robot no responde como deseáramos al intentar enlazar movimientos, por ejemplo, a la hora de cambiar la velocidad a la que está avanzando el robot es necesario colocarse en una posición de estabilidad y reiniciar la marcha con la nueva velocidad.

Este objetivo principal de crear una caminata propia lo hemos articulado en varios subobtetivos:

1. Crear un driver que nos permita acceder a los componentes del robot Nao: en especial nos interesan los actuadores y sus sensores pero también sería útil acceder a otro tipo de sensores como el inercial, el de presión, etc. Este driver sentará la base para el segundo subobjetivo.

2. Crear un esquema capaz de teleoperar el robot controlando todo lo que nos ofrece el driver del que hemos hablado en el punto anterior: este esquema deberá ser capaz de interactuar con el robot real y permitirnos el control del mismo de una forma fácil e intuitiva. Este esquema será básicamente un teleoperador completo del robot Nao en el cual podemos ir añadiendo herramientas con las que crear y evaluar movimientos.

3. Conseguir una forma de andar totalmente parametrizada para nuestro robot diseñando una manera de modelizar el recorrido que debe realizar cada actuador

del robot durante el ciclo de la marcha.

4. Automatizar el procedimiento de búsqueda, creación y evaluación de movimientos para su ejecución en un simulador, lo cual nos va a permitir encontrar caminatas que se ajusten a nuestras especificaciones (rápida, estable y lineal). En este subobjetivo nos centraremos en le interconexión del simulador con el esquema así como el desarrollo de todas las herramientas y aplicaciones necesarias para su correcto funcionamiento. También tenemos que encontrar la forma más eficiente de realizar la evaluación del movimiento para que cada evaluación dure lo menos posible y así poder probar el máximo número de movimiento en el menor tiempo posible.

2.2. Requisitos

Para cumplir los objetivos presentados en el punto anterior, nuestro proyecto deberá satisfacer una serie de requisitos:

1. Este proyecto se desarrollará bajo la arquitectura JDErobot, programando todos los componentes usados mediante C, C++ u Objective C, y usando el middleware del fabricante para la conexión con el Robot tanto real como simulado.

2. Una vez obtengamos una caminata que cumpla nuestros criterios y que funcione de manera correcta en el simulador deberemos validarla usando el robot real.

3. La velocidad de la nueva forma de andar deberá ser, como mínimo, igual de rápida que la ofrecida por el fabricante.

2.3. Metodología

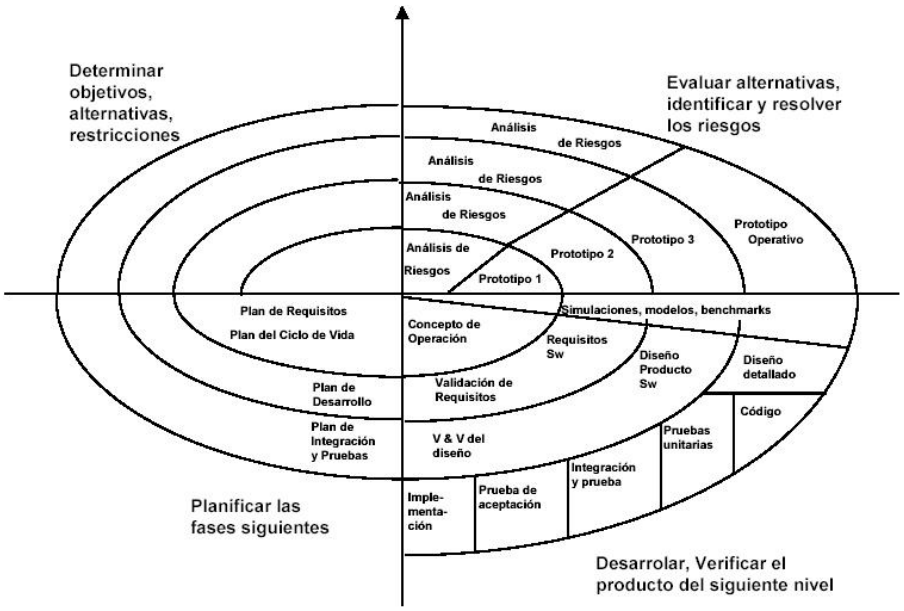
En el desarrollo de los componentes software de nuestro trabajo, el modelo de ciclo de vida utilizado ha sido el modelo en espiral basado en prototipos, ya que permite desarrollar el proyecto de forma incremental, aumentando la complejidad progresivamente y haciendo posible la generación de prototipos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

- 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.

Durante toda la realización de este proyecto se ha utilizado una herramienta web donde se han ido reflejando las avances realizados, el mediaWiki. Esta herramienta nos ha servido como cuaderno de bitácora y como apoyo fundamental en las reuniones semanales, en las páginas web [1] y [2] están reflejados todos los pasos dados desde el

[1] https://jderobot.org/index.php/Frivas-pfc-itis
[2] http://jderobot.org/index.php/Frivas-pfc-itis2

comienzo de este proyecto a principios de 2009.

Debido al alto coste del robot Nao es necesario la utilización de una plataforma alternativa para la realización de las pruebas iniciales. Un entorno simulado sería perfecto para este fin ya que supone disponer de una plataforma en la que poder realizar pruebas sin ningún tipo de inconveniente. Lo que tratamos en este proyecto es que el robot aprenda a andar *por sí solo* a partir de una forma de andar parametrizada lo que supone numerosas caídas por lo que sería muy arriesgado realizar estas pruebas con el robot real.

Otra de las ventajas que supone la utilización de un simulador es la posibilidad de automatizar las pruebas, mientras que el reinicio de un simulador se puede hacer directamente desde nuestro programa colocar al robot en una posición inicial con el robot real es algo mas complejo.

Por todo esto finalmente nos hemos decantado por la utilización de un simulador para la realización de las pruebas iniciales y las búsquedas de movimiento y en una segunda fase probar las mejores formas de andar encontradas directamente sobre el robot real.
tuen

2.4. Planificación

A lo largo del desarrollo de este proyecto se han ido proponiendo una serie de requisitos siendo asesorado y en parte fijado por el tutor del proyecto en las diferentes tutorías que han tenido lugar. Las principales tareas a destacar son:

- **Familiarización con el entorno JDErobot:**
  En esta primera fase nos centraremos en recopilar información acerca del entorno JDErobot así como la integración en el grupo de robótica y el funcionamiento del mismo. Siguiendo la sistemática del grupo y a través de su lista de correo el conocimiento del funcionamiento de JDErobot fue bastante sencillo. En esta fase también nos familiarizaremos con el concepto de driver (componente que se comunica directamente con los componentes robóticos disponibles en un sistema) y esquema (componente encargado de gestionar y controlar los componentes a través de los drivers), siendo la principal tarea de esta fase la creación de nuestro propio driver y nuestro propio esquema para sentar la base de nuestro proyecto.

- **Webots y el Robot Nao simulado:**
  En esta fase nos tendremos que familiarizar con Webots y más concretamente con el robot Nao Simulado. Hay que distinguir entre un robot Nao Simulado en Webots y casi cualquier otro robot en este simulador ya que no controlaremos el robot directamente con el API de Webots sino que utilizaremos directamente el API ofrecido por el fabricante el robot (NaoQi), Aldebaran. El objetivo de utilizar

el API el fabricante y no el de Webots es simplemente porque si utilizamos el de Webots no lo podremos utilizar directamente sobre el robot real, pero si utilizamos el API del fabricante, el cambio del robot simulado al robot real es totalmente directo.

- **Creación del driver NaoBody:**
  La primera tarea importante de este proyecto es la creación de un driver que va a ser el encargado de conectarse con el robot (tanto simulado como real, siempre que utilice NaoQi). En esta tarea recibimos la ayuda de Eduardo Perdices quién ya había avanzado en el desarrollo de un driver que nos permitía acceder al cuello del Nao y a la cámara del robot. Eduardo se encargó de la creación de un script necesario para el compilado y enlazado entre todo el código y librerías que utiliza este driver (NaoQi, JDErobot, etc). A partir de este punto el resto de módulos disponibles en el driver NaoBody a excepción del módulo de odometría de los actuadores del robot fueron creados en este proyecto.

- **Desarrollo de un teleoperador para el Nao:**
  La segunda gran tarea de es la realización de una herramienta que nos permita teleoperar el robot (tanto simulado como real) usando el driver NaoBody. Este teleoperador recibe el nombre de NaoOperator y a través de él podemos tener acceso a cada uno de los componente físicos del robot real (del robot simulado tenemos acceso a todos los componentes disponibles). Usando este esquema de JDErobot podemos controlar de forma completa al robot Nao (acceso a actuadores, cámara, sensores lares, sensores inerciales, etc) de forma bastante intuitiva.

- **Incorporación de herramientas para creación de movimientos:**
  Debido a la necesidad de tener una herramienta que nos permita crear movimientos con el robot incorporamos al NaoOperator una nueva serie de funciones con este fin: creación de posiciones, movimientos, movimientos simétricos, opciones de carga y salvado de movimientos y configuraciones.

- **Modelización de la caminata del robot de modo compacto:**
  Una vez capturado un movimiento (la forma de andar ofrecida por el fabricante) la idea es crear una serie de funciones que emulen este movimiento para ser capaces de generar los mismos movimientos pero sin usar las funciones del API. Una vez conseguido esto y modelizadas las funciones que generan los movimientos podremos, a través de una serie de parámetros, crear nuevos movimientos. Como estas funciones serán ondas, los movimientos generados dependerán de una frecuencia, una amplitud de onda, un desfase y un desplazamiento de la onda.

- **Búsqueda de los mejores movimientos:**
  Una vez tenemos los movimientos totalmente parametrizados tenemos que crear una herramienta que sea capaz de evaluarlos. Estas nuevas funciones también han

sido añadidas al NaoOperator. Para evaluar los movimientos de forma correcta debemos conectar el simulador que vamos a usar con el esquema generador de movimientos para poder recoger toda la información necesaria y así disponer de los máximos datos posibles a la hora de fijar la nota que obtiene el movimiento.

- **Pruebas de los movimientos:**
  Una vez encontrados los movimientos con los requisitos buscados tendremos que evaluar los movimientos en entornos reales. En el simulador podremos comprobar el correcto funcionamiento del mismo y una vez verificado lo probaremos en el robot real para comprobar físicamente las mejoras.

Capítulo 3
Plataforma de desarrollo

En este capítulo vamos a describir tanto la plataforma hardware haciendo una breve descripción del robot utilizado así como los componentes del mismo y el software utilizado: plataformas, simuladores, librerías...

El sistema operativo utilizado en la versión final de este proyecto ha sido Ubuntu 10.04, una de las distribuciones más importantes de Linux basada en Debian, la versión más reciente de este sistema operativo que tiene soporte para todas las herramientas y aplicaciones que necesitamos.

3.1. El Robot NAO de Aldebaran

'Nao'[1] es un robot autónomo, programable y de mediana estatura, desarrollado por la empresa francesa Aldebaran Robotics con sede en París.
En el 15 de agosto de 2007, Nao sustituye al perro robot Aibo de Sony como la plataforma estándar para la RoboCup. A finales de marzo del año siguiente se les hace entrega a los participantes de la RoboCup la primera versión estable del robot: el *Nao RoboCup Edition* también conocido como V2. En 2009 Aldebaran Robotics pone a disposición de las Universidades y de la RoboCup una nueva versión con las correcciones de los problemas de fiabilidad, descubiertos durante el desarrollo de la RoboCup de 2008, denominada V3 y finalmente a mediados de 2009 aparece la versión V3+, que es la versión utilizada para el desarrollo de este proyecto, disponible únicamente para laboratorios y universidades. La versión comercial de este robot está prevista para 2011.
A día de hoy existen dos tipos del robot Nao, uno diseñado para su uso en la RoboCup y un segundo tipo, un poco más complejo que el primero, para Universidades y fines académicos. La principal diferencia entre ellos es que el segundo tiene en las dos manos capacidades de agarre mientras que el primero no.

Las características principales de la versión que nosotros hemos utilizado para nuestro proyecto son:

- 58 cm de altura.
- 4,3 Kilos de peso.
- Autonomía: 45 minutos (15 caminando)

[1] http://www.aldebaran-robotics.com

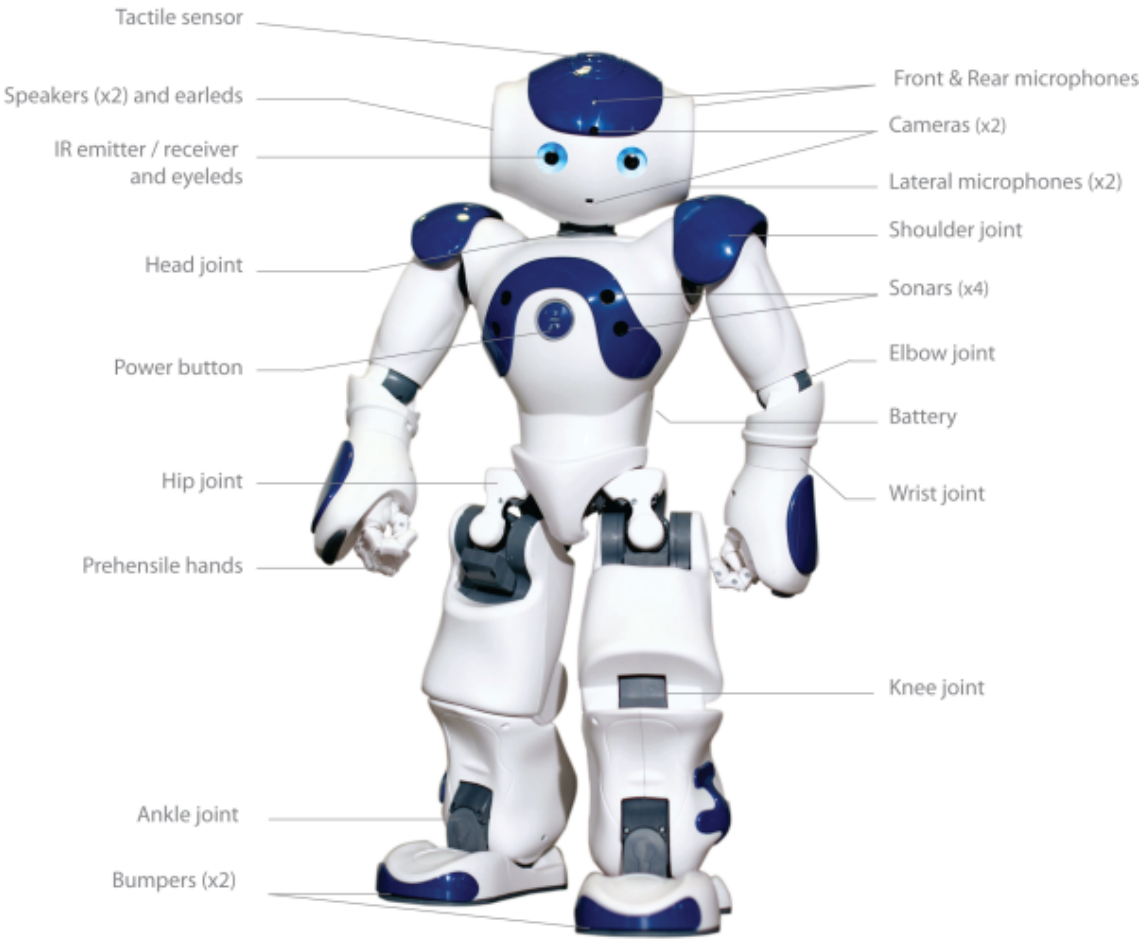


Figura 3.1: Descripción NAO (`Aldebaran`)

- Grados de libertad: de 21 a 25
- CPU: x86 AMD Geode a 500 MHz
- Plataforma: Linux
- Lenguajes de programación: C++, C, Python, Urbi
- Ethernet, Wi-Fi
- Sensor inercial
- 4 sensores de ultrasonido
- 4 micrófonos, 2 Hi-Fi, 2 cámaras CMOS
- Sintetizador de voz

El robot Nao dispone de una batería para uso autónomo que nos proporciona unos 45 minutos si el robot está en espera y unos 15 si está en movimiento. Para programar software para el robot el fabricante nos ofrece un API bastante completo para los lenguajes de programación enumerados anteriormente y una serie de herramientas que nos permiten la creación de módulos programables de forma bastante intuitiva. Estos módulos son llamados *Brokers* y serán explicados más a fondo en la próxima sección.

3.2. JDErobot

JDErobot [2] es una herramienta de desarrollo para aplicaciones de robótica, automatización de casas y de visión computacional. Este campo incluye sensores (por ejemplo, cámaras), actuadores, y software inteligente intermedio. JDErobot ha sido diseñado para ayudar a programar como un software inteligente. Está escrito en C y nos proporciona un entorno de programación básico donde las aplicaciones están hechas como una colección de hebras concurrentes y asíncronas llamadas **esquemas**. Cada uno de estos esquemas están cargados dinámicamente sobre esta aplicación.

JDErobot simplifica el acceso a los componentes físicos desde nuestro programa. Obtener la medida de un sensor es tan simple como leer de una variable local mientras que mandar una orden para un motor es como escribir en la variable. Esta plataforma actualiza todas las variables de los sensores con lecturas de refresco así como las variables de los actuadores. Nuestra aplicación robótica lee y escribe las variables como describe nuestro comportamiento. Internamente estas variables pueden estar conectadas a un simulador o directamente a los sensores reales, de las dos formas puede ser localmente o remotamente. Varios **drivers** han sido desarrollados para dar soporte a diferentes sensores físicos, actuadores y simuladores. Los **drivers** son usados como *plugins* instalados y dependen de nuestra configuración. Están incluidos en las versiones oficiales.

Debido al gran nivel de encapsulación que nos ofrece JDErobot la tarea de interactuar con el gran número de componentes de los que dispone el robot será más sencilla. Dispondremos de todos los componentes en el driver diferenciados por módulos que podrán ser activados de forma individual.
JDErobot también nos ayuda a solucionar problemas a la hora de acceder a datos de manera concurrente y de controlar los tiempos de iteración de cada uno de los esquemas y de los drivers.
La versión de JDErobot utilizada para realizar este proyecto es la última versión estable que es la 4.3.

[2] http://JDErobot .org

3.3. NaoQi

El robot Nao funciona bajo la plataforma Linux y puede, para programar con el robot, usar un kit software de desarrollo llamado NaoQi el cual ofrece la posibilidad de programar en C, C++, Ruby y Urbi. Este kit de desarrollo también es compatible con el simulador Webots. NaoQi esta disponible tanto para Windows como para Linux.

Debido a la larga duración de este proyecto hemos utilizado diferentes versiones de NaoQi. Empezamos con una de las primeras versiones, la 1.2.0 y continuamos hasta la ultima versión compatible con webots, la 1.3.17, que es la versión que hemos utilizado finalmente para asentar todo el proyecto.

Este SDK está basado en una arquitectura cliente-servidor donde es el propio NaoQi quien actúa como servidor. Los diferentes módulos son incorporados en NaoQi como librerías o como *Brokers*, la comunicación se hace a través de IP con NaoQi. Gracias a esto tenemos la posibilidad de ejecutar el código tanto directamente sobre el robot o desde una máquina remota. Nosotros nos centraremos en esta segunda parte ya que todo nuestro código va a ser ejecutado en un principio en una máquina remota con la que, mediante la comunicación que ofrece NaoQi, controlaremos al robot, ya sea simulado a través de webots o en el robot real. Algunos de los módulos que ofrece NaoQi y que hemos utilizado en este proyecto son:

- ALCamera: este módulo es el encargado de la comunicación con las dos cámaras del robot. Al no poder acceder a las dos cámaras a la vez también es el encargado de gestionar el cambio entre cámaras así como toda la configuración de las mismas. La configuración básica de este módulo es:
  - Nombre identificador: es el nombre con el que se subscribe el módulo al sistema del robot y actúa como identificador.
  - Resolución: las cámaras del Nao pueden trabajar con tres tamaños diferentes: VGA (640*480), QVGA (320*240) y QQVGA (160*120)
  - Espacio de colores: el Nao es capaz de ofrecernos imágenes en diferentes formatos: YUV422 (formato nativo de la cámara), YUV (24 bits), Y (8 bits), RGB (24 bits), BGR (24 bits), HSY (24 bits).
  - Frames por segundo.

- ALMotion: este es el módulo encargado de la locomoción del robot. Se encarga de facilitar el control de los movimientos del Nao, ofreciendo funciones simples para el control de los actuadores en el espacio, manipulación del centro de masa y movimientos de alto nivel como por ejemplo: *camina recto 10cm*.
  Este módulo nos ofrece diferentes posibilidades para las siguientes funcionalidades:
  - Resolver el modelo cinemático del robot.

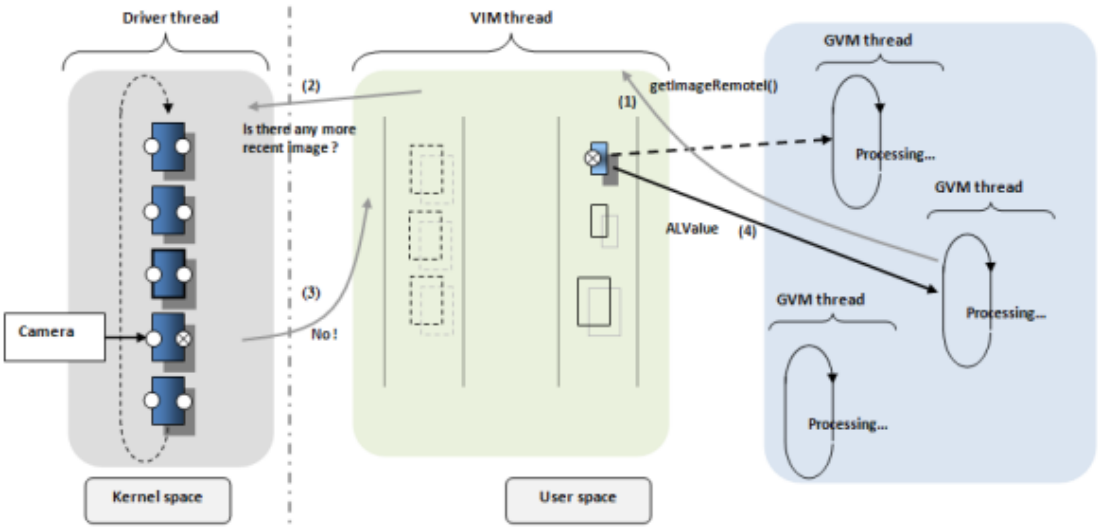


Figura 3.2: Esquema del funcionamiento estándar de la cámara del Nao

- Controlar el espacio articular del robot.
- Controlar la orientación del torso.
- Controlar el centro de masa.
- Crear y controlar patrones de movimiento.
- Controlar parámetros de los componentes físicos del robot como puede ser la rigidez de la articulaciones.

Una de las funcionalidades que nos ofrece este módulo y que será utilizada a lo largo de todo el proyecto es la posibilidad de interpolar entre distintas posiciones, por lo que, no es necesario proporcionar un afluente de posiciones con pequeños cambios, solo una posición y un tiempo.

Como ya hemos dicho anteriormente ALMotion es el encargado de la locomoción del robot por lo que tendremos que utilizarlo para conseguir que el robot se mueva. Este módulo nos da acceso a las formas de andar ofrecidas por el fabricante que son básicamente cuatro:

1. Andar en línea recta.
2. Andar de forma circular.
3. Andar de forma lateral.
4. Girar Sobre sí mismo.

En todos los casos la velocidad de la marcha se fija no siguiendo unos parámetros espacio-temporales sino en el tiempo que tarda el robot en dar un paso.

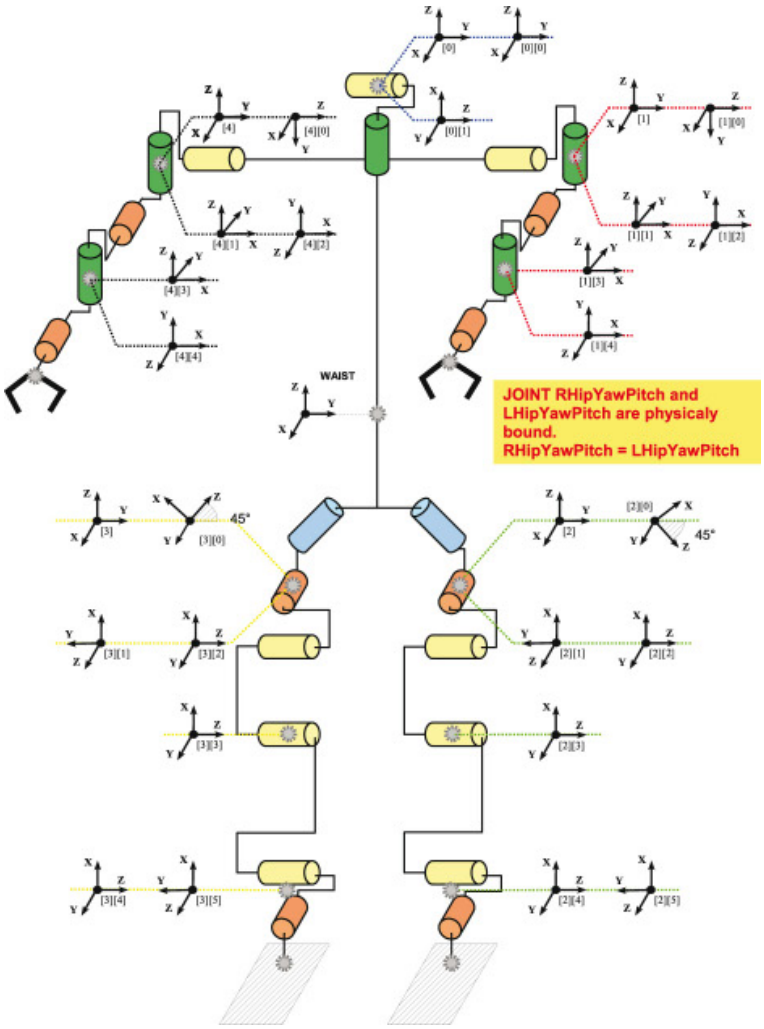


Figura 3.3: Esquema Denavit-Hartenberg del robot Nao

Ya que este módulo nos da acceso individual a cada uno de los actuadores será el más usado en nuestro proyecto sobre todo a la hora de generar movimientos. Las formas que nos ofrece NaoQi para acceder a los actuadores del robot son básicamente dos: de forma directa asignándole una posición con una velocidad o como hemos explicado antes, asignándole una posición y el tiempo que queremos que tarde en llegar a ella.

- ALMemory: memoria basada en eventos, usado principalmente para leer los valores de los sensores. Esta implementación es muy útil ya que ALMemory es capaz de generar eventos o notificaciones cada vez que un valor almacenado es modificado.
- ALTextToSpeech: módulo sintetizador de voz.
- ALSonar: módulo que ofrece acceso al sensor de ultrasonidos.

NaoQi basa su programación en módulos independientes llamados *Brokers* que son básicamente ejecutables que se conectan con el robot a través de una IP y un Puerto.

Todos estos nuevos *Brokers* se conectan a un *Broker* principal llamado *MainBroker* como se puede apreciar en el figura 3.4.
La forma de crear un *broker* sencillo es la siguiente:

```
int main (int argc, char *argv[])
    // create a broker that contain module.
    // Create broker will automaticaly load network dynamic
    //library and run server
    ALPtr<ALBroker> broker = ALBroker::createBroker("myBroker",
        "127.0.0.1", "9999", "", 0);
    ......
}
```

Listing 3.1: Creación de un *broker*

Al no utilizar *Brokers* para nuestro programa nuestro driver de JDErobot tendrá un *proxy* activo con cada uno de los componentes habilitados y éste sera el punto de conexión entre el robot y JDErobot . La forma de creación de un *proxy* a un componente es sencilla, veamos como se crearía un *proxy* a ALMotion:

```
int main (int argc, char *argv[])
    try {
        std::cout << "Trying to conect to motion in " << "127.0.0.1" << ":"
            << "9999" << std::endl;
        this->motionProxy = new AL::ALMotionProxy("127.0.0.1", "9999");
    } catch(AL::ALError& e) {
        std::cerr << "NaoBody: exception connecting to NaoQi:
            "<<e.toString()<< std::endl;
        return -1;
    }
}
```

Listing 3.2: Creación de un *proxy*

Nosotros no vamos a utilizar esta arquitectura para el grueso de nuestra aplicación ya que no vamos a programar sobre *Brokers* sino que queremos utilizar JDErobot como arquitectura base de nuestro proyecto. NaoQi también nos da la posibilidad de acceder a cada uno de los componentes del robot de forma independiente utilizando proxies. Podemos crear un *proxy* a cada componente que necesitemos desde nuestro código en JDErobot y así utilizarlo desde nuestro driver.

3.4. Webots

Para comprobar el funcionamiento de las herramientas que hemos ido desarrollando nos hemos apoyado en un simulador para no tener que usar el robot real cada vez que queramos probar una mejora. Esto nos ha reducido mucho el tiempo de pruebas y tras varias consideraciones decidimos usar el simulador Webots de Cyberbotics [3]. El

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

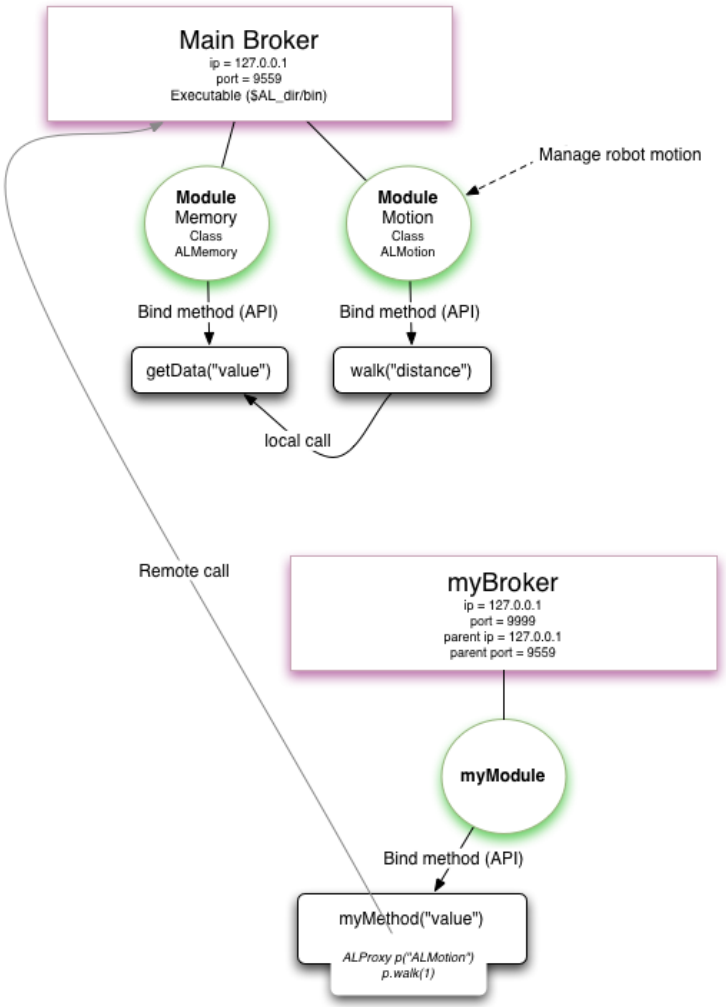


Figura 3.4: Esquema del funcionamiento de un *broker* en NaoQi

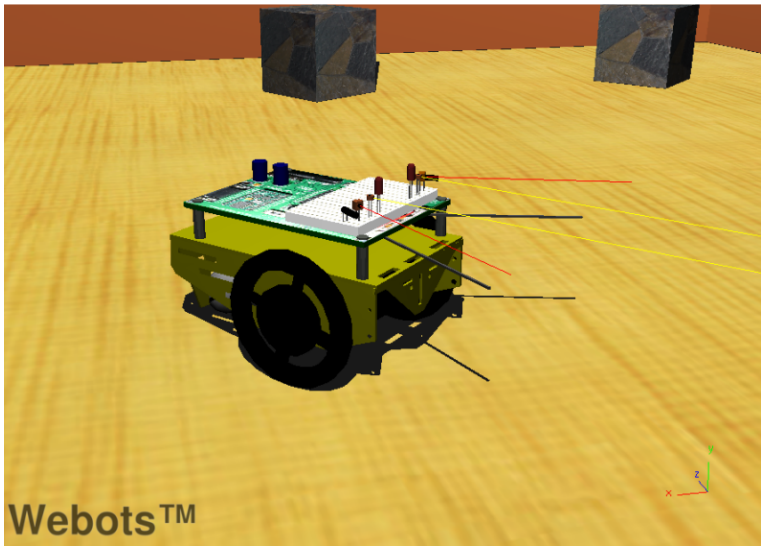


Figura 3.5: Boe-Bot en Webots

proyecto Webots se inicia en 1996 y fue desarrollado por el Doctor Olivier Michel en el *Swiss Federal Institute of Technology* en Suecia.

Webots es un simulador profesional de robots orientado principalmente a fines educativos que se apoya en ODE (Open Dynamics Engine) para la detectar colisiones y para simular la rigidez del robot. ODE permite simular propiedades físicas de objetos como la velocidad, la inercia o la fricción, con este sistema conseguimos una simulación del robot Nao bastante realista.

Además este simulador incorpora numerosos modelos de robots que podemos simular pero a su vez nos da la posibilidad de crear nuevos robots desde cero. Algunos de éstos modelos incorporados en Webots son el Aibo (ERS7 y ERS210), Boe-Bot 3.5, E-puck, Hoap2, Lego Minstorms, Pioneer 2, Scout 2 y el Nao (V2 y V3).
La programación de robots en el simulador Webots está basada en controladores, cada robot ejecuta el código de su controlador que es quien va a definir su comportamiento, a su vez hay un controlador superior al de los robots, llamado supervisor, encargado de controlar la posición de los robots, objetos y diferentes situaciones que pueden surgir, por ejemplo, en el entorno de la RoboCup el encargado de definir qué pasa cuando el balón entra en la portería o sale fuera del campo es el supervisor.

El controlador que hemos usado ha sido desarrollado conjuntamente entre *Cyberbotics* y *Aldebaran Robotics* para Webots y es un ejecutable cerrado que no podemos modificar y que nos permite conectarnos con el robot simulado en Webots a través del API de NaoQi por lo que podremos conectarnos a él directamente usando *Brokers* a través de una IP y un puerto (configurable desde el simulador). Todo esto nos permite probar aplicaciones en el simulador y no tener que hacer ningún cambio para ejecutarlo en el robot real (simplemente tendremos que cambiar la IP y el puerto en caso de que difieran entre el robot simulado y el robot real). Sin embargo al no dar

soporte completo del robot, hay componentes del robot que no estan disponibles en el simulador y como es cerrado nosotros tampoco podemos incorporarlos, siendo algunos de estos componentes el sensor inercial, el sensor de ultrasonidos, los LEDs, etc.

Los controladores pueden estar escritos en C, C++, Java, Python y MATLAB, aunque los robots AIBO, Nao y E-puck también pueden escribirse en URBI con su correspondiente licencia.

Para determinadas labores a la hora de evaluar los movimientos hemos necesitado utilizar algunas funciones propias de Webots como la posición del robot dentro del campo para determinar la distancia recorrida por el robot, debido a esto al ejecutar nuestro programa tendremos que hacer algunas especificaciones en el fichero de configuración de JDErobot sobre si estamos usando el robot real o el simulado.

3.5. Gazebo

Gazebo[4] es una alternativa de código abierto (licencia GNU) a Webots, de código cerrado, que ha sido desarrollado por un grupo internacional de investigadores orientados a la robótica. Gazebo es un simulador en 3D para robots que, al igual que Webots, utiliza ODE para simular la física de los robots y del entorno, la gran pega es que por el momento no hay ningún humanoide para este simulador y por este motivo nos hemos decantado por Webots como simulador base para nuestro proyecto.

Sin embargo hay un proyecto fin de carrera en curso con el propósito de crear un robot Nao dentro del entorno de Gazebo, este proyecto lo está realizando Jorge Bermejo[5] ,alumno de la Universidad Rey Juan Carlos, y utilizará los mismos interfaces desarrollados en este proyecto. De esta manera se podrá usar el robot Nao de Gazebo con las herramientas creadas en este proyecto, desarrolladas para Webots, sin necesidad de realizar ningún cambio.

Gazebo es, posiblemente, uno de los simuladores de robots más completos que existen actualmente ya que nos permite no sólo usar robots que ya están incorporados en el simulador como son el Pioneer2DX sino que también nos permite la creación de forma bastante intuitiva tanto de nuevos mundos, como objetos y obstáculos.

En JDErobot ya hay incorporado un driver para Gazebo que nos permite el acceso a diversos componentes de robots simulados como, en el caso de Pioneer, a la cámara, los motores, el cuello, etc.

Con el fin de trabajar siempre orientados hacia la RoboCup (entorno que utilizaremos en webots) hemos creado un mundo para Gazebo (figura 3.6) siguiendo las especificaciones del campo, mostradas en la figura 3.7, según el reglamento de la

[4]http://playerstage.sourceforge.net/gazebo/gazebo.html
[5]http://JDErobot.org/index.php/Jbermejo-pfc-itis

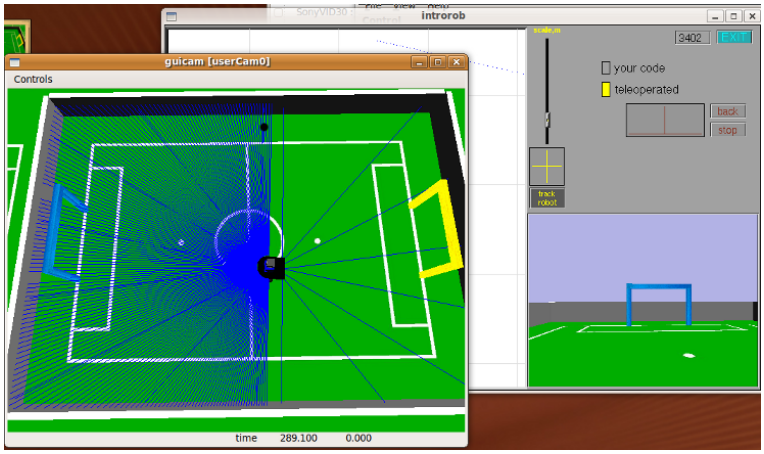


Figura 3.6: Mundo RoboCup en Gazebo

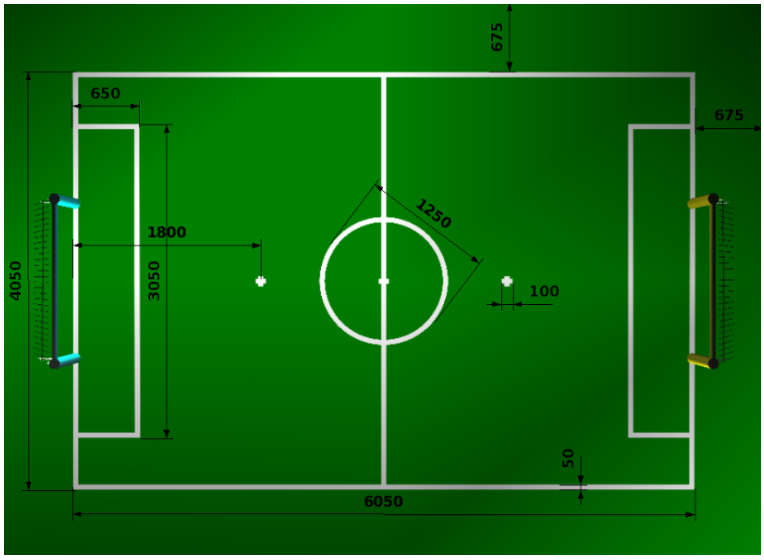


Figura 3.7: Especificaciones del campo en la RoboCup

RoboCup [6].

3.6. Glade y librerias GTK

Glade (Glade Interface Designer o Diseñador de Interfaz Glade) es una herramienta de desarrollo visual de interfaces gráficas a través de GTK/GNOME. Este interfaz es totalmente independiente de lenguaje de programación utilizado para el desarrollo del proyecto. Todo el interfaz es generado en un fichero xml donde son almacenados todos los elementos del interfaz. Estos archivos pueden emplearse para construir el interfaz en tiempo de ejecución mediante una biblioteca llamada libglade. Un pequeño ejemplo del formato de estos ficheros es el mostrado en el código 3.3 que es simplemente una ventana con un botón.

[6] http://www.tzi.de/humanoid/pub/Website/Downloads/HumanoidLeagueRules2009.pdf

```
<interface>
<requires lib="gtk+" version="2.16"/>
<!-- interface-naming-policy project-wide -->

<object class="GtkWindow" id="window1">

<child>

<object class="GtkButton" id="button1">
<property name="label" translatable="yes">button</property>
<property name="visible">True</property>
<property name="can_focus">True</property>
<property name="receives_default">True</property>
</object>
</child>
</object>
</interface>
```

Listing 3.3: Fichero xml con un interfaz de FTK

Algunas herramientas de Glade si que permiten la generación directa de código en la que solo tenemos que rellenar la función de cada uno de los elementos para generar un interfaz interactivo aunque este sistema es desaconsejado y descontinuado.

GTK+ (The GIMP Toolkit) es un conjunto de bibliotecas multiplataformas que son utilizadas por Glade para la creación de interfaces y está orientado principalmente para los entornos gráficos GNOME, XFCE y ROX aunque también se están abriendo paso en el escritorio de Windows, MacOS y otros.

En este proyecto lo utilizaremos también para asignar eventos a cada uno de los elementos del interfaz. GTK+ esta diseñado para programar con C, C++, C#, Java, Ruby, Perl, PHP o Python por lo que no tendremos ningún problema al utilizar esta librería, en concreto, el API de C.

GTK+ está bastante extendido sobre todo en el mundo Linux, herramientas como Nero-Linux, Firefox o Evolution utilizan esta librería para generar sus interfaces gráficas.

En este proyecto hemos utilizado GTK+ para la creación de todos los interfaces gráficos usados por JDErobot para la iteración con el usuario. Aunque todos estos interfaces se podrían haber escrito directamente en xml nosotros nos hemos decantado por apoyarnos en Glade, un programa con un interfaz gráfico muy intuitivo que simplifica el desarrollo de interfaces gráficos.

3.7. OpenGL

OpenGL (Open Graphics Library) es una especificación estándar que define una API multilenguaje y multiplataforma para escribir aplicaciones que producen gráficos

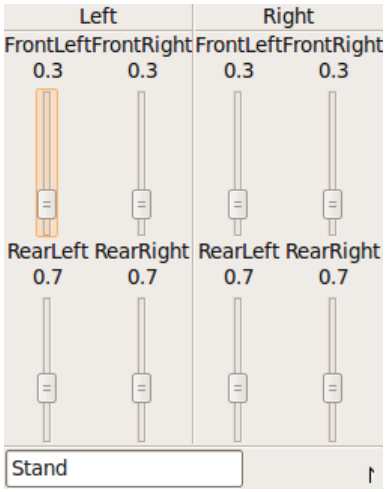


Figura 3.8: Interfaz gráfico de unos de los componentes del NaoOperator utilizando GTK y Glade

en 2D y 3D. La interfaz consiste en más de 250 funciones diferentes que pueden usarse para dibujar escenas tridimensionales complejas a partir de primitivas geométricas simples, tales como puntos, líneas y triángulos. Fue desarrollada originalmente por Silicon Graphics [7] en 1992 y se usa ampliamente en CAD, realidad virtual, representación científica, visualización de información y simulaciones de vuelo, también está muy extendido en el desarrollo de videojuegos.

En este proyecto se ha utilizado el API de C que nos ofrece OpenGL para el Visor en 3D del campo de la RoboCup con el objetivo principal de realizar un seguimiento en 3D de la posición del robot dentro del simulador así como la representación gráfica del sonar del robot.

[7] http://www.sgi.com/products/software/opengl/?/overview.html

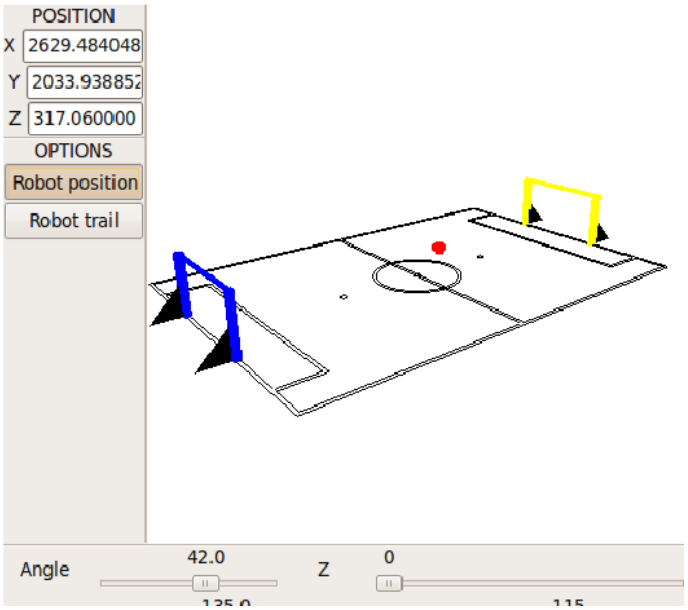


Figura 3.9: Visor en 3D del NaoOperator desarrollado utilizando OpenGL

Capítulo 4
Soporte en JDErobot para el humanoide Nao

Una vez explicado el contexto y los objetivos del proyecto, en este capítulo vamos a explicar todas las nuevas funcionalidades que hemos ido añadiendo a JDErobot y todas las aplicaciones desarrolladas en este proyecto. En primer lugar tenemos que incluir en JDErobot todo lo necesario para dar soporte a un robot humanoide como el robot Nao, nuevos sensores, nuevas articulaciones... De esta forma conseguiremos que las aplicaciones se conecten con el robot y puedan leer los datos sensoriales y ordenar movimientos a los actuadores y por consiguiente nuevas aplicaciones que se desarrollen en un futuro podrán utilizar estas herramientas que nos dan acceso a todos los componentes del robot Nao.

En este proyecto hemos desarrollado dos aplicaciones, una para teleoperar el robot desde un ordenador y otra para su control desde dispositivos móviles basados en iOS.

4.1. Diseño Global

El diseño global de las aplicaciones de este proyecto, al estar basadas en JDErobot, ha sido bastante sencillo. Por una parte tenemos un *driver* con una serie de módulos que funcionan de forma independiente, estos módulos se conectan directamente o al robot real o al robot simulado. Los *drivers* obtienen información de los diferentes componentes del robot y la ofrece a JDErobot en forma de interfaz, dondese pueden conectar aplicaciones propias de JDErobot como esquemas o aplicaciones externas. Como también hemos desarrollado un componente para conexiones de red también se pueden conectar a este entorno aplicaciones remotas de red, aplicaciones en dispositivos móviles... Todo este diseño lo podemos ver de forma gráfica en la figura 4.1.

Basándonos en la arquitectura del Teleoperator [1] hemos desarrollado un nuevo teleoperador que incorpora todas las funcionalidades de este esquema (movimiento del cuello, captura de imágenes, control de motores para locomoción) por lo que se podrá utilizar con todos los robots que utilicen el mismo formato pero a su vez se pueden activar todas las funcionalidades de un robot humanoide como el Nao. Debido a esta compatibilidad todas las aplicaciones que utilicen el Teleoperator pueden usar el nuevo teleoperador sin problemas y podrá ser utilizado en futuros proyectos tanto

[1] http://jderobot.org/index.php/Manual#Teleoperator

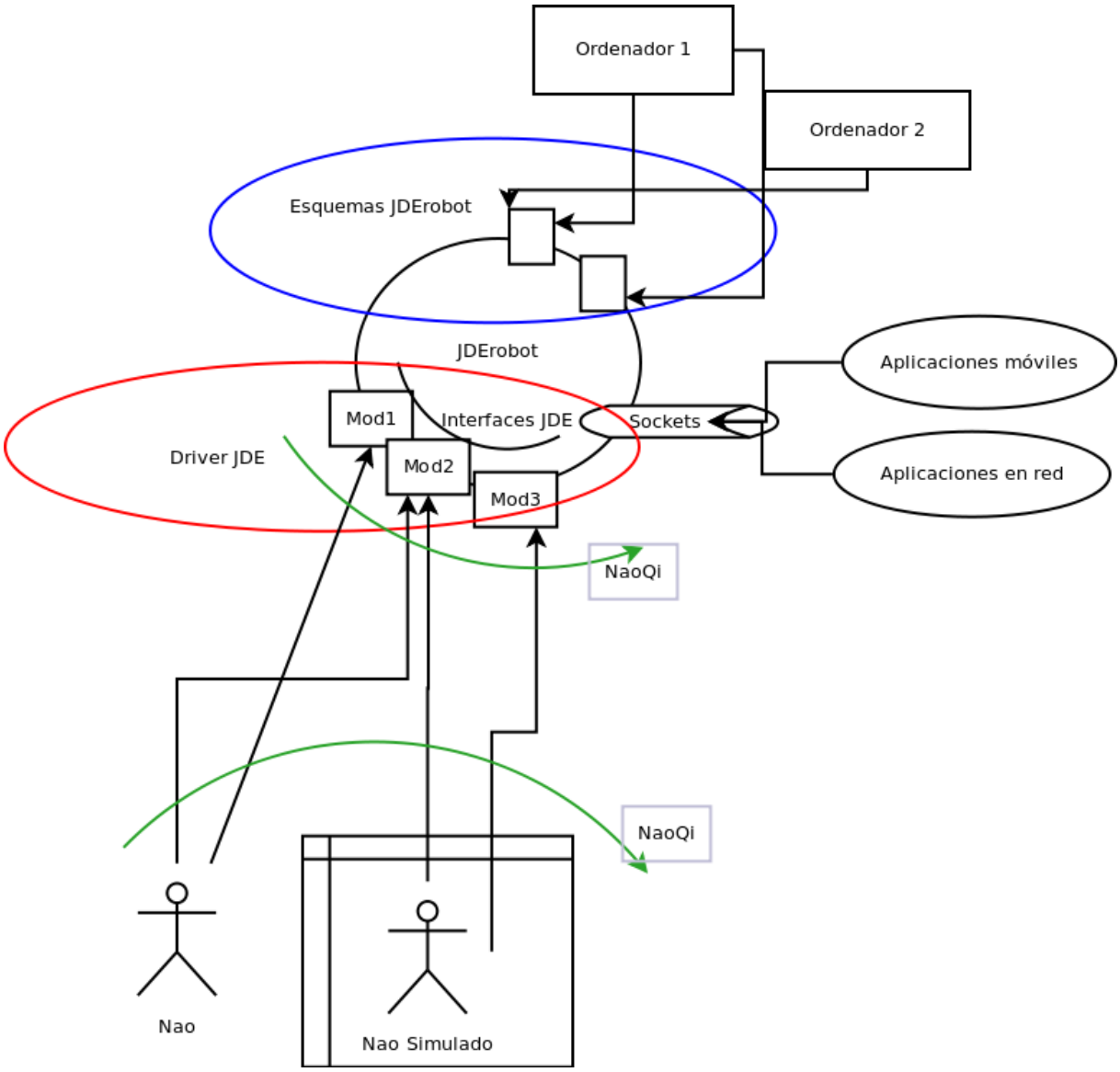


Figura 4.1: Esquema del proyecto sobre la arquitectura JDErobot

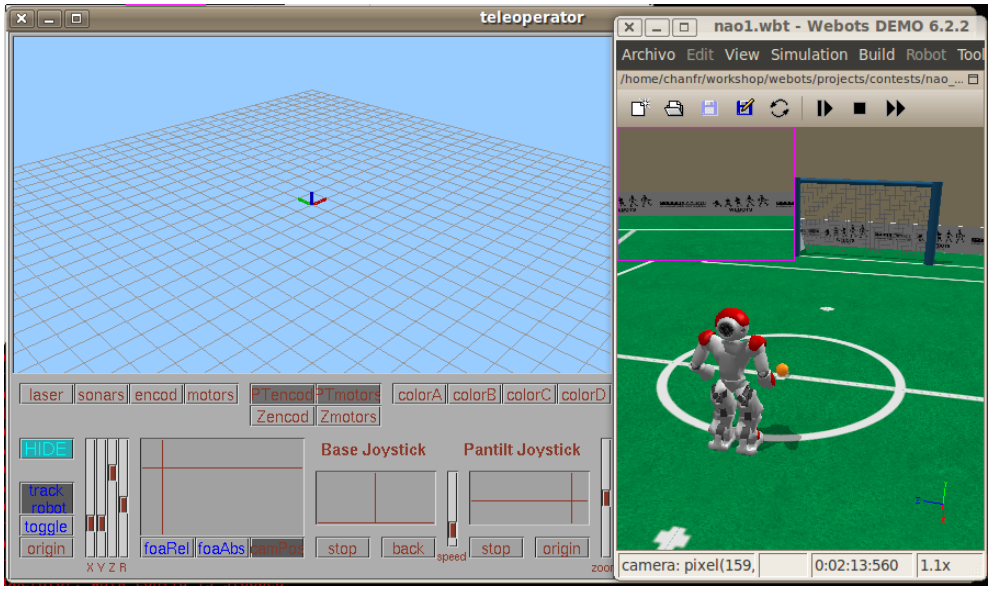


Figura 4.2: Esquema Teleoprator de JDErobot

como teleoperador de robot humanoides como de robots con ruedas.

Un interfaz dentro del entorno de JDErobot es un tipo de dato necesario para la comunicación entre el *driver* y el esquema. En el interfaz están definidos todos los datos necesarios para que esta comunicación sea lo suficientemente completa y eficiente. Al ser el primer robot de tipo humanoide en en este entorno hemos tenido que crear todos los interfaces prácticamente desde cero. Estos tipos de datos han sido encapsulados en una librería escrita en C llamada NaoTypes.

Para la implementación de las articulaciones del robot, hemos dividido al Nao en 5 partes, como podemos ver en la figura 4.3, pierna izquierda, pierna derecha, brazo izquierdo, brazo derecho y cabeza. Para todas las partes hemos tenido que desarrollar nuevos interfaces menos para la cabeza ya que puede ser utilizada como un cuello mecánico estándar y para este tipo de actuadores ya había un interfaz predefinido, *pantilt*. Lo que conseguimos al utilizar este interfaz para el cuello es, no sólo ahorrarnos la implementación de un nuevo interfaz, sino que podremos utilizar el cuello del Nao con todos los esquemas ya existenes en JDErobot que utilicen este interfaz como, por ejemplo, el teleoperator (figura 4.2). No sólo ha sido necesario el desarrollo de nuevos interfaces para la locomoción del robot sino para cada uno de los componentes del robot: sensor de presión, sensor inercial, configuración de la cámara, etc. Éstos interfaces así como los detalles de cada uno de los módulos que componen el *driver* serán explicados en el siguiente punto.

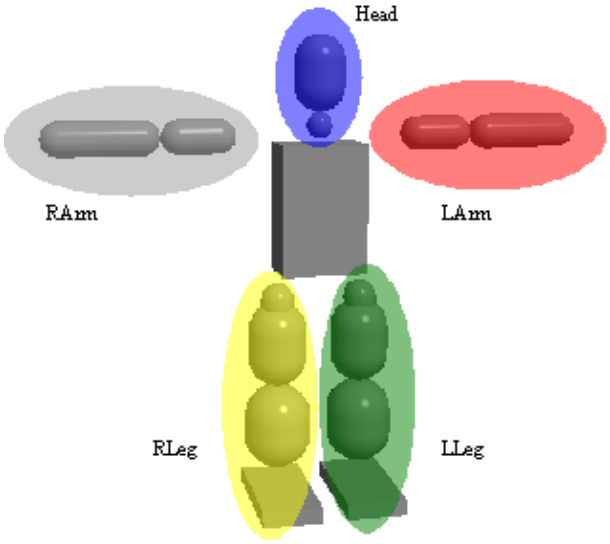


Figura 4.3: División de los actuadores del Nao

4.2. NaoBody

NaoBody es el nombre que le hemos dado al *driver* de JDErobot encargado de conectarse con el robot. Al estar basado sobre NaoQi funciona tanto con el robot real como con cualquier robot simulado compatible con este middleware, como por ejemplo Webots. Este *driver* se encarga, básicamente, de capturar las señales de los distintos sensores del robot y de mandar órdenes a cada uno de los componentes del mismo. NaoBody es bastante completo y soporta todos los componentes del robot real, así como alguna funcionalidad extra para los simuladores como la captura de la posición del robot necesaria para algunas comprobaciones como el reinicio del propio simulador, la posición del robot, etc. Siguiendo la arquitectura de los *driver* en JDErobot, NaoBody está separado en diferentes módulos que pueden ser activados cada uno de forma independiente.

Este componente ha sido desarrollado de manera conjunta con Eduardo Perdices [2] quien se encargó de la creación de un *script* para la compilación y enlzado de JDErobot y NaoQi así como la incorporación al *driver* la cámara, el cuello del robot y un módulo de odometría.

NaoBody está dividido en varios módulos o *drivers* virtuales, los cuales pueden ser cargados y activados de forma totalmente independiente (figura 4.5). Cada uno de estos módulos necesita de un interfaz para poder comunicarse con otros componentes de JDErobot o directamente con aplicaciones externas. El *driver* se encarga de exportar cada una de las variables o interfaces que necesite el módulo, podemos ver un ejemplo en el código 4.1

[2]http://jderobot.org/index.php/Eperdices_PFC

```
myexport((char*)"motors",(char*)"id",&motion_schema_id);
// exporta la variable motion_schema_id al módulo motors
```

Listing 4.1: Ejemplo de exportación de una variable desde un *driver*

JDErobot	NaoQi
varcolor	ALVideoDevice
camera_configuration	ALVideoDevice→SetParam*
ptmotors	ALMotionProxy
ptencoders	ALMotionProxy
left_arm	ALMotionProxy
right_arm	ALMotionProxy
left_leg	ALMotionProxy
right_leg	ALMotionProxy
motors	ALMotionProxy
walk_config	ALMotionProxy→setWalkConfig*
movement	ALMotionProxy
synthesizer	ALTextToSpeechProxy
leds	ALLedsProxy
sonar	ALSonarProxy
gtlocation	Gestionado por el *driver* conectado directamente con simulador utilizando librería arpa
simulator_restart	Gestionado por el *driver* conectado directamente con simulador utilizando librería arpa

Figura 4.4: Tabla con las relaciones entre interfaces de JDErobot y NaoQi

Los interfaces marcados con * no utilizan un tipo de dato definido en NaoQi sino que simplemente se utiliza una función con varios parámetros. Para facilitar el trabajo hemos simplificado esta función englobando todos los parámetros en un solo tipo de dato.

En la tabla de la figura 4.2 se puede ver la relación entre los interfaces que usa JDErobot con los de NaoQi.

4.2.1. Módulos básicos

- **Imágenes de la cámara**
  Para la obtención de imágenes del robot hemos usado otro interfaz estándar que es VARCOLOR (código 4.2). El *driver* es capaz, mediante este interfaz, de servir las imágenes que obtiene el robot. Para ampliar la funcionalidad del robot y, para poder usar las imágenes que nos ofrece el *driver* desde varios esquemas de JDErobot, NaoBody nos da la posibilidad de usar VARCOLOR desde A hasta F.
  Estos dos interfaces (ptmotors y VARCOLOR) junto con el básico de locomoción (*Motors*) son los interfaces ya desarrollados en JDErobot que hemos utilizado, sin embargo también hemos tenido que crear unos interfaces nuevos para nuestro robot ya que, hasta el momento, no se había incorporado ningún *driver* para

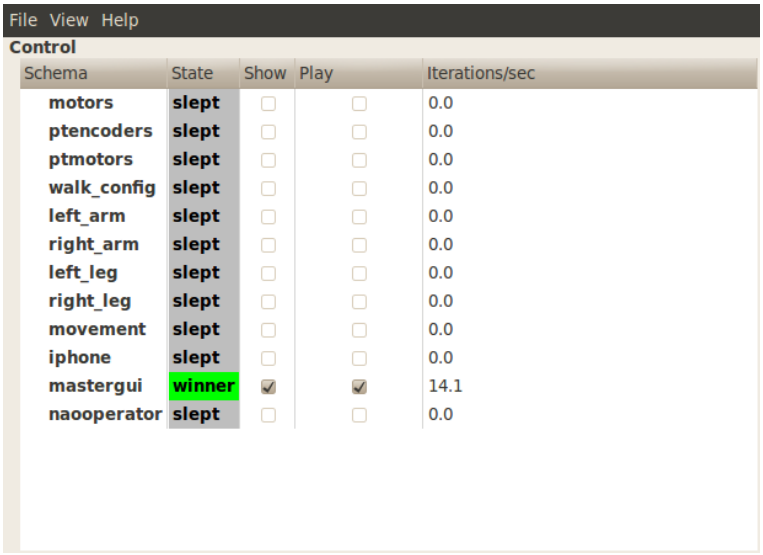


Figura 4.5: Pantallazo del JDErobot con varios módulos de NaoBody cargados

robots bípedos. Hemos intentado hacer unos interfaces, un tanto estándares, para que puedan ser reutilizados en un futuro para cualquier otro robot de las mismas características.

```
typedef struct {
    char *img;
    /* RGB order */
    unsigned long int clock;
    int width;
    int height;
} Varcolor;
```

Listing 4.2: Interfaz para la obtención de imágenes

- **Configuración de la cámara.**
  Utilizando el NaoBody no sólo tenemos la posibilidad de capturar imágenes desde la cámara del Nao sino que también podemos configurar todos los parámetros que nos permite el fabricante del robot de forma sencilla. Los parámetros que podemos fijar a través de este módulo de *Configuración de la cámara* son:

  - Brillo.
  - Contraste.
  - Saturación.
  - Matiz.
  - Croma rojo.
  - Croma azul.
  - Ganancia.
  - Volteado vertical.

- Volteado horizontal.
- Exposición automática.
- Ganancia automática.
- Resolución.
- Número de *frames* por segundo.
- Elección de cámara (superior o inferior).
- Cambio rápido de cámaras.
- Nitidez.
- Hacer un *reset* de la cámara

El interfaz utilizado para este módulo se puede apreciar en el código 4.3: Cada uno de los parámetros tiene asignado un valor entero y un *flag* de cambio,

```
typedef struct T_cam_value{
    int value;
    int change;
}T_cam_value;

typedef struct T_cam_nao{
    T_cam_value brightness;
    T_cam_value contrast;
    T_cam_value saturation;
    T_cam_value hue;
    T_cam_value red_chroma;
    T_cam_value blue_chroma;
    T_cam_value gain;
    T_cam_value horizontal_flip;
    T_cam_value vertical_flip;
    T_cam_value auto_exposition;
    T_cam_value auto_white_balance;
    T_cam_value auto_gain;
    T_cam_value camera_resolution;
    T_cam_value frames;
    T_cam_value camera;
    T_cam_value fast_switch;
    T_cam_value sharpness;
    int reset;
} T_cam_nao;
```

Listing 4.3: Interfaz para la configuración de la cámara

por lo que cada vez que se produce algún cambio en un parámetro el *flag* de cambio debe ser activado a 1 y el *driver* se encargará de fijar su nuevo valor en la cámara.

- **Cuello del robot**
  Para seguir el estándar de cuellos mecánicos existentes hasta la fecha en JDErobot hemos implementado el cuello del robot como *PanTilt-motors* y *PanTilt-encoders*. La parte de *motors* se encarga de mandar señales a los actuadores y la parte de *encoders* de recibir información. Estos módulos nos permiten mover la cabeza del robot y obtener la posición en la que se encuentra en todo momento. Los nombres

de estos dos módulos son *ptmotors* y *ptencoders*. En este caso no se utiliza un tipo de dato que englobe todos los valores del interfaz, sino que por el contrario, se utilizan variables independientes para cada uno de los valores proporcionados, como podemos ver en el código 4.4.

```
myexport((char*)"ptencoders",(char*)"pan_angle",&yreal);
myexport((char*)"ptencoders",(char*)"tilt_angle",&preal);
myexport((char*)"ptencoders",(char*)"clock",&my_clock);


myexport((char*)"ptmotors",(char*)"cycle",&cycle);
myexport((char*)"ptmotors",(char*)"longitude",&ynao);
myexport((char*)"ptmotors",(char*)"latitude",&pnao);
myexport((char*)"ptmotors",(char*)"max_longitude",&max_longitude);
myexport((char*)"ptmotors",(char*)"max_latitude",&max_latitude);
myexport((char*)"ptmotors",(char*)"min_longitude",&min_longitude);
myexport((char*)"ptmotors",(char*)"min_latitude",&min_latitude);
myexport((char*)"ptmotors",(char*)"longitude_speed",&vy);
myexport((char*)"ptmotors",(char*)"latitude_speed",&vp);
myexport((char*)"ptmotors",(char*)"max_longitud_speed",&max_y_speed);
myexport((char*)"ptmotors",(char*)"max_latitud_speed",&max_p_speed);
myexport((char*)"ptmotors",(char*)"odometry",odometry);
myexport((char*)"ptmotors",(char*)"head_st",&head_st);
```

Listing 4.4: Interfaz para uso del cuello mecánico del robot

- **Articulaciones del robot.**
  Cada una de las articulaciones del robot pueden tener 3 actuadores: *pitch*, *roll* y *yaw*. NaoBody nos ofrece un control básico de cada uno de los actuadores de las articulaciones de manera independiente y simultánea excepto de uno: el *yaw* de la cadera. Tanto el *yaw* de la cadera izquierda como el de la derecha deben moverse simultáneamente puesto que utilizan un solo actuador para ambas articulaciones. Para cada motor tenemos tanto el valor al que queremos mover el actuador como el valor de la posición en la que se encuentra, todos estos valores del NaoBody, en lo que a actuadores se refiere, están en grados.

  El interfaz de un motor es el mostrado en el código 4.5, dentro de este interfaz tenemos una variable *clock* que hace de contador y se incrementa cada vez que alguno de los valores de articulación varia. Este *flag* se utiliza para saber si el último valor de la posición que hemos leído sigue actualizado y debemos volver a leerlo.

  Todas las articulaciones así como los diferentes actuadores de los que dispone cada una se pueden ver en la figura **??**. La única diferencia es que el Nao utilizado en este proyecto es la versión para educación que no dispone de actuadores ni en las muñecas ni en las manos.

- **Motors**
  Este módulo nos permite mover al robot usando una velocidad v y una velocidad

```
typedef struct T_motor{
    float pitch;
    float roll;
    float yaw;
    float enc_pitch;
    float enc_roll;
    float enc_yaw;
    unsigned long int clock;
}T_motor;
```

Listing 4.5: Interfaz de articulaciones

angular w. Es uno de los módulos más complejos de NaoBody ya que trata de simplificar la complejidad de una forma de andar de un robot bípedo en dos valores. Para simplificar este problema y centrarnos en mejorar la forma de andar hemos utilizado las locomoción ofrecida por el fabricante. Las funciones que utilizamos son para movimientos en linea recta (w=0) y para simular una velocidad angular hemos utilizado una forma de andar curva. Este tipo de interfaz también es estándar de JDErobot y nos permite reducir toda la locomoción de un robot a dos parámetros: velocidad lineal y velocidad angular.

- **Configuración de la forma de andar**
  NaoQi no sólo nos da unas forma de andar predefinidas para el robot sino que también nos permite hacer pequeñas configuraciones.

  Los parámetros que podemos modificar de la forma de andar del fabricante son los siguientes:

  - MaxStepLength: longitud máxima de una zancada.
  - MaxStepHeight: altitud máxima en todo el ciclo del paso.
  - MaxStepSide: valor máximo del ancho de cadera en el robot durante el paso.
  - MaxStepTurn: cambio máximo (en radianes) en la orientación del robot.
  - HipHeight: Altura de la cadera.
  - TorsoYOrientation: define la orientación del torso en el eje Y durante el movimiento (Verticalidad del robot).

  El interfaz ofrecido para este fin es el mostrado en el código 4.6:

- **Leds**. NaoBody nos permite usar los leds del robot tanto individualmente como por grupos y también realizar una serie de secuencias con los leds previamente establecidas por el fabricante como, por ejemplo, que los leds de las orejas emulen una rueda. Este módulo es muy útil a la hora de depurar código usando el robot real.

```
typedef struct T_walk_config
{
    float pMaxStepLength;
    float pMaxStepHeight;
    float pMaxStepSide;
    float pMaxStepTurn;
    float pHipHeight;
    float pTorsoYOrientation;
}T_walk_config;
```

Listing 4.6: Interfaz para la configuración de la cámara

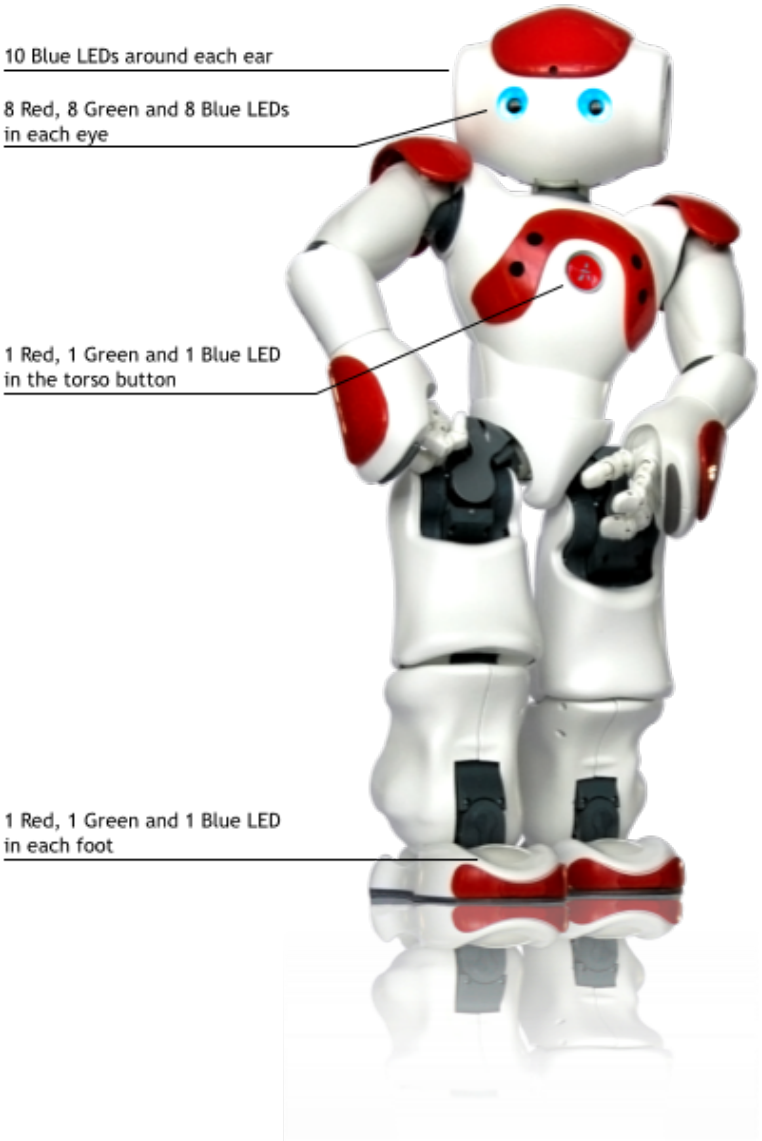


Figura 4.6: Localización de los leds en el robot Nao

El robot Nao de Aldebaran dispone de 77 leds localizados como se explica en la figura 4.6. Debido al gran número de componentes leds, la estructura definida para el interfaz de los leds es amplia pero bastante sencilla ya que sólo necesitaremos un entero por led para saber si está encendido o apagado. Podemos ver el interfaz desarrollado para este módulo en el código (4.7), donde las últimas siete variables son utilizadas para realizar animaciones definidas por el fabricante: Do_angle, do_rasta. do_rotate y do_random son los activadores de dichas animaciones.

```
typedef struct T_leds_nao{
    int left_ear[10];
    int left_ear_change;
    int right_ear[10];
    int right_ear_change;
    int left_eye_blue[8];
    int left_eye_blue_change;
    int left_eye_red[8];
    int left_eye_red_change;
    int left_eye_green[8];
    int left_eye_green_change;
    int right_eye_blue[8];
    int right_eye_blue_change;
    int right_eye_red[8];
    int right_eye_red_change;
    int right_eye_green[8];
    int right_eye_green_change;
    int torso_red;
    int torso_blue;
    int torso_green;
    int torso_change;
    int left_foot_red;
    int left_foot_blue;
    int left_foot_green;
    int left_foot_change;
    int right_foot_red;
    int right_foot_blue;
    int right_foot_green;
    int right_foot_change;
    float duration;
    float time_turn;
    int angle;
    int do_angle;
    int do_rasta;
    int do_rotate;
    int do_random;
} T_leds_nao;
```

Listing 4.7: Interfaz de los leds

- **Sonar**. El módulo Sonar lo que nos permite es leer los valores obtenidos por los sensores de ultrasonidos que están incorporados en el robot. Las adquisiciones del sonar nos devuelven dos valores, la de la parte izquierda y la de la parte derecha, ambos valores en metros. La disposición de los sensores de ultrasonidos en el robot Nao se puede apreciar en la figura 4.7. El interfaz para el sonar es bastante sencillo ya que como hemos explicado anteriormente sólo tenemos dos valores, la lectura de la parte izquierda y la lectura de la parte derecha, podemos ver su estructura en el código 4.8.

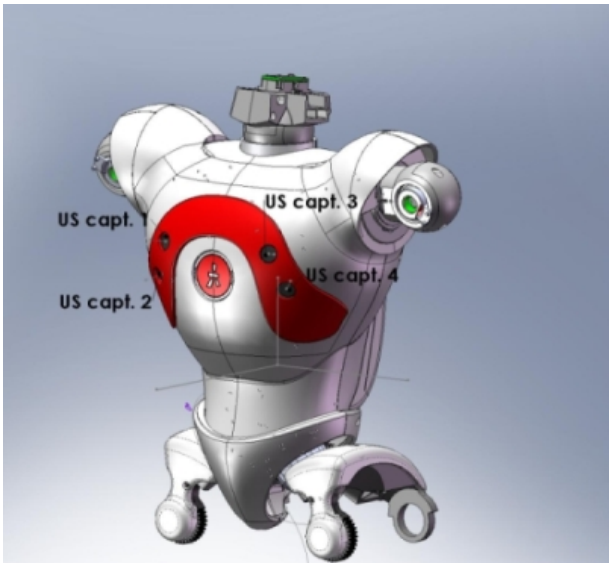


Figura 4.7: Localización de los sensores de ultrasonido.

```
typedef struct T_sonar_nao{
    float left;
    float right;
}T_sonar_nao;
```

Listing 4.8: Interfaz del sonar

- **Sintetizador**. Usando este módulo del NaoBody tenemos acceso al sintetizador de voz del robot Nao mediante el cual podemos hacer que el robot reproduzca un texto. En este módulo también tenemos una serie de valores configurables:
  - Tipo de voz.
  - Volumen de la voz.
  - Velocidad de reproducción.
  - Tono

  En la figura 4.8 se puede ver el funcionamiento del sintetizador de voz [3]. El interfaz utilizado por este módulo lo podemos ver en el código 4.9.

- **Sensores de presión** El robot Nao dispone de 4 sensores de presión en cada pie situados como indica la figura 4.9. Estos sensores tienen soporte completo con este módulo y son útiles a la hora de analizar movimientos, estabilidad y otros parámetros. El interfaz utilizado en este módulo es el mostrado en el código 4.10

[3] TTS: Text-To-Speech
API: Application Programming Interface
ALSA: Advanced Linux Sound Architecture
FX: Sound Effect

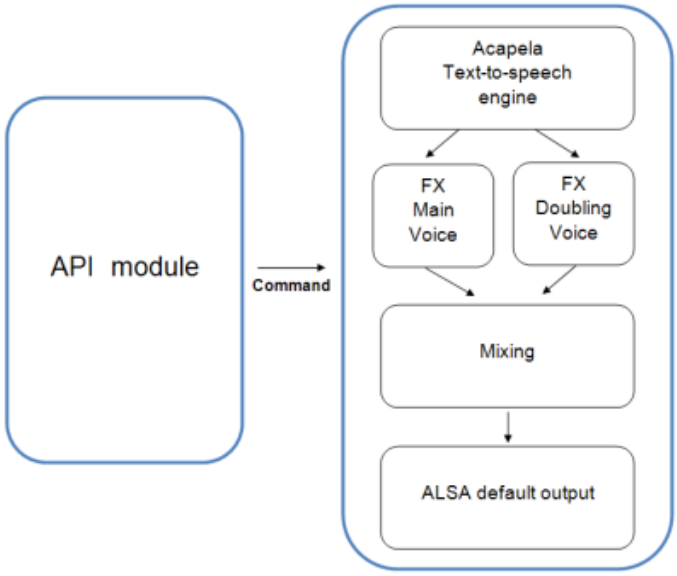


Figura 4.8: Sistema de Audio del Nao

```
typedef struct T_voice_nao{
    float doublevoice;
    float doublevoicelevel;
    float doublevoicetimeshift;
    float pitchshift;
    char text[255];
    int say;
} T_voice_nao;
```

Listing 4.9: Interfaz del sintetizador de voz

```
typedef struct T_foot_fsr{
    float front_left;
    float front_right;
    float rear_left;
    float rear_right;
}T_foot_fsr;

typedef struct T_fsr{
    T_foot_fsr left;
    T_foot_fsr right;
}T_fsr;
```

Listing 4.10: Interfaz del sensor de presión

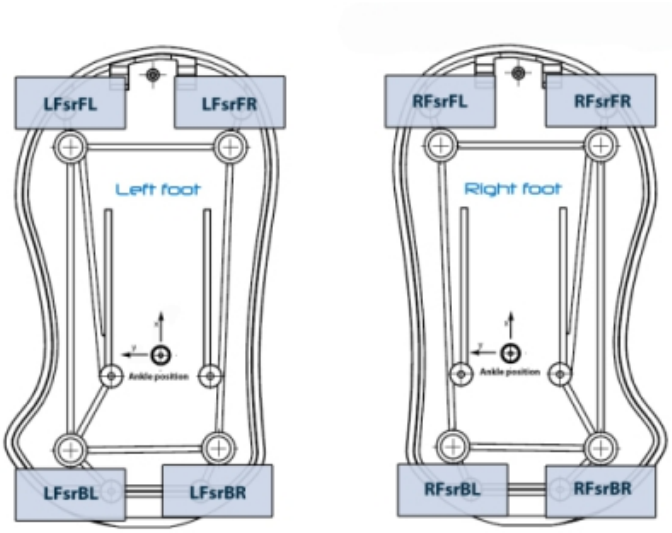


Figura 4.9: Sensores de presión del robot Nao

4.2.2. Módulos elaborados

- **Movimientos fijos.** Con este módulo podemos recrear movimientos propios no definidos por el fabricante. Una vez cargado un movimiento, es decir, una serie de valores para cada uno de los motores que intervengan en el movimiento y un valor de tiempo asignado a cada posición fijado desde la posición inicial (figura 4.10), éste modulo es el encargado de que el robot haga los movimientos indicados.

  Todos los movimientos generados y predefinidos de este proyecto están basados en **fotogramas**. La idea es generar posiciones mediante el acceso a los diferentes actuadores del robot e ir guardando estos movimientos en forma de secuencia. Si a cada una de estas posiciones le asignamos un valor en el tiempo y somos capaces de recrearlo, tendremos nuestra propia secuencia fija.

  Centrándonos en esta idea, generar movimientos es bastante sencillo incluso a la hora de la implementación, ya que NaoQi nos simplifica mucho el uso de este tipo de movimientos encargándose él mismo de la interpolación entre los distintos fotogramas.

  Éste módulo también tiene cargados tres movimientos básicos para manejar el robot:

  - Posición inicial: muy útil cuando queremos realizar alguna comprobación siempre desde la misma posición.
  - GetUpBack: movimiento necesario para que el robot se levante cuando se encuentre en una posición decúbito supino.

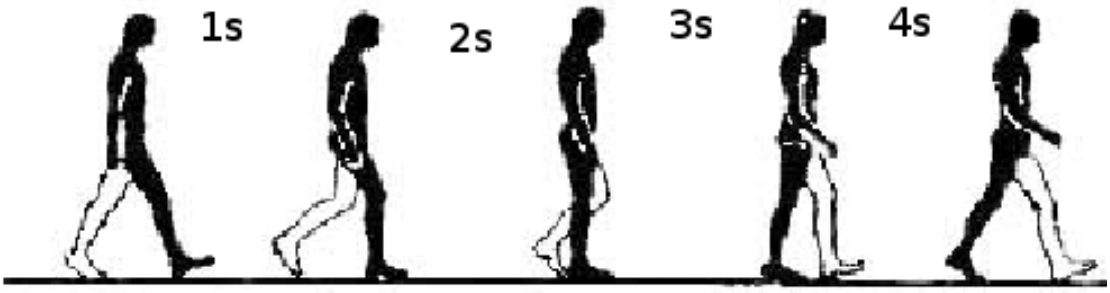


Figura 4.10: Secuencia de un movimiento

- GetUpFront: movimiento necesario para que el robot se levante cuando se encuentre en una posición decúbito prono.

Con estos movimientos y el sensor inercial hemos desarrollado un comportamiento para el robot Nao con el fin de levantarse automáticamente. El robot funciona de forma normal: anda, realiza movimientos... hasta que el robot se cae, en este momento, el *driver* detecta que el Nao se ha caído y pone en marcha el comportamiento que supone suprimir todos los movimientos acolados en el robot de tal forma que bloquea la recepción de nuevos movimientos hasta que el robot no se encuentre de pie. Este comportamiento puede ser activado mediante el interfaz de este módulo, lo cual es muy útil tanto en el robot simulado como en el real durante la realización de pruebas dinámicas o incluso en el transcurso de un partido.

Los interfaces utilizados para este módulo son varios:

- Estructura para las posiciones fijas, mostrado en el código 4.11
  Como ya habíamos explicado anteriormente tiene un valor guardado para

```
typedef struct T_position{
    T_motor left_shoulder;
    T_motor left_elbow;
    T_motor right_shoulder;
    T_motor right_elbow;
    T_motor left_hip;
    T_motor left_knee;
    T_motor left_ankle;
    T_motor right_hip;
    T_motor right_knee;
    T_motor right_ankle;
    float head_yaw;
    float head_pitch;
    float time;
    int saved;
}T_position;
```

Listing 4.11: Estructura para las posiciones fijas

cada motor, así como, un valor en segundos para la transición. El *flag saved*

nos sirve para saber si esa posición está realmente ocupada o la podemos sobreescribir (muy útil a la hora de editar los movimientos).

- Estructura para los movimientos, motrado en el código 4.12
  El formato elegido en un principio para guardar la secuencia de movimientos

```
typedef struct T_position_nodo T_position_nodo;

struct T_position_nodo{
    T_position pos;
    T_position_nodo* siguiente;
    T_position_nodo* anterior;
};


typedef struct T_movement{
    T_position_nodo* inicio;
    T_position_nodo* fin;
    unsigned int counter;
} T_movement;
```

Listing 4.12: Estructura para los movimientos basados en secuencias fijas

era mediante el uso de un array para posiciones fijas pero, debido a la necesidad de almacenar movimientos largos sin definir su longitud y para aumentar el número de posiciones posibles, decidimos cambiar el formato a una lista doblemente enlazada. Con esto reducimos el uso de memoria en el caso de movimientos con pocas posiciones y obtenemos la posibilidad de aumentar el número de posiciones dentro de un movimiento de manera dinámica.

Finalmente el interfaz completo que exporta el *driver* para este módulo es el mostrado en el código 4.13.

```
    myexport((char*)"movement",(char*)"initpos",&initpos);
    myexport((char*)"movement",(char*)"getupback",&getupback);
    myexport((char*)"movement",(char*)"getupfront",&getupfront);
    myexport((char*)"movement",(char*)"up_behavior",&up_behavior);
    myexport((char*)"movement",(char*)"movement_test",&test_m);
    myexport((char*)"movement",(char*)"movement",&movement);
    myexport((char*)"movement",(char*)"move",&do_move);
```

Listing 4.13: Valores exportados para los movimientos

- **Sensor inercial**. Este módulo es el encargado de darnos información sobre la posición en la que se encuentra el robot, y es utilizado, sobre todo, para saber si el robot se encuentra de pie o si se ha caído. La información del sensor inercial no está disponible para el robot simulado en Webots a través del API de NaoQi, sin embargo, si que está disponible directamente en el simulador. Para obtener esta información sin utilizar NaoQi es necesario incorporar a este módulo la opción de que, en caso de estar utilizando un robot simulado, active una nueva

herramienta de comunicación directa con el simulador, de esta manera podremos obtener información sobre la posición del robot simulado. Los valores ofrecidos por el robot real y por el robot simulado no son iguales, por lo que el *driver* se encargará de transformar esta información para que la aplicación que lo utilice no tenga que preocuparse de realizar ningún cambio y sea totalmente transparente. El interfaz utilizado para este módulo es el mostrado en el código 4.14.

```
typedef struct T_inertial{
    int falled_back;
    int falled_front;
    int change;
}T_inertial;
```

Listing 4.14: Interfaz del sensor inercial

- **Posición verdadera**. Este módulo sólo funciona con un simulador y lo que nos ofrece es la posición real del robot en 3D dentro del campo siendo capaz de crear una sombra de las posiciones anteriores en las que ha estado el robot como se puede apreciar en la figura 4.18. Este componente sólo funciona con el simulador y usa *sockets* para la comunicación *driver*-simulador.
  Dos interfaces son necesarios para este módulo, uno simple para la posición actual (código 4.15) y uno complejo para el *tracking* (código (4.16).

```
typedef struct T_com_pos{
    double x;
    double y;
    double z;
}T_com_pos;
```

Listing 4.15: Interfaz para la posición actual

```

typedef struct T_lista_pos T_lista_pos;

typedef struct T_positions_gl
{
    int counter;
    T_lista_pos* inicio;
}T_positions_gl;

struct T_lista_pos{
    T_com_pos pos
    T_lista_pos* siguiente;
};
```

Listing 4.16: Interfaz para el *tracking*

- **Reinicio del simulador**. Está conectado al simulador y al igual que la*Posición Verdadera* se comunica a través de *sockets*. La funcionalidad que nos aporta este módulo es la posibilidad de reiniciar el simulador directamente desde el *driver*: el módulo de reinicio le envía una señal al simulador para que empiece el reinicio

y espera una respuesta del simulador indicándole que el reinicio se ha realizado correctamente.

Como en este módulo sólo necesitamos una variable que funcionará como *flag* de reinicio se exporta directamente desde el *driver* como muestra el código 4.17.

```
    myexport((char*)"restart",(char*)"restart_simulator",&restart_simulator);
```

Listing 4.17: Exportación de la variable de reinicio para el simulador

4.2.3. Configuración del *driver*

NaoBody necesita de un fichero de configuración donde deberán ser fijados una serie de parámetros para el correcto funcionamiento del *driver* como la IP y el puerto del robot, si es real o simulado así como las partes que queremos activar del robot. Un ejemplo de fichero de configuración para el *driver* podría ser el mostrado en el código 4.18: en el que hemos fijado el robot como simulado pero hemos intentado cargar con *provides* componentes del robot que no estén disponibles en el simulador. Esto no genera un error ya que el *driver* se encarga de deshabilitar aquellos componentes seleccionados que no estén disponibles en el tipo de robot utilizado antes de intentar cargarlos.

```
    driver naobody
    provides motors
    provides ptmotors
    provides ptencoders
    provides walk_config
    provides left_arm
    provides right_arm
    provides left_leg
    provides right_leg
    provides movement
    provides synthesizer
    provides camera_configuration
    provides leds
    provides sonar
    provides gtlocation
    provides simulator-restart
    robot simulated 127.0.0.1 9559
    provides varcolorE 0 320 240
    provides varcolorA 0 320 240
    provides varcolorB 0 320 240
    provides varcolorF 0 320 240
    provides varcolorC 0 320 240
    fps 20
    network 192.168.1.44
end_driver
```

Listing 4.18: Configuración del *driver*

4.2.4. Implementación

La implementación de este *driver* no es del todo compleja a la hora de ejecutarse en una máquina. Básicamente lo que hace es, por una parte, tomar medidas de todos los componentes activos del robot y, por otra parte, mandar las órdenes recibidas a actuadores o dispositivos. Todas las operaciones que realiza el *driver* se hacen de forma secuencial y en una única hebra cíclica como podemos ver en la figura 4.11 y está diseñado para ejecutarse a unas 22 iteraciones por segundo.

Como el *driver* NaoBody es el encargado de convertir los datos de NaoQi a los de JDErobot hemos decidido crear una nueva librería llamada *adapter.h* escrita en C++ donde hemos definido objetos que utilizaremos como paso intermedio de conversión entre NaoQi y JDErobot. También esta librería es la encargada de la comunicación directa con NaoQi y la conexión son los *proxies* necesarios.

El hilo de ejecución de este *driver*, en el que están cargados los módulos de configuración de la cámara, la captura de imágenes, y las articulaciones, se puede apreciar en la figura 4.11.

4.2.5. Módulo de comunicaciones con dispositivos de red

El NaoBody incorpora un pequeño módulo que permite recibir órdenes mediante la utilización de las librerías *inet* y *socket* a través de la red. Básicamente es una hebra que se crea al activar este componente y que espera instrucciones en un socket con una dirección IP y un puerto que se pueden fijar desde el fichero de configuración del *driver*. Una vez recibida una instrucción el *driver* la procesa y en caso de ser una instrucción válida la ejecuta. Las instrucciones que recibe desde el socket son del tipo *mueve el motor left-arm-pitch a la posición x con velocidad y* pero no de forma redactada sino que recibe toda la información directamente desde una línea de texto con todos los parámetros necesarios para construir una instrucción. Un ejemplo real seria: *2 10* , donde el 2 corresponde con la instrucción andar y el 10 es a la velocidad que queremos que ande.

Para las comunicaciones se utiliza un formato de línea creado por nosotros en el que, en el diseño actual del módulo, se pueden recibir tres tipos de instrucciones, realizar el movimiento que tenga cargado el robot, mover un actuador o andar. El parámetro que diferencia entre estos tres tipos de instrucciones es el primero y a continuación describiremos el protocolo que se utiliza según el valor del primer argumento. Este abanico de instrucciones se puede ampliar desde este módulo de forma bastante sencilla.

- **0**: Se realiza un movimiento que se encuentre cargado en el *driver*.

Primer parámetro
Identificador del movimiento a realizar

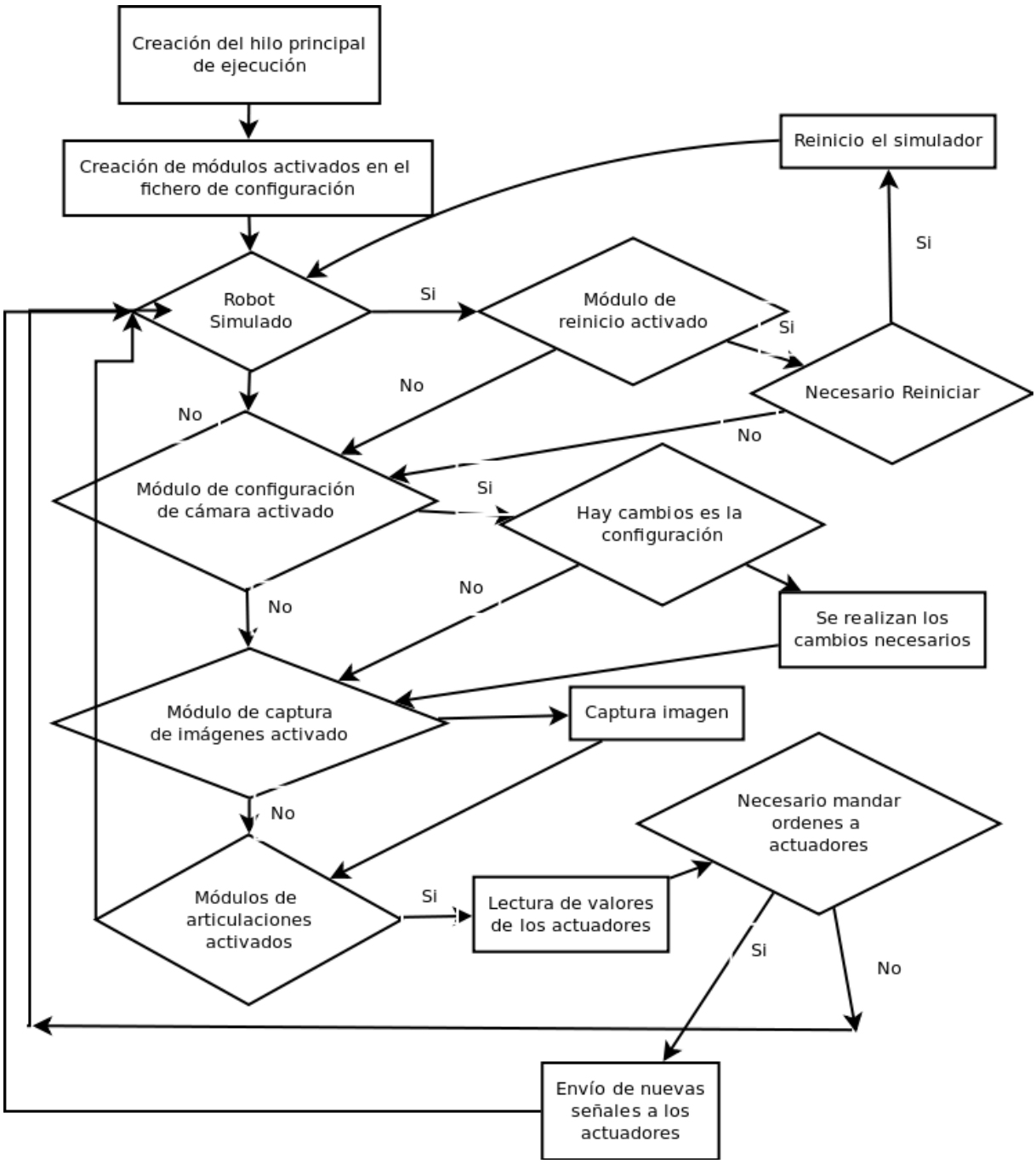


Figura 4.11: Diagrama de ejecución del *driver* NaoBody.

- **1**: Se mueve el actuador seleccionado.

Primer parámetro	Segundo parámetro	Tercer parámetro
Posicion del actuador (grados)	Izquierdo(0)/Derecho(1)	Identificador del ángulo

- **2**: Andar.

Primer parámetro
velocidad

Este módulo no sólo nos permite la utilización de dispositivos móviles, fin con el que lo hemos utilizado en este proyecto, sino que nos permite una amplia gama de posibilidades. Cualquier aplicación que se conecte a la red donde se está ejecutando el *driver* podrá comunicarse directamente con el robot a través de este módulo del *driver*. No será necesaria ninguna configuración previa con el robot ni el uso de NaoQi, sólo deberá conocer el formato de línea utilizada y conectarse al *driver* mediante *sockets*.

4.3. Aplicación NaoOperator

El NaoOperator es un esquema de JDErobot que surge de la necesidad de un teleoperador completo para el robot Nao que utilizara el *driver* NaoBody. Es capaz de controlar todos los componentes del robot Nao mediante una interfaz gráfica intuitiva y muy sencilla. Otra funcionalidad importante que nos ofrece el NaoOperator es un dominio total de los actuadores y sensores del robot. Los objetos utilizados en el interfaz son solamente una serie de visores de OpenGL, *sliders*, *radiobuttons*, *checkbuttos* y botones.

El NaoOperaor, al igual que la mayoría de los componentes de JDErobot, dispone de un fichero de configuración en el que especificaremos qué partes del robot queremos utilizar con este esquema así como una serie de parámetros que tendremos que fijar antes de arrancar el esquema como el fichero de líneas del campo, si el robot es simulado o real (recordemos que hay partes tanto de este esquema como del NaoBody que sólo funcionan en el robot real (láser, luces, etc) o que sólo funcionan en el simulador (GroundTruth)), etc.

Un ejemplo de fichero de configuración para un robot simulado es el mostrado en el código 4.19:

Al igual que ocurre con el *driver*, no se produce ningún error si intentamos cargar algún componente que no está disponible en el tipo de robot seleccionado ya que son previamente deshabilitados.

Debido a la gran funcionalidad de este esquema se han ido incorporando los diferentes módulos necesarios para la evaluación y creación de movimientos sobre él, lo que nos permite reducir el trabajo de crear un nuevo esquema con parte de la funcionalidad del NaoOperator así como enriquecer el propio NaoOperator con nuevas funcionalidades orientadas a la generación de movimientos.

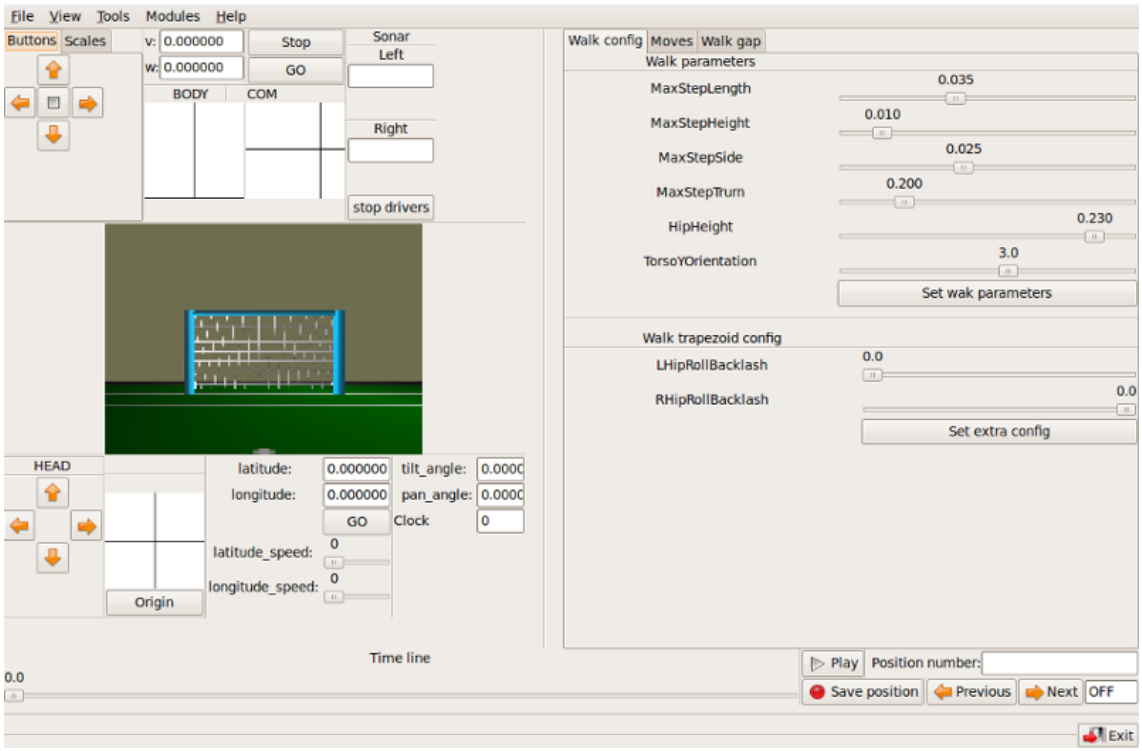


Figura 4.12: Pantallazo NaoOperator

```
esquema naooperator
uses ptmotors
uses varcolorC
uses ptencoders
uses motors
uses walk_config
uses left_arm
uses right_arm
uses left_leg
uses right_leg
uses movement
uses camera_configuration
uses synthesizer
uses leds
uses sonar
uses gtlocation
uses simulator-restart
robot simulated
worldfile campo.conf
end_schema
```

Listing 4.19: Configuración del schema

La implementación de esta aplicación, al tratarse de una aplicación de teleoperación, está basada en eventos del interfaz gráfico. Existen únicamente dos hebras, una encargada de la ejecución principal de la aplicación en la que se realizan las tareas de actualización de variables y el estado del esquema y otra hebra encargada del manejo del interfaz gráfico. La hebras mas compleja es la del interfaz gráfico ya que es la encargada de actualizar en todo momento los nuevos valores recibidos de los componentes activos del robot. La ejecución de esta hebra de control gráfico se produce de forma secuencial tal y como podemos apreciar en la figura 4.13.

Las principales funcionalidades que nos ofrece el NaoOperator son las siguientes:

4.3.1. Visión

El NaoOperator es capaz de obtener imágenes desde el robot usando el interfaz estándar de JDErobot VarColor. Con el módulo de visión del NaoOperator también podemos modificar todos los parámetros de configuración que nos ofrece el Robot como son el brillo, contraste, saturación, ganancia... También podemos modificar el tiempo de ejecución el número de *frames* por segundos que captura la cámara así como cambiar entre la cámara superior y la inferior. También se ha incorporado un botón con la opción *Fast Switch* para el cambio rápido entre las dos cámaras incorporado en las últimas versiones del API NaoQi que reduce notoriamente el tiempo que conlleva el cambio entre las cámaras. Para esta nueva funcionalidad las dos cámaras deben estar funcionando con la misma configuración.
En la figura 4.14 podemos ver el interfaz mediante el cual podemos cambiar la configuración de las cámaras.

4.3.2. Locomoción y articulaciones

NaoOperator nos permite tener un control total sobre la locomoción del robot Nao. Podemos acceder a cada uno de los actuadores independientemente y ejecutar algunos de los movimientos que nos ofrece el fabricante del robot (como las funciones para que el robot ande, tanto en línea recta como realizando un movimiento arqueado). Todos estos actuadores pueden ser controlados con una serie de *gtkRanges* o *sliders* como podemos apreciar en la figura 4.15 Este mismo interfaz nos sirve tanto para saber en qué posición se encuentra cada uno de los actuadores del Robot como para modificarla. El cambio entre modificación y sensado se produce pulsando el botón *Active sensors*. Con esta opción podemos ver la posición de los motores durante la realización de cualquier movimiento.
Para el control del cuello robot hemos incorporado un controlador de pantilt. Mediante un pequeño visor incluido en el NaoOperator podemos modificar la posición del cuello mediante eventos de ratón. Otras dos opciones que disponemos en el NaoOperator para controlar el cuello es mediante la utilización de *slider* (como en el resto de los actuadores del robot) o mediante botones que modifican los valores de los actuadores. Todas estas posibilidades las podemos apreciar en la figura 4.16.

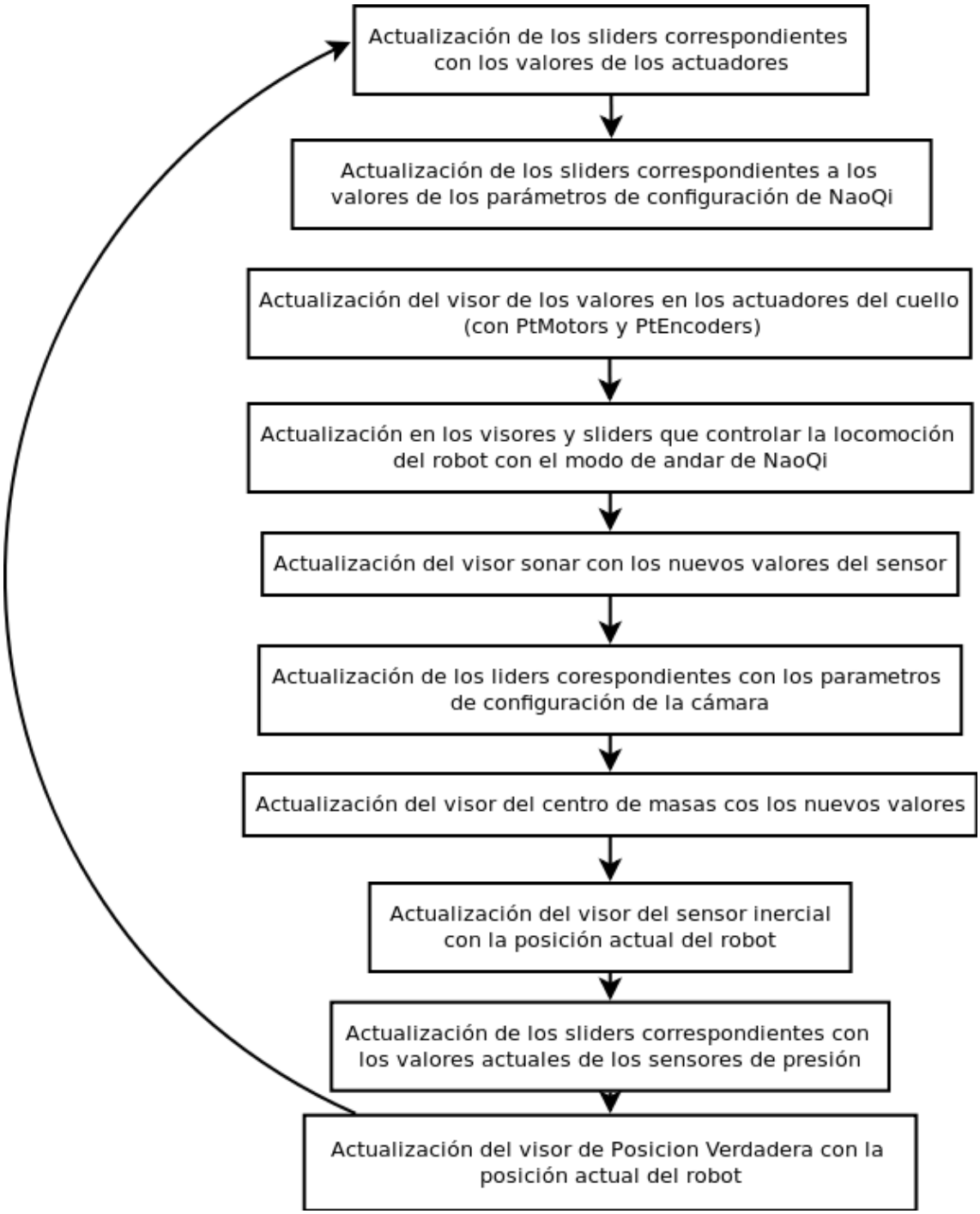


Figura 4.13: Diagrama de ejecución del esquema NaoBody

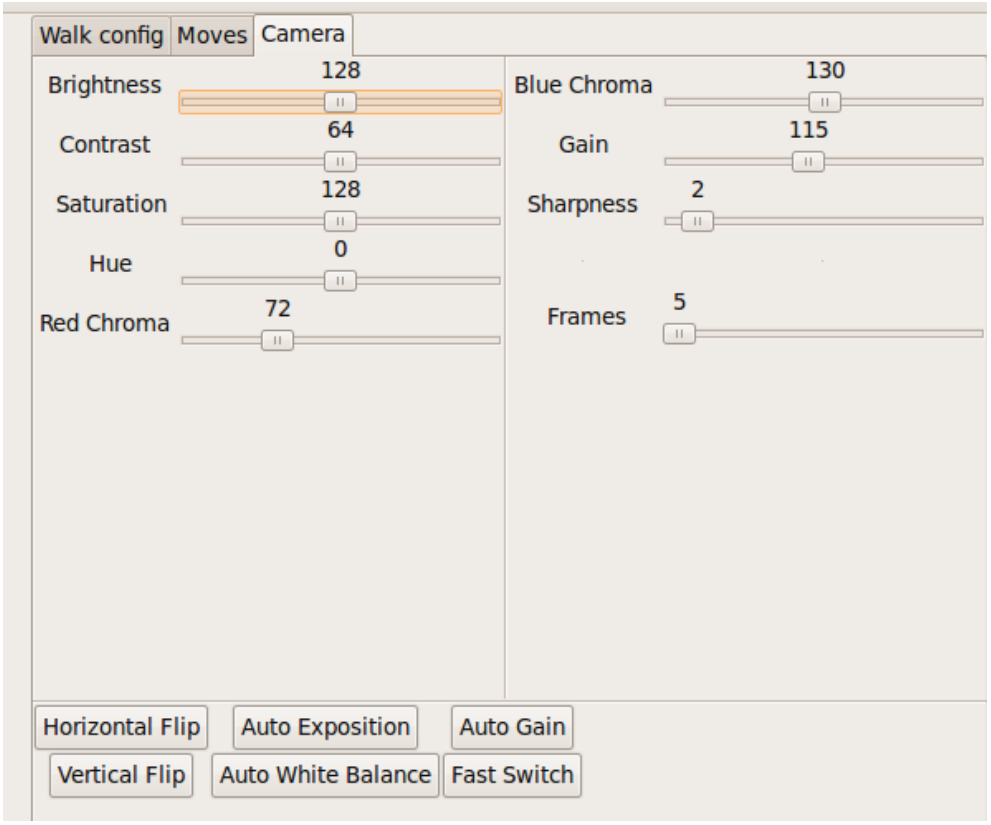


Figura 4.14: Configuración de las cámaras desde el NaoOperator

Este componente es la herramienta básica para la creación de nuevos movimientos y nos ofrece algunas opciones para facilitar esta tarea. Una de las más importantes a la hora de crear posiciones es la de poder fijar actuadores simétricos para que se muevan simultáneamente. Los *toggles* que vemos en la imagen en la parte inferior izquierda sirven justo para eso. Si activamos alguno de los motores lo que vamos a conseguir es que al desplazar el *slider* correspondiente también se mueva su actuador simétrico, es decir, si fijamos el hip-pitch y movemos el *slider* izquierdo del hip-pitch también cambiarán los valores del derecho. Solamente tienen esta opción los actuadores más conflictivos, es decir, los actuadores que al variarlos es muy probable que el robot caiga. El resto de actuadores se pueden manipular de forma independiente de manera bastante simple.

Preferencias de la forma de andar de NaoQi

Dentro del NaoOperator tenemos una pestaña en la que podemos modificar los diferentes parámetros que nos ofrece el API de NaoQi para la forma de andar. Esta interfaz se puede apreciar en la figura 4.17.

4.3.3. Posición Verdadera

Como ya hemos comentado en la descripción del *driver* NaoBody tenemos un componente que nos ofrece información acerca de la posición real en 3D del robot

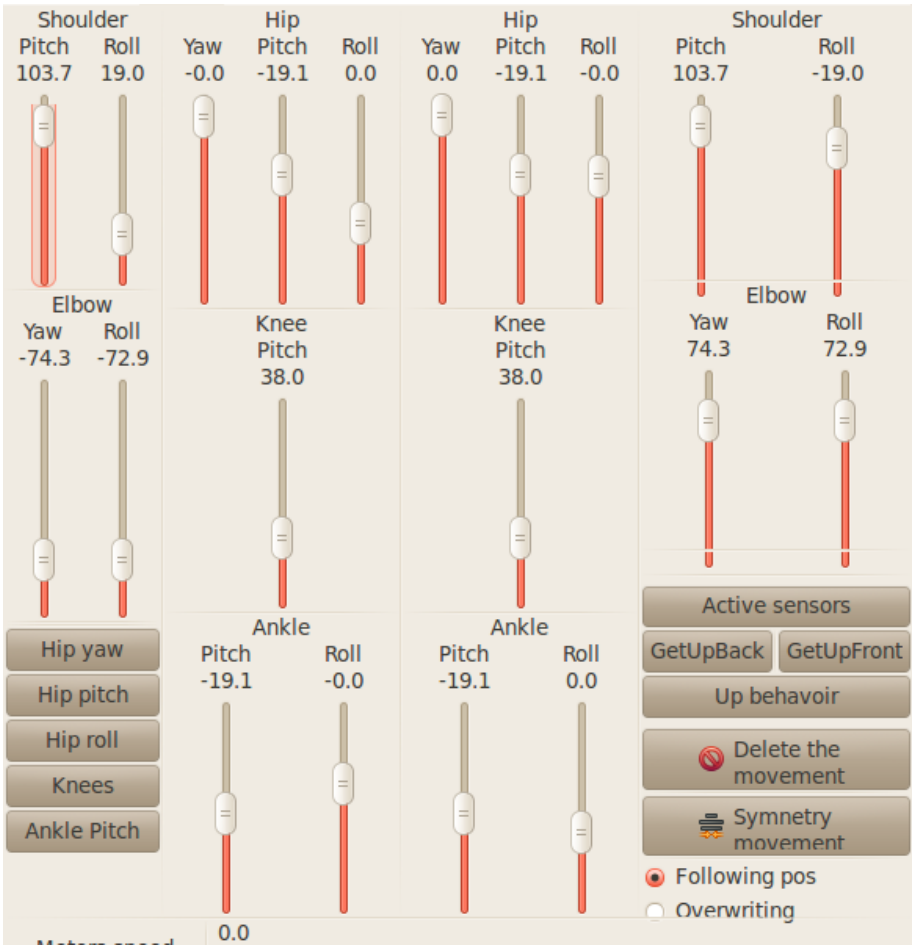


Figura 4.15: Intergaz gráfico para el control de los actuadores

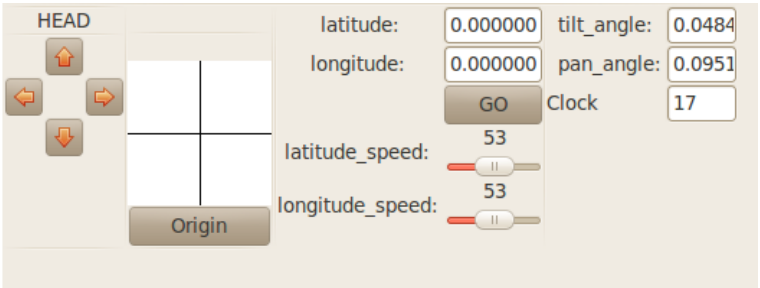


Figura 4.16: Controladores del cuello del robot.

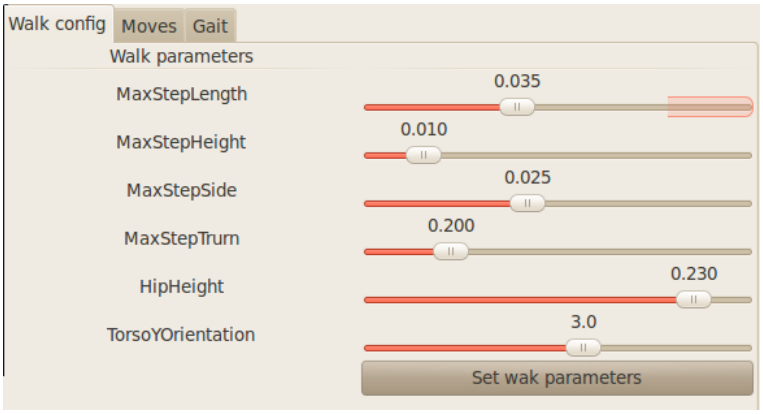


Figura 4.17: Configuración de la forma de andar desde el NaoOperator.

dentro del simulador. Para obtener esta información necesitamos incorporar un pequeño módulo al código del controlador principal de Webots llamado Supervisor. Este módulo lo hemos basado y toma su nombre de un componente desarrollado por el profesor Francisco Martín Rico [4] en el cual es capaz de obtener la posición del robot dentro del simulador, mostrarlo por pantalla dentro de la consola de Webots y mandar dicha información a otro componente software del Teamchaos [5].
NaoBody conecta el código incorporado al supervisor de Webots con JDErobot y añade nuevas funcionalidades como la sombra de posiciones del robot.
El módulo de *Posición Verdadera* del NaoOperator nos muestra de manera visual toda esta información dentro de un visor OpenGL. Este visor lo podemos controlar mediante eventos de ratón y su vez podemos elegir las opciones que deseamos que se muestren.

Este componente es muy útil para aplicaciones fundamentadas en la autolocalización ya que nos ofrece los valores de posición real del robot dentro del campo así como una representación gráfica en un visor de OpenGL. Estas herramientas nos servirían para probar nuevos algoritmos de localización y poder calcular lo fiables que son realmente.

4.3.4. Sintetizador de voz

El robot Nao dispone, como ya hemos comentado, de un dispositivo sintetizador de voz. Para el control de este dispositivo hemos incorporado un módulo en el NaoOperator para el control del mismo de una manera gráfica y sencilla4.19. Es este mismo interfaz podemos modificar los parámetros ofrecidos por el API para el sintetizador como escribir el texto que queremos que el robot sintetice.

[4]http://sites.google.com/site/franciscomartinrico/
[5]http://www.teamchaos.com

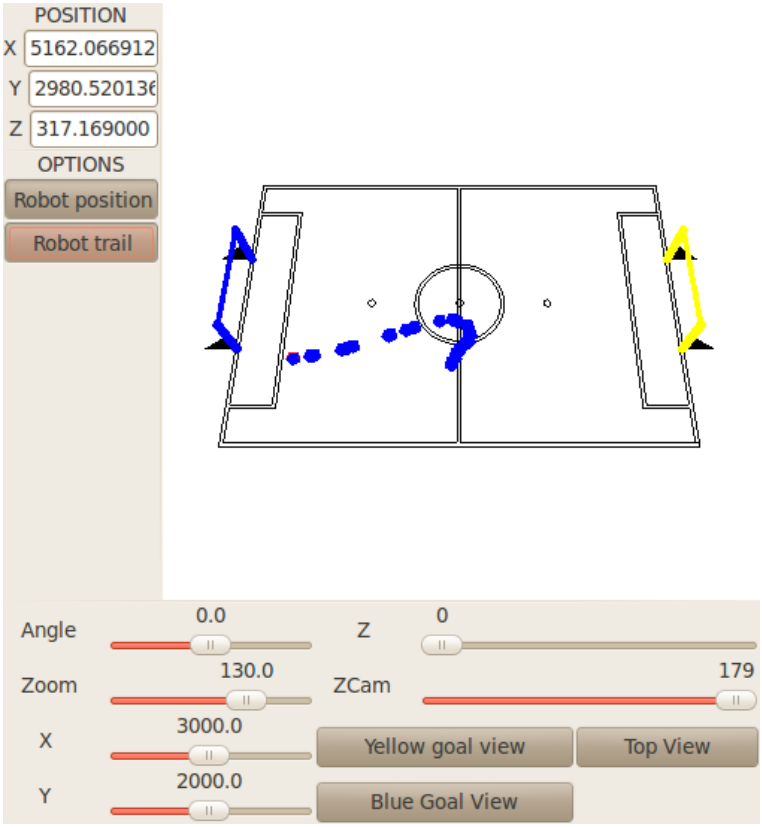


Figura 4.18: Visor del Ground Truth.

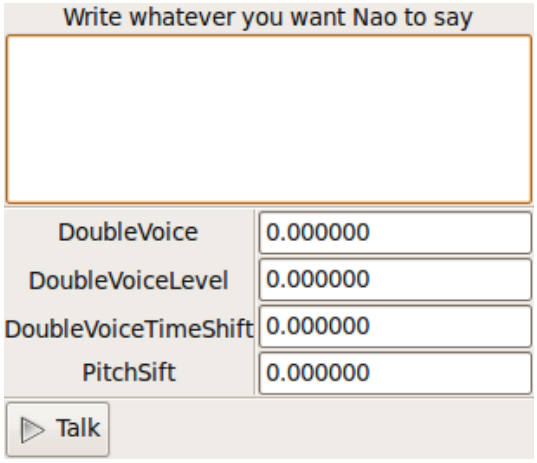


Figura 4.19: Sintetizador de voz.

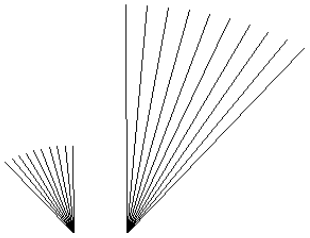


Figura 4.20: Visor gráfico del sensor de ultrasonidos.

4.3.5. Sensor de ultrasonidos

Hemos incorporado también dentro del interfaz gráfico del NaoOperator un visor gráfico 4.20 donde se muestra el valor obtenido por los dispositivos del robot así como su valor numérico.

4.3.6. Dispositivos Leds

Uno de los componentes más importantes a la hora de la depuración con el robot real son los dispositivos leds. Como ya hemos comentado en el capítulo anterior el NaoBody da soporte completo a todos los dispositivos leds del robot. Para comprobar el correcto funcionamiento de este componente y crear un sistema con el cual poder manejarlos hemos desarrollado un interfaz gráfico incorporado al NaoOperator con este fin4.21.
Este interfaz gráfico nos permite controlar los leds del robot de manera muy fácil mediante el uso de botones y podemos controlar los leds tanto independientemente como por grupos.

4.3.7. Editor de movimientos

Para crear movimientos con secuencias fijas hemos desarrollado un editor de movimientos incorporado dentro del propio NaoOperator. Con el NaoOperator podemos modificar las posición de los motores del robot y una vez en la posición deseada asignarle un valor de tiempo y guardar el fotograma. Repitiendo este paso las veces necesarias este editor es capaz de recrear el movimiento simplemente presionando un botón de PLAY.

El NaoOperator dispone de numerosas herramientas para la creación de nuevos movimientos, unas ya explicadas y otras que serán explicadas en los capítulos 5 y 6. Hasta ahora la única forma que nos ofrece el NaoOperator para crear posiciones fijas es mediante el uso del su interfaz gráfica y de los *sliders* de cada actuador. Sin embargo existe una forma aún más simple de crear posiciones fijas y es mediante la manipulación del robot real. El NaoOperator nos ofrece la posibilidad de eliminar la rigidez de uno o varios de sus actuadores para así poder crear una posición manipulando manualmente el robot. Una vez colocada la articulación en la posición deseada podemos volver al

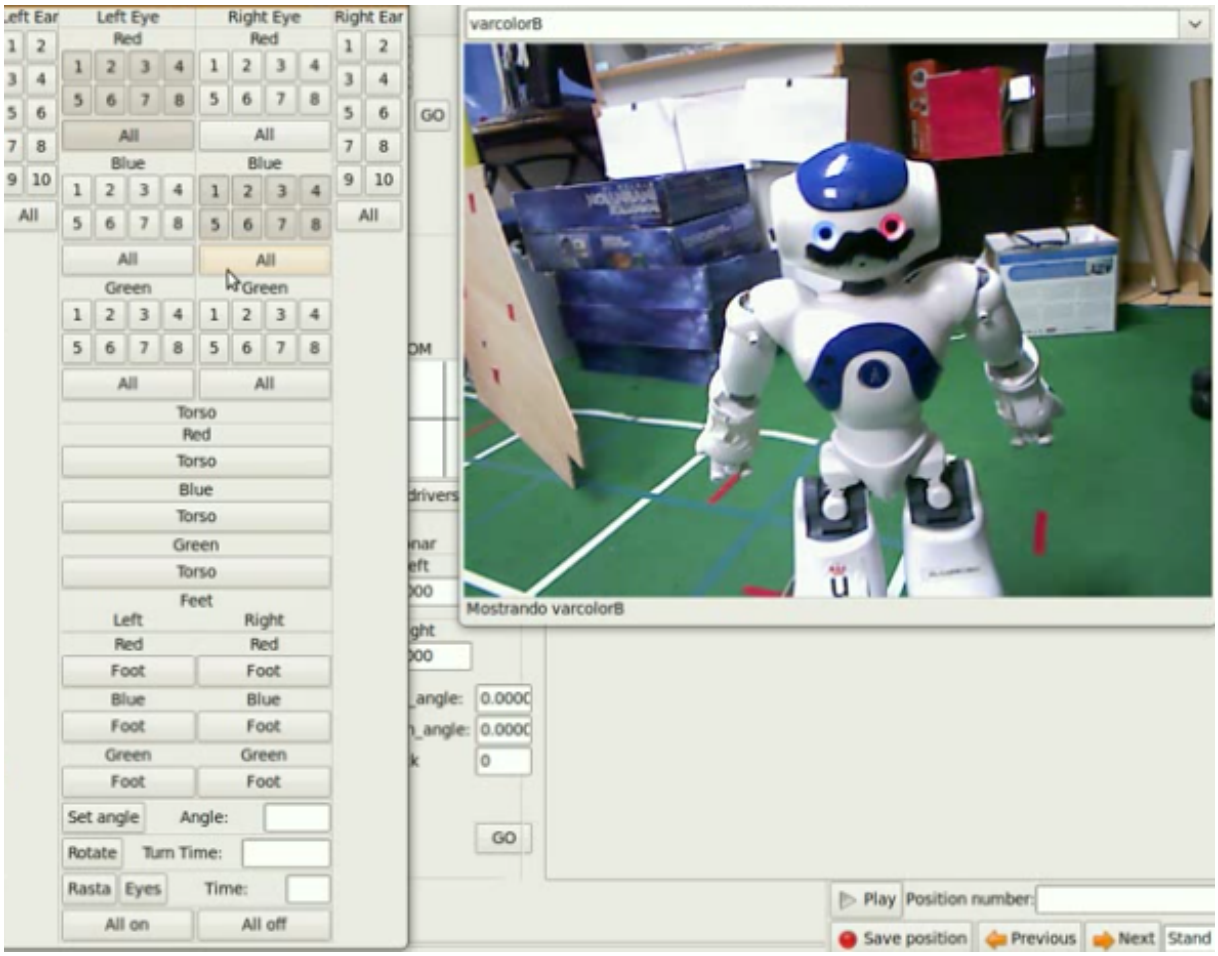


Figura 4.21: Interfaz gráfico para el control de los leds del robot.

Figura 4.22: Interfaz gráfico para el control de la rigidez de los actuadores.

fijar la rigidez y guardar esta posición como parte de un movimiento. Esto es realmente útil para la generación de movimientos usando el robot real.
El interfaz gráfico de este módulo se muestra en la figura 4.22.

Una posibilidad que nos ofrece el editor de movimiento del NaoOperator es la de cargar y guardar movimientos. Podemos guardar movimientos completos creados con el propio editor,cargarlos posteriormente y también cargar movimientos creados con cualquier otra herramienta que utiliza cualquiera de los dos formatos soportados por el NaoOperator: el propio del NaoOperator y el usado por el TeamChaos.

- **Formato de movimientos propios del NaoOperator.** El formato de movimientos propio utilizado por el NaoOperaor es un formato lineal en el tiempo y con una estructura de movimiento conjunto. La primera linea del fichero es el número de posiciones que forman el movimiento y seguidamente una secuencia de posiciones formadas por el tiempo en el cual el robot debe estar en la posición actual y seguidamente los valores correspondientes a cada uno de los actuadores. Podemos ver un ejemplo de este formato de fichero en el código 4.20.

- **Formato de movimientos utilizado por el Teamchaos.** El formato para ficheros con movimientos del TeamChaos también es utiliza un formato lineal en el tiempo pero con movimientos independientes para cada motor. Asigna individualmente a cada actuador una posición y un tiempo mientras en el formato propio del NaoOperator el tiempo es común para todos los actuadores.
  La primera línea del fichero corresponde al nombre del movimiento. La segunda, constituye una lista con los diferentes actuadores que van a formar parte del movimiento. Seguidamente una serie de lineas, tantas como actuadores, que fijan el valor de los motores en cada posición y finalmente otra serie de líneas, también tantas como actuadores, que corresponden a los valores de tiempo para cada uno de los actuadores.
  Podemos ver un ejemplo de este formato de fichero en el código 4.21.

El NaoOperator no sólo nos ofrece herramientas para la creación de movimientos sino que también disponemos de una serie de utilidades con las que podemos modificar los movimientos una vez creados. Estas utilidades van desde poder modificar

```
2
#Primera posición
time 0.300000
head_yaw 0.000000
head_pitch 0.000000
right_shoulder.pitch 109.808685
right_shoulder.roll -15.000418
right_elbow.yaw 84.999588
right_elbow.roll 20.191315
left_shoulder.pitch 100.166626
left_shoulder.roll 15.043817
left_elbow.yaw -84.999588
left_elbow.roll -39.817230
right_hip.yaw 0.152196
right_hip.pitch -32.514576
right_hip.roll -2.099405
right_knee.pitch 44.788189
right_ankle.pitch -15.385811
right_ankle.roll 2.087641
left_hip.yaw 0.152196
left_hip.pitch -21.614721
left_hip.roll -6.022437
left_knee.pitch 44.608257
left_ankle.pitch -26.121635
left_ankle.roll 6.100786
#Segunda posición
time 0.700000
head_yaw 0.000000
head_pitch 0.000000
right_shoulder.pitch 99.750000
right_shoulder.roll -13.750000
right_elbow.yaw 79.250000
right_elbow.roll 25.500000
left_shoulder.pitch 100.166626
left_shoulder.roll 15.043817
left_elbow.yaw -84.999588
left_elbow.roll -35.747837
right_hip.yaw 0.154033
right_hip.pitch -26.607229
right_hip.roll 8.688103
right_knee.pitch 42.710365
right_ankle.pitch -19.254148
right_ankle.roll -9.763948
left_hip.yaw 0.154033
left_hip.pitch -16.048689
left_hip.roll 5.311640
left_knee.pitch 42.396385
left_ankle.pitch -29.471247
left_ankle.roll -5.298602
```

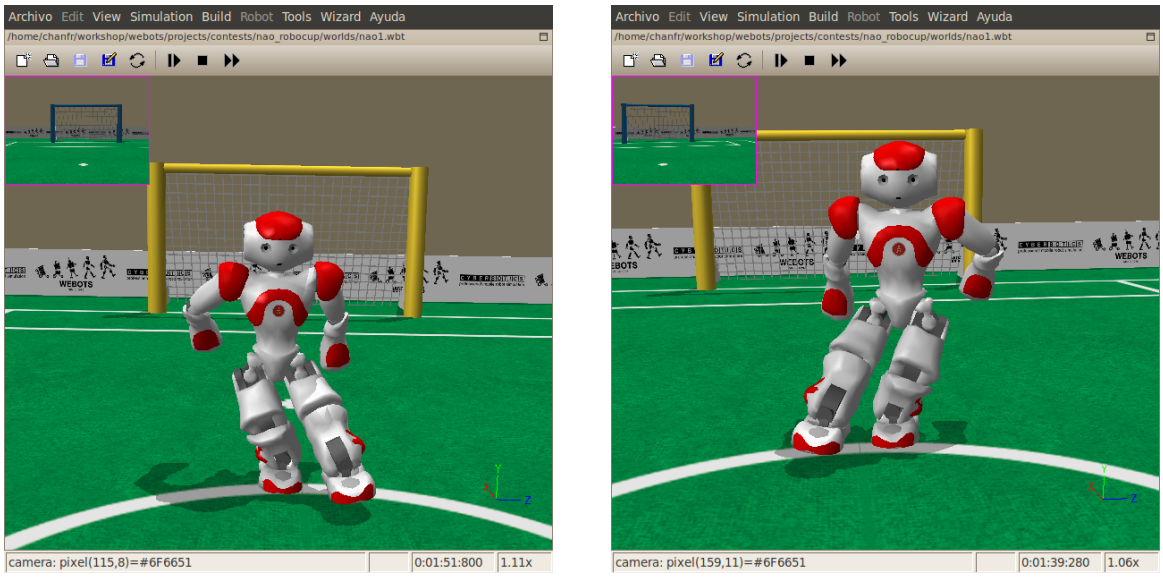
Listing 4.20: Fichero con un movimiento con el formato propio del NaoOperator

```
movement1
LShoulderPitch LShoulderRoll LElbowYaw LElbowRoll LHipYawPitch LHipRoll
LHipPitch LKneePitch LAnklePitch LAnkleRoll RHipYawPitch RHipRoll
RHipPitch RKneePitch RAnklePitch RAnkleRoll RShoulderPitch RShoulderRoll
RElbowYaw RElbowRoll
1.3999999 1.3999999 1.415924 2.13537 1.92828 1.902202 1.4
0.35 0.35 0.354396 0.783916 0.58603 0.34365797 0.35
-1.4 -1.4 -1.388228 0.28229797 -1.317664 -1.328402 -1.4
-0.35000 -0.35000 -0.35132 -0.96 -0.940 -0.917 -0.3499
0.0 -5.825886E-4 -0.058705244 -0.08768 -0.101 0.0152 -7.4505E-9
-0.034708 -0.32242477 -0.39433804 -0.396414 -0.3986 -0.43632 -0.034708
-0.493795 -0.47214437 -0.5065343 -0.56492 -0.51371 -0.6140565 -0.4935
0.89512 0.89705706 0.9329779 1.0098867 1.0127198 1.082835 0.89512
-0.401324 -0.42857915 -0.43015054 -0.4449698 -0.498666 -0.02167 -0.41324
0.034708 0.3146474 0.3869 0.39672518 0.3982089 0.3767433 0.034708053
0.0 -5.825886E-4 -0.058744 -0.08411 -0.1013692 0.015205957 -7.4505806E-9
0.03488 -0.25807 -0.36745423 -0.41158 -0.35126376 -0.423168 0.034879923
-0.52 -0.5004 -0.856 -0.8766 -0.88775 -0.1798 -0.5102
0.913006 0.988262 1.4514009 0.1933743 1.6900845 1.1888793 0.91300607
-0.398 -0.48833 -0.677 0.6445 -0.78684 -0.33349 -0.39804
-0.0301 0.250042 0.324413 0.30610943 0.325148 0.464203 -0.034879982
1.4001 1.40001 1.428112 2.1276162 1.797806 1.6536101 1.4
-0.35 -0.35 -0.61355805 -0.29602003 -0.5982181 -0.315905 -0.39996
1.4 1.4 1.385244 0.23474397 0.87288797 1.472682 1.4
0.35000002 0.35000002 0.33743805 0.231204 0.277203 0.31006 0.35
2.0 4.0 6.0 6.3 8.3 10.3 12.3
2.0 4.0 6.0 6.3 8.3 10.3 12.3
2.0 4.0 6.0 6.3 8.3 10.3 12.3
2.0 4.0 6.0 6.3 8.3 10.3 12.3
2.0 4.0 6.0 6.3 8.3 10.3 12.3
2.0 4.0 6.0 6.3 8.3 10.3 12.3
2.0 4.0 6.0 6.3 8.3 10.3 12.3
2.0 4.0 6.0 6.3 8.3 10.3 12.3
2.0 4.0 6.0 6.3 8.3 10.3 12.3
2.0 4.0 6.0 6.3 8.3 10.3 12.3
2.0 4.0 6.0 6.3 8.3 10.3 12.3
2.0 4.0 6.0 6.3 8.3 10.3 12.3
2.0 4.0 6.0 6.3 8.3 10.3 12.3
2.0 4.0 6.0 6.3 8.3 10.3 12.3
2.0 4.0 6.0 6.3 8.3 10.3 12.3
2.0 4.0 6.0 6.3 8.3 10.3 12.3
2.0 4.0 6.0 6.3 8.3 10.3 12.3
2.0 4.0 6.0 6.3 8.3 10.3 12.3
2.0 4.0 6.0 6.3 8.3 10.3 12.3
```

Listing 4.21: Fichero con un movimiento con el formato del Teamchaos

independientemente cada una de las posiciones que componen un movimientos como herramientas más complejas que se aplican a todos las posiciones de un movimiento:

- **Movimientos simétricos** El NaoOperator incorpora una herramienta para la generación de un movimiento simétrico con respecto a uno dado 4.23. Esto es muy útil para trabajar con movimientos unilaterales ya que una vez creado para un lado la generación del mismo movimiento para el lado contrario es automática.



(a) (b)

Figura 4.23: Golpeo izquierdo (a). Golpeo derecho (simétrico)(b).

- **Herramienta para la modificación de intervalos** Una vez generado un movimiento nos puede surgir la necesidad de variar la velocidad general del movimiento o el tiempo de transición de una posición a otra. Con esta herramienta podemos modificar el tiempo asignado a cada posición por separado o incluso asignar un valor común a todas las transiciones entre las distintas posiciones.

- **Máximos y mínimos sobre movimientos fijos** Una forma de modificar de forma automática esta serie de movimientos es mediante la variación de máximos y mínimos. Buscamos el máximo y el mínimo de cada motor dentro de la secuencia y los variamos, de esta manera con las nuevas amplitudes asignamos de forma

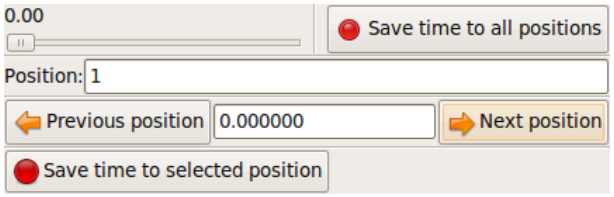


Figura 4.24: Interfaz gráfico para la variación de los tiempos entre posiciones.

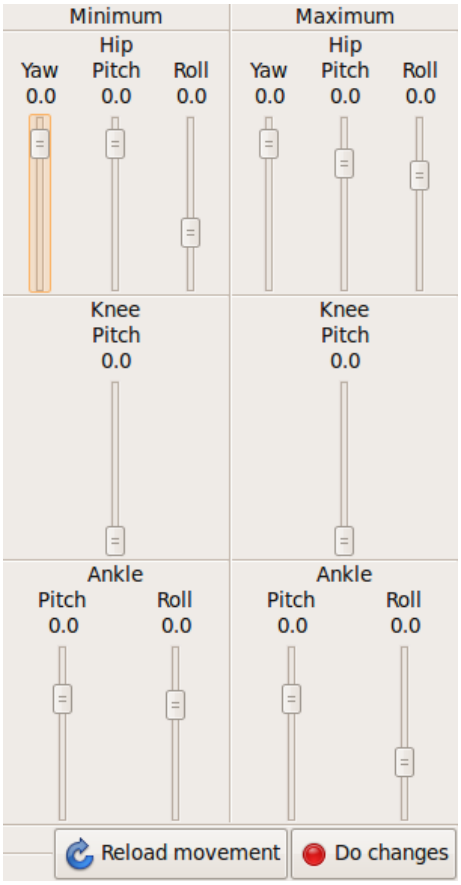


Figura 4.25: Interfaz gráfico para la variación de valores máximos y mínimos de un movimiento.

proporcional los nuevos valores a cada uno de los fotogramas.
Usando esta herramienta lo que conseguimos es suavizar o acentuar un movimiento sobre uno o varios motores, por ejemplo, una vez creado un movimientos con secuencias fijas y observamos que se producen movimientos muy bruscos tenemos la posibilidad de reducir la amplitud de los valores en uno o varios motores para suavizar el movimiento.

4.4. Aplicación para controlar del robot Nao desde iPhone.

Para ampliar la funcionalidad del *driver* NaoBody y las posibilidades que nos ofrece a la hora de controlar el robot Nao hemos desarrollado un nuevo componente que nos permite el control del mismo sin tener que estar sentado delante de un ordenador. Este componente de control remoto ha sido desarrollado para le dispositivo iPhone de Apple bajo la plataforma iOS 4.0.1 y usando Objetive-C.
Esta nueva aplicación4.26 nos da acceso a todos los actuadores del robot así como herramientas para controlar el su movimiento o desplazamiento desde un terminal móvil.

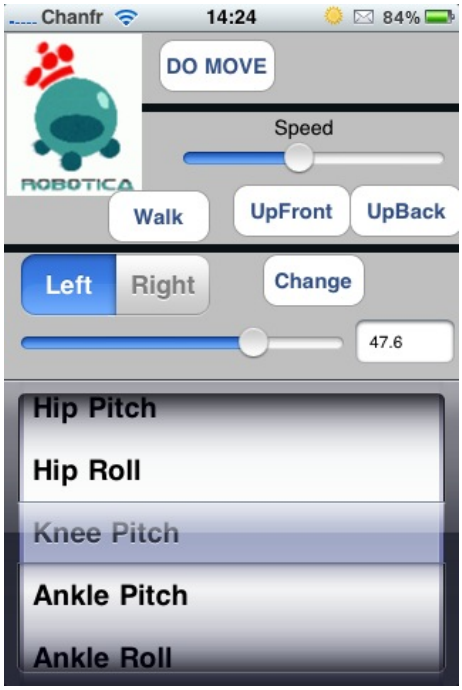


Figura 4.26: Aplicación iPhone para controlar el robot Nao.

El funcionamiento de esta aplicación precisa del NaoBody para su funcionamiento. Utiliza un protocolo de cliente-servidor donde el Naobody actúa como servidor esperando información mediante la escucha en una IP y un Puerto donde el cliente instalado en el dispositivo manda la información. El NaoBody procesa esta información y crea las señales oportunas para cada uno de los actuadores.
Esta aplicación móvil es una alternativa sencilla para el control del robot Nao con algunas limitaciones respecto del NaoOperator.
La aplicación está desarrollada utilizando el entorno de programación Xcode proporcionado por Apple para la creación de nuevos programas tanto para MacOS X como para iOS. Al igual que en el componente incorporado en el NaoBody para la comunicación con dispositivos móviles hemos utilizado *sockets* desde Objetive C para la comunicación con JDErobot desde el *driver*.
El implementación de la aplicación está basada en eventos del interfaz gráfico, por lo que a cada objeto de la pantalla le están asignadas una serie de funciones que se disparan al interactuar con ellos.

Capítulo 5
Modos de caminar

Después de un primer paso en el que hemos explicado las herramientas creadas en este proyecto, desarrollaremos en este capítulo cómo hemos modelado la forma de andar de un robot bípedo con una serie de parámetros.

La marcha de un robot humanoide es similar a la forma de andar de una persona humana. Los humanoides tienen 2 patas con 3 articulaciones en cada pata: la cadera, la rodilla y el tobillo. La marcha tanto de un robot bípedo como de un humano es periódica, para movernos encadenamos cierto número de *pasos* con lo que conseguimos desplazarnos. Si nos movemos a una velocidad constante estos pasos que componen la marcha son exactamente iguales. Basándonos en este hecho podemos simplificar la marcha de un robot humanoide en pasos. Otra de las características importantes son las simetrías, para conseguir una marcha correcta y lineal el movimiento de una de las patas tiene que ser simétrico al de la pata contraria pero con un cierto retardo, exactamente la mitad del tiempo que dura un paso. Para que la marcha sea lo suficientemente estable como para que el robot no se caiga todos los actuadores que intervienen deben estar correctamente acompasados y sincronizados.

Todo lo explicado anteriormente se complica si hablamos de correr, la acción de correr se define como *Andar rápidamente y con tanto impulso que, entre un paso y el siguiente, quedan por un momento amos pies en el aire*. Esta marcha a velocidad aumentada es tremendamente compleja para aplicarla en un robot humanoide. En primer lugar tendríamos que tener un sistema de amortiguación para paliar el impacto al apoyar el pie contra el suelo, un sistema avanzado de estabilidad... El fabricante Honda, interesado en movimientos con robots humanoides, creó en 1986 creó su primer robot andador (En la figura 5.1 podemos ver la evolución de estos robots). Finalmente, tras muchos años de estudio de modelos basados en movimientos humanos, ha conseguido que su último robot humanoide, ASIMO, sea capaz de correr a una velocidad máxima de 7km/hora, para esto Honda ha tenido que combinar hardware muy perceptivo una tecnología software muy avanzada capaz de aportar gran versatilidad y flexibilidad a los movimientos como mantener equilibrio, evitar patinazos, giros en el aire...

Debido a la gran dificultad que supone modelizar esta marcha con velocidad aumentada nos centraremos en la caminata normal de un robot humanoide. Aún así conseguir una forma de andar simple para un robot humanoide ya supone gran

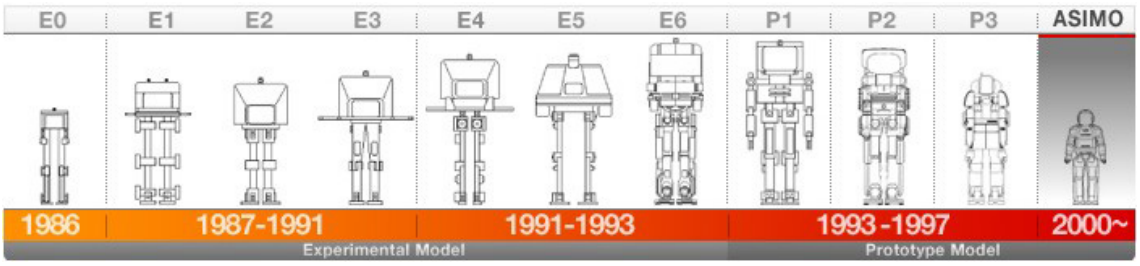


Figura 5.1: Evolución de humanoides del fabricante Honda

complejidad, sobre todo en el diseño de software ya que tenemos que controlar muchos actuadores y debe haber una buena coordinación entre todos ellos y bastante estabilidad.

5.1. Control de movimientos desde NaoQi

NaoQi nos ofrece herramientas para realizar movimientos con el robot Nao. Aunque estas formas de realizar movimientos no son suficientes, ya que no nos ofrece unas opciones lo suficientemente flexibles y rápidas como para conseguir una locomoción óptima, hemos desarrollado este proyecto precisamente para ampliarlas. Muchas de nuestras aplicaciones sí que usan NaoQi como base de funcionamiento. Básicamente nos ofrece tres formas para conseguir movimientos con el robot Nao: acceso de bajo nivel a los actuadores de forma independiente, funciones de alto nivel para hacer que el robot ande y una serie de funciones para crear movimientos específicos a partir de secuencias fijas.

5.1.1. Acceso a actuadores de forma independiente

El API de NaoQi nos ofrece funciones de bajo nivel para acceder de forma independiente a cada uno de los actuadores del robot. Esto nos permite fijar una velocidad para cada uno de los motores y una posición, base fundamental del movimiento.

En lo referente a la marcha, los actuadores que nos interesan son los que componen los miembros inferiores. Como hemos introducido anteriormente, el robot Nao dispone de 3 articulaciones en las patas: cadera, rodilla y tobillo. En la cadera tenemos 3 grados de libertad: hip-pitch, hip-yawpitch y hip-roll; en la rodilla sólo uno: knee-pitch y en el tobillo dos: ankle-pitch y ankle-roll. En la figura 5.2

En la tabla 5.1.1 tenemos una descripción de rangos, velocidades y fuerzas de cada uno de estos actuadores.

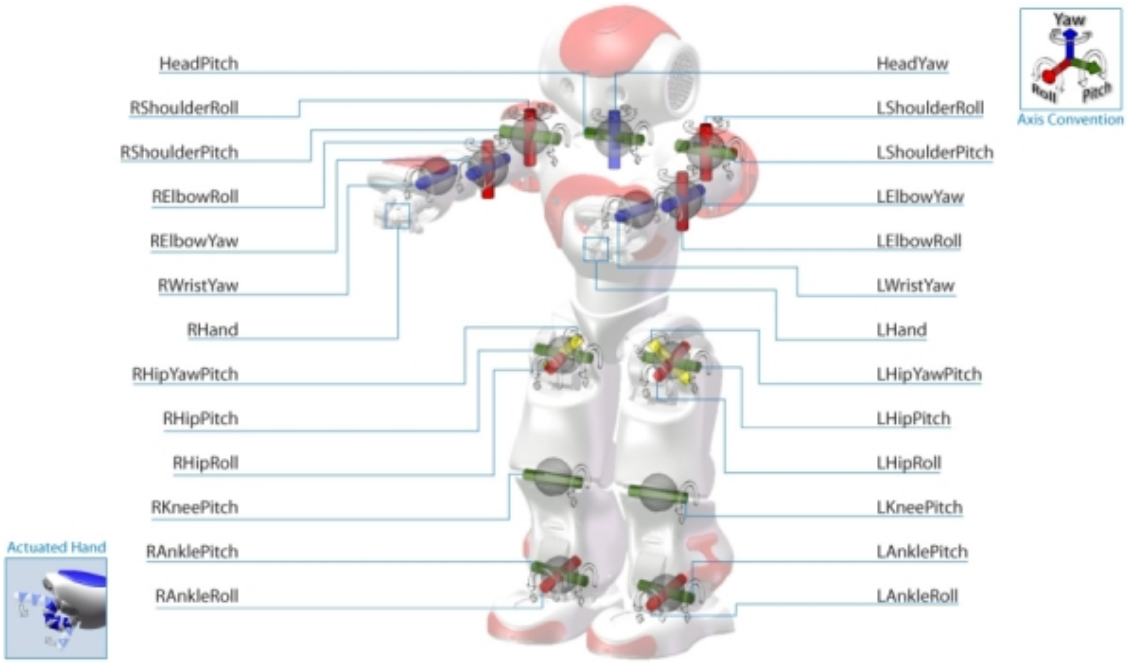


Figura 5.2: Situación de todos los actuadores del robot Nao

Motor	min	max	Velocidad Nominal	Par nominal
LHipYawPitch	-90	0	188.67 °/s (3.77°/20ms)	2.47 Nm
RHipYawPitch	-90	0	188.67 °/s (3.77°/20ms)	2.47 Nm
LHipRoll	-45	25	188.67 °/s (3.77°/20ms)	2.47 Nm
RHipRoll	-25	45	188.67 °/s (3.77°/20ms)	2.47 Nm
LHipPitch	-100	25	290.25 °/s (5.80°/20ms)	1.61 Nm
RHipPitch	-100	25	290.25 °/s (5.80°/20ms)	1.61 Nm
LKneePitch	0	130	290.25 °/s (5.80°/20ms)	1.61 Nm
RKneePitch	0	130	290.25 °/s (5.80°/20ms)	1.61 Nm
LAnklePitch	-75	45	290.25 °/s (5.80°/20ms)	1.61 Nm
RAnklePitch	-75	45	290.25 °/s (5.80°/20ms)	1.61 Nm
LAnkleRoll	-45	25	188.67 °/s (3.77°/20ms)	2.47 Nm
RAnkleRoll	-25	45	188.67 °/s (3.77°/20ms)	2.47 Nm

Cuadro 5.1: Tabla descriptiva de los actuadores

Las funciones ofrecidas por el fabricante para controlar los actuadores de forma independiente son básicamente 2:

La única diferencia entre estas dos funciones es cómo fijamos la velocidad, en la primera lo que tenemos que fijar es el tiempo que queremos que tarde en la transición mientras que en la segunda fijamos la velocidad de manera explícita mediante un valor porcentual de la velocidad máxima.

5.1.2. Formas de andar de NaoQi

El movimiento descrito en el apartado anterior no es suficiente para conseguir que el robot se desplace de forma correcta, ya que las funciones ofrecidas son de muy bajo nivel y controlar cada uno de los actuadores por separado es una tarea compleja. Por

```
void gotoAngle (string pJointName, float pAngle, float pDuration,
    int pInterpolationType);


void gotoAngleWithSpeed (string pJointName, float pAngle, int pSpeedPercent,
    int pInterpolationType);


//Descripción de los parámetros:

string pJointName

    /*Name of the Joint. Could be: "HeadYaw", "HeadPitch", "LShoulderPitch",
        "LShoulderRoll", "LElbowYaw", "LElbowRoll", "LWristYaw", "LHand",
        "LHipYawPitch", "LHipRoll", "LHipPitch", "LKneePitch", "LAnklePitch",
        "LAnkleRoll", "RHipYawPitch", "RHipRoll", "RHipPitch", "RKneePitch",
        "RAnklePitch", "RAnkleRoll", "RShoulderPitch", "RShoulderRoll",
        "RElbowYaw", "RElbowRoll", "RWristYaw", "RHand"*/

float pAngle

    //Ángulo al que queremos mover el robot.

int pSpeedPercent

    //Valor porcentual relativo a la velocidad máximas (1~100).

int pInterpolationType

    /*Tipo de interpolación para el movimiento. { INTERPOLATION_LINEAR = 0,
        INTERPOLATION_SMOOTH = 1 }*/
```

Listing 5.1: Funciones ofrecidas por NaoQi para el acceso de bajo nivel de forma independiente a los actuadores

ello NaoQi nos ofrece una serie de funciones de más alto nivel y más abstractas con las que podemos hacer que el robot ande, pudiendo elegir entre varias opciones:

- Andar el línea recta: nos permite un desplazamiento frontal del robot, tal y como se aprecia en la figura 5.3(a), utilizando la función *walkStraight* explicada en el código 5.2.

```
void walkStraight (float pDistance, int pNumSamplesPerStep)
Parameters

float pDistance

    //Distancia en metros a recorrer

int pNumSamplesPerStep

    //Número de ciclos de 20ms por paso
```

Listing 5.2: Funciones ofrecidas por NaoQi para el desplazamiento frontal del robot.

- Desplazamiento lateral: nos permite movernos lateralmente sin tener que girar y luego reanudar la marcha, tal y como se aprecia en la figura 5.3(b), utilizando la función *walkSideways* explicada en el código 5.3.

```
void walkSideways (float pDistance, int pNumSamplesPerStep)
Parameters

float pDistance

    //Distancia en metros a recorrer

int pNumSamplesPerStep

    //Número de ciclos de 20ms por paso
```

Listing 5.3: Funciones ofrecidas por NaoQi para el desplazamiento lateral del robot.

- Andar de forma arqueada: nos permite realizar una marcha formando un arco en nuestra trayectoria, tal y como se aprecia en la figura 5.3(c), utilizando la función *walkArc* explicada en el código 5.4.

- Giro: nos permite realizar giros en el robot sobre si mismo para cambiar la dirección de la marcha tal y como se aprecia en la figura 5.3(d) utilizando la función *turn* explicada en el código 5.5.

Todas estas formas de desplazarse admiten, además de los parámetros específicos para cada una de estas funciones, una serie de parámetros generales configurables a

```
void walkArc (float pAngle, float pRadius, int pNumSamplesPerStep)
Parameters

float pAngle

    //Ángulo de giro del el circulo en radianes.

float pRadius

    //Radio del círculo en metros

int pNumSamplesPerStep

    //Número de ciclos de 20ms por paso
```

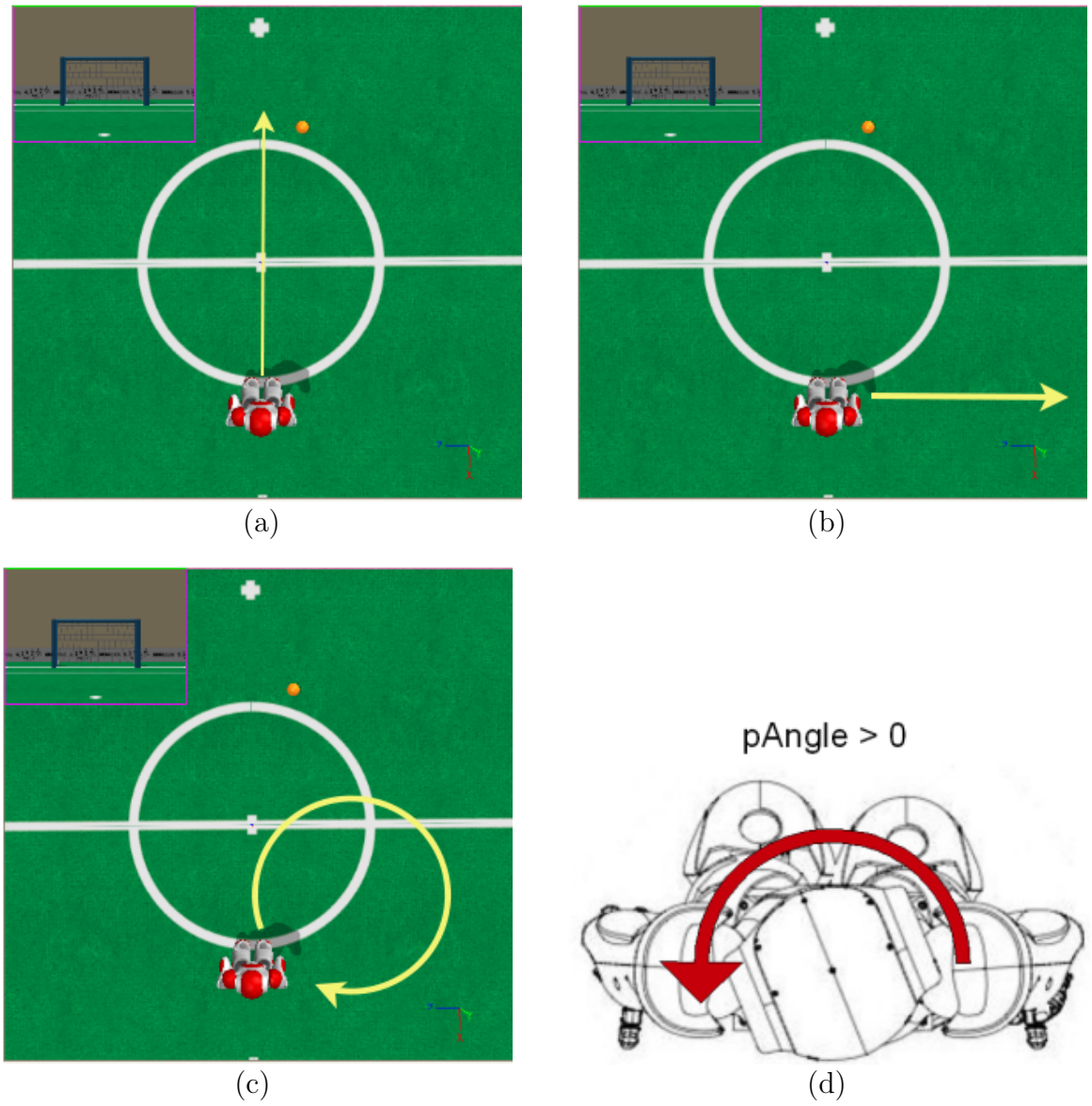
Listing 5.4: Funciones ofrecidas por NaoQi para el desplazamiento arqueado del robot.

```
void turn (float pAngle, int pNumSamplesPerStep)
Parameters

float pAngle

    // Angulo de giro en radianes

int pNumSamplesPerStep

    //Número de ciclos de 20ms por paso
```

Listing 5.5: Funciones ofrecidas por NaoQi para el desplazamiento giro del robot.

```
    void setWalkConfig(float pMaxStepLength, float pMaxStepHeight,
        float pMaxStepSide, float pMaxStepTurn, float pHipHeight,
        float pTorsoYOrientation);
```

Listing 5.6: Parámetros configurables en la marcha del robot.



(a) (b) (c) (d)

Figura 5.3: Trayectoria de un movimiento frontal (a). Trayectoria de un movimiento lateral (b). Trayectoria de un movimiento arqueado (c). Trayectoria de un movimiento giro(d).

través de la función presentada el código 5.6. Los parámetros que se pueden configurar son:

- pMaxStepLength: longitud máxima de un paso en metros (0.0 - 0.09).
- pMaxStepHeight: la altura máxima durante el paso en metros (0.0 - 0.08).
- pMaxStepSide: distancia lateral máxima en metros durante un paso en metros (0.0 - 0.06).
- pMaxStepTurn: variación máxima en el eje Z durante un paso en radianes (0.0 - 1.0).
- pHipHeight: altura de la cadera durante la marcha en metros (0.15 - 0.244).
- pTorsoYOrientation: define la orientación del torso en grados con respecto al eje Y (-10.0 - 10).

Todas estas formas de andar tienen un cierto número de parámetros que podemos modificar aunque sigue siendo muy cerrada, ya se nos permite una permutación directa entre las distintas opciones. NaoQi valora, ante todo, la estabilidad del robot, por lo que hace que el robot vuelva a una posición estable antes de iniciar la marcha cada vez que variamos alguno de los parámetros. Por ello estos cambios son costosos desde el punto de vista temporal. Estas funciones son útiles, ya que conseguimos hacer que el robot ande con cierta estabilidad y con una velocidad adecuada, sin embargo al ser ofrecidas por el fabricante cualquiera las puede usar. Una ventaja importante en cualquier competición robótica es lógicamente la locomoción y conseguir una marcha que mejore la ofrecida por el fabricante supone partir con cierta ventaja. Por ello buscamos conseguir una forma de andar que mejore los aspectos que nos parecen insuficientes en la ofrecida por NaoQi.

5.1.3. Soporte de secuencias fijas por parte de NaoQi

Una tercera posibilidad que nos ofrece NaoQi para mover al robot es una función capaz de generar secuencias a partir de posiciones fijas. El API nos da soporte directo a nuestra idea de las secuencias fijas explicado en el capitulo anterior. Nosotros le tendremos que pasar las posiciones de los actuadores en cada uno de los fotogramas y el tiempo de transición y NaoQi se encargará de interpolarlas para crear un movimiento continuado y suave. La función ofrecida por el API para este fin es *doMove* explicada en el código 5.7.

Este tipo de se movimientos es muy útil a la hora de realizar cierto tipo de movimientos que no son variables como por ejemplo colocar al robot en una posición inicial, diferentes tipos movimientos para levantarse, disparos con las piernas...

```
    void doMove (array pJointNames, array pAngles, array pTimes,
        int pInterpolationType);

//Parámetros
array pJointNames
   //Nombres de los actuadores comunes a la matriz de ángulos y de tiempos.
array pAngles
    //Matriz con los ángulos (en radianes) correspondiente a cada uno
        //de los actuadores
array pTimes
    //Matriz con el tiempo (en segundos) correspondiente a cada uno
        //de los actuadores
int pInterpolationType
    //Tipo de interpolación{ INTERPOLATION_LINEAR = 0,
            //INTERPOLATION_SMOOTH = 1 }

//EJEMPLO:
    /*Este código hará el mismo movimiento en el mismo plazo con
        dos articulaciones
    Cada articulación puede tener diferentes ángulos y tiempos,
    siempre y cuando el número de los tiempos para cada articulación
    corresponda con el número de ángulos*/

    ALValue motors, allAngles, allTimes, angles, times;
    motors.arrayPush("HeadYaw");
    motors.arrayPush("HeadPitch");
    times.arrayPush(1.0f); // tiempo en segundos
    times.arrayPush(2.0f);
    times.arrayPush(3.0f);
    allTimes.arrayPush(times); // tiempos del HeadYaw
    allTimes.arrayPush(times); // tiempos del HeadPitch
    angles.arrayPush(0.0f); // ángulos en radianes
    angles.arrayPush(0.2f);
    angles.arrayPush(0.0f);
    allAngles.arrayPush(angles); //ángulos de HeadYaw
    allAngles.arrayPush(angles); //ángulos de HeadPitch
    int smoothInterpolation = 1;
    fMotion->doMove(motors,allAngles,allTimes,smoothInterpolation);
```

Listing 5.7: Función con soporte para secuencias fijas.

5.2. Forma de andar propia usando secuencias fijas.

Como hemos explicado en los apartado anteriores, para generar una marcha con un robot humanoide sólo necesitamos tener el movimiento de un paso. Mediante la repetición de este movimiento podremos generar la marcha del robot.

Lo primero que tenemos que conseguir es que el robot sea capaz de dar un paso sin necesidad de utilizar directamente las funciones que nos ofrece NaoQi para este fin. Para ello podemos utilizar el editor de movimientos incorporado en el NaoOperator del que hemos hablado en el capítulo anterior. Con esta herramienta podemos crear cualquier movimiento basándolo en la idea de los fotogramas. Como crear un movimiento capaz de hacer que el robot ande de manera efectiva desde cero es una tarea muy compleja lo que haremos será capturar un paso de la forma de andar de NaoQi, el cual nos servirá de base para continuar con nuestro estudio.

Para capturar el paso de NaoQi hemos utilizado una herramienta incorporada en el NaoOperator para la de captura automática de movimientos. Con esta herramienta somos capaces de dividir un movimiento realizado por el robot Nao de forma automática, por ejemplo, mediante el API del fabricante, en diferentes fotogramas. La aplicación se pone en modo pasivo y sólo lee valores de los sensores sin dar ninguna orden a los actuadores por lo que no interviene en el movimiento y hace una captura de la posición cada x ms. Una vez capturados los fotogramas la secuencia es directamente compatible con el editor de movimientos del NaoOperator por lo que podremos utilizar todas las posibilidades que éste nos ofrece: variar el tiempo de intervalos (con lo que conseguimos aumentar o reducir la velocidad de la marcha), cambiar fotogramas específicos, borrar fotogramas innecesarios o guardar los movimientos. Esto nos da la posibilidad de refinar fotograma a fotograma el movimiento capturado.

El problema de esta forma de andar es la difícil parametrización que nos ofrece, únicamente podemos variar la velocidad de forma indirecta. Para generar formas distintas de locomoción tendríamos que modificar el valor de los actuadores en cada fotograma, tarea muy compleja teniendo en cuenta que en la marcha del robot intervienen 12 actuadores, suponiendo que utilizamos movimientos que 10 fotogramas serían *120 parámetros* que tenemos que fijar para conseguir una nueva forma de andar. Las gráficas de los recorridos que realizan cada uno de los actuadores las podemos ver en la figura 5.2.

En este punto ya somos capaces de conseguir que el robot se mueva sin la necesidad de utilizar directamente la forma de andar de NaoQi, mediante movimientos basados en secuencias fijas que emula la forma de andar del fabricante. Con esta forma de locomoción hemos paliado el problema del enlace de movimientos, ya no es necesario poner el robot en una posición específica para iniciar la marcha ni es necesario parar el movimiento para incrementar la velocidad. Como realizamos la marcha a partir de pasos simples podemos incrementar la velocidad del paso (reduciendo el intervalo entre cada fotograma) y acolar este paso al movimiento, una vez que éste llegue al robot

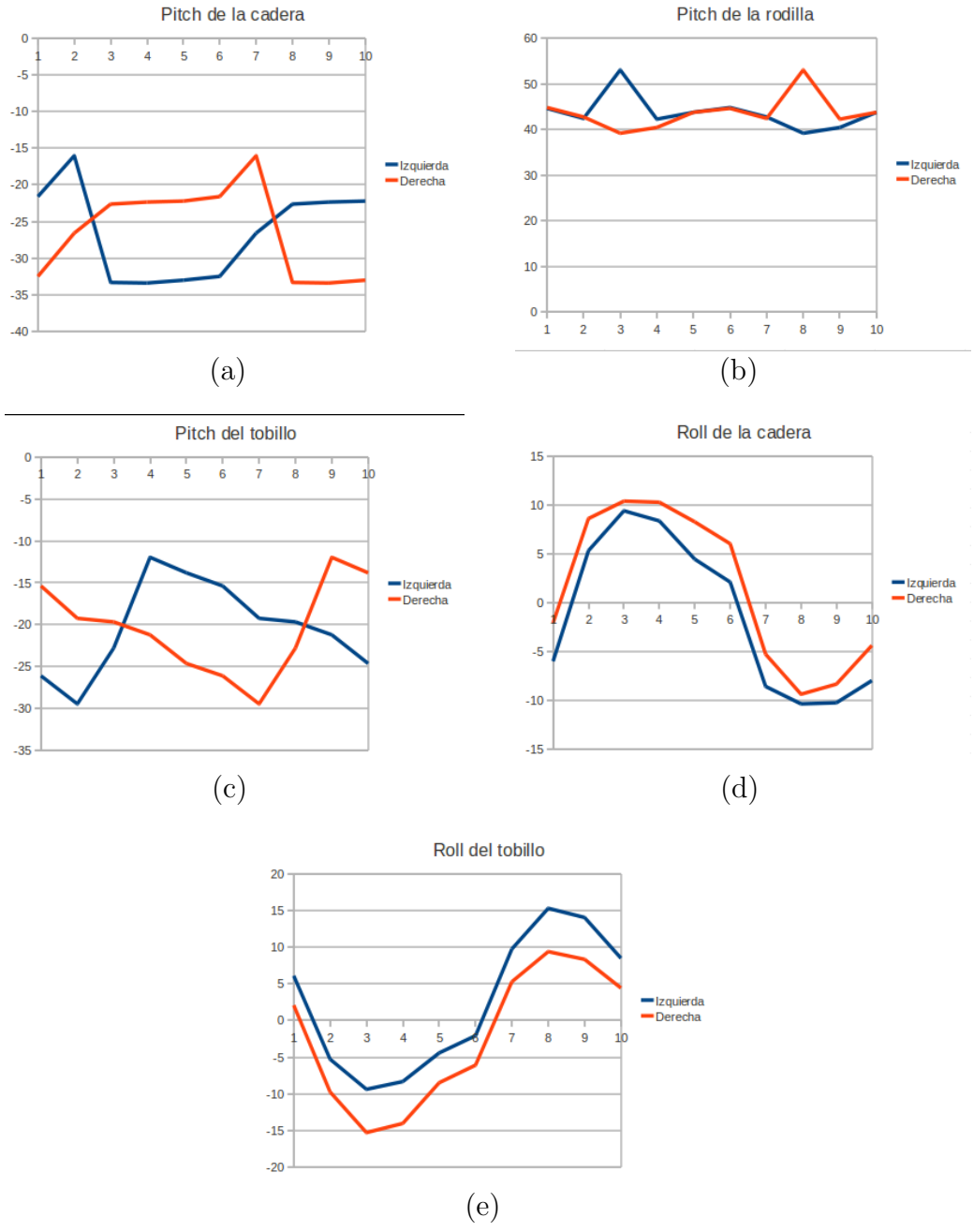


Figura 5.4: Gráficas de los actuadores que intervienen en la marcha

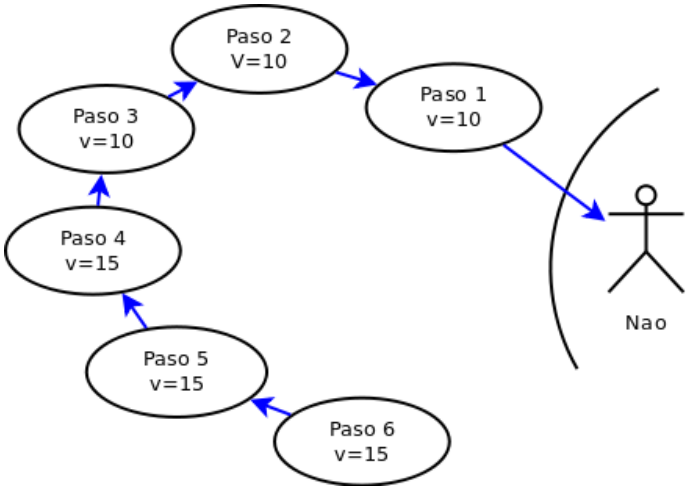


Figura 5.5: Esquema del enlazamiento de movimientos con cambio de velocidad.

empezará a andar con una velocidad modificada sin necesitar de realizar ninguna pausa. Este idea la podemos ver de forma bastante gráfica en la figura **??**.

5.3. Modelización de caminata como ondas periódicas

Como la parametrización de un paso con 120 parámetros es excesivo para buscar en el espacio de las posibles caminatas tenemos que conseguir alguna forma de conseguir disminuirlos para conseguir un espacio que sí sea factible de explorar.

Una posible solución para reducir el número de parámetros es, por cada actuador que interviene en la locomoción, conseguir una función que defina el movimiento. De esta manera los parámetros no dependen del número de fotogramas que componen el movimiento sino de los parámetros de la función característica. La idea final de la modelización es conseguir un número lo suficientemente reducido de parámetros para poder realizar una búsqueda y encontrar las mejores formas de andar.

5.3.1. Modelo parametrizado de la caminata

El movimiento que va a ser base para nuestra parametrización es el capturado con secuencias fijas en al apartado anterior. Para la captura de este movimiento hemos utilizado un total de 10 fotogramas por lo que tendremos un total de 10 valores que describen la trayectoria de cada uno de los motores.

Los actuadores que intervienen en la marcha del robot Nao en cada pata son:

1. Pitch de la cadera.
2. Yaw de la cadera.

3. Roll de la cadera.
4. Pitch de la rodilla.
5. Roll del tobillo.
6. Pitch del tobillo.

De estos 12 actuadores obviaremos 2 de ellos desde un principio, los yaw de la cadera. Estos actuadores lo único que controlan es el rodamiento de la cadera, es decir la apertura de las piernas a la hora de andar. Como queremos que el robot ande con las piernas cerradas para obtener una mayor estabilidad éste estará fijado siempre a 0. Si por algún motivo queremos modificar este valor en un futuro se podrá hacer de forma sencilla y directa ya que este valor no varía a lo largo de la marcha sino que se mantiene fijo.

En la figura 5.2 tenemos una representación gráfica de los valores obtenidos en la captura del movimiento de NaoQi. Nos ayudaremos de estas gráficas junto con los valores obtenidos en la captura para crear nuestras propias funciones.

Tras el estudio de la tesis doctoral de Juan González Gómez [1], presentada en 2008, sobre robótica modular y locomoción hemos decido basar nuestra modelización en una de las ideas presentadas. En su tesis Juan González trata de simplificar el control de la locomoción de robots ápodos mediante el uso de funciones sinusoidales. A partir del diseño de un generador de movimientos basado en este tipo de ondas es capaz de generar diferentes tipos de locomoción variando únicamente los parámetros de una onda sinusoidal (frecuencia, amplitud y fase).

Debido a los grandes resultados en robots ápodos de la aplicación de ondas sinusoidales para la generación de movimientos decidimos aplicar esta idea a nuestro robot humanoide. De esta manera simplificaremos la locomoción compleja de este robot a un número reducido de parámetros. No obstante esta idea no es directamente aplicable debido a nuestro robot debido a sus características complejas. Como podemos ver en la figura 5.2 los movimientos que describen los actuadores, en la mayoría de los casos, no corresponden a ondas sinusoidales. Sin embargo, trataremos de modelizar estos recorridos con funciones que queden caracterizadas con los mismos parámetros que una onda sinusoidal.

Una onda sinusoidal (figura 5.6) es una señal analógica que oscila a lo largo de todo su dominio. Un onda sinusoidal queda caracterizada por:

- **Amplitud:** es la variación máxima de la onda.
- **Frecuencia:** es el número de veces que se repite un ciclo en un segundo.
- **Fase:** es el ángulo de fase inicial.

[1] http://www.iearobotics.com/personal/juan/doctorado/cube-revolutions/index.html

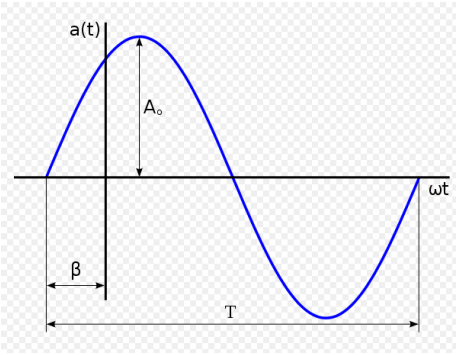


Figura 5.6: Onda sinusoidal

Si la funcion de una onda sinusoidal es $a(t) = A_0 * sin(\omega t + \beta)$, A_0 sería la amplitud, ω sería la frecuencia y β la fase inicial.

Tenemos que añadir un nuevo parámetro a esta función, el desplazamiento de la onda. Las ondas sinusoidales normalmente oscilan entre $-A_0$ y A_0 tomando como referencia de oscilación la recta y=0. En nuestro caso, sin embargo, las ondas no van a oscilar alrededor de y=0 sino que van a estar desplazadas. De modo que la función quedaría de la siguiente manera:
$a(t) = A_0 * sin(\omega t + \beta) + \gamma$, donde, en este caso, la γ representa el ***desplazamiento*** de nuestra función.

Si conseguimos utilizar esta nueva función para modelizar el recorrido de los actuadores durante la caminata del robot solamente tendremos 4 parámetros por actuador: la amplitud, la frecuencia, la fase de la onda y el desplazamiento. Como debemos fijar valores en los 10 actuadores que intervienen en la marcha cada uno tendrá una función de este tipo, con lo que tendremos *40 parámetros* para generar una marcha en el robot Nao, con lo que hemos reducido el número de parámetros con respecto a la forma de andar anterior que tenía 120.

Frecuencia fundamental

Todos los actuadores que intervienen en la marcha del robot tienen que estar perfectamente sincronizados y coordinados para que la locomoción sea correcta, por ello la frecuencia de todos los actuadores debe ser la misma ya que el movimiento debe empezar y terminar al mismo tiempo. De esta forma tendremos 3 parámetros independientes en cada función (fase, amplitud, y desplazamiento) y un parámetro común en todos que es la frecuencia, por lo que tendremos $30 + 1 = 31$ *parámetros*.

Movimientos simétricos

Como hemos introducido anteriormente, el paso de un robot humanoide es simétrico por lo que podemos obviar uno de los lados del robot. Si, por ejemplo, obviamos el lado

izquierdo (I) podemos fijar su movimiento desde el derecho (D) sumándole a la función que define su movimiento un cierto desfase. Con ello volvemos a reducir el número de parámetros, teniendo ahora 5 actuadores * 3 parámetros por actuador 15 + la frecuencia común a todos tenemos *16 parámetros.*

Funciones periódicas para el modelado de los actuadores

Una vez reducido el espacio de parámetros y fijados los criterios para la modelización del movimiento de los actuadores situados en el lado derecho empezaremos a crear las funciones necesarias. Al ser funciones cíclicas basadas en ondas sinusoidales sólo definiremos estas funciones entre 0 y 2π. Para extender estas funciones a todo el dominio lo que haremos será, antes de evaluar la función, obtener su valor correspondiente en este intervalo. Para ello hemos creado un pequeño bucle (código 5.8 que obtiene el valor en radianes si nuestra función estuviese definido en todo el dominio ($x = \omega t + \beta$) y seguidamente lo reduce a su valor correspondiente entre 0 y 2π.

```
my_frequency= (float)2*PI/(frequency); // obtenemos el valor de la
    //frecuencia seleccionada en Hz (frequency es la frecuencia en segundos)
x = (my_frequency * time_pos) + phase; // calculamos el valor en radianes
    //que vamos a evaluar

while (x < 0){
    x = (2* PI) + x;
}
while (x > 2*PI){
    x = x -(2*PI);
}
```

Listing 5.8: Bucle desarrollado para convertir cualquier valor del dominio en nuestro intervalo definido. De esta manera x sustutuirá a $\omega t + \beta$.

- **Pitch de la cadera** Como podemos ver en la gráfica de la figura 5.7, el recorrido que realiza el actuador es muy similar al de una onda oscilatoria, únicamente hay un intervalo en el que difiere, justamente donde coincide con el valor máximo de la gráfica. Este intervalo lo hemos aproximado a $(\frac{3\pi}{4}, \pi)$. Este actuador queda modelizado con la siguiente función:

$$f(t) = \begin{cases} \frac{6}{11}Asin(x) + \gamma & si \quad x \leq \frac{3\pi}{4} \\ A + \gamma & si \quad \frac{3\pi}{4} < x < \pi \\ \frac{6}{11}Asin(x) + \gamma & si \quad x \geq \pi \end{cases}$$

  Como comprobación gráfica entre el recorrido del robot y el conseguido mediante nuestra función hemos creado una gráfica conjunta donde se puede apreciar el resultado (figura 5.8)

  Como se puede apreciar en la gráfica de la figura figura 5.9 el desfase entre en actuador derecho y el izquierdo es de π, es decir, la mitad de la duración del paso.

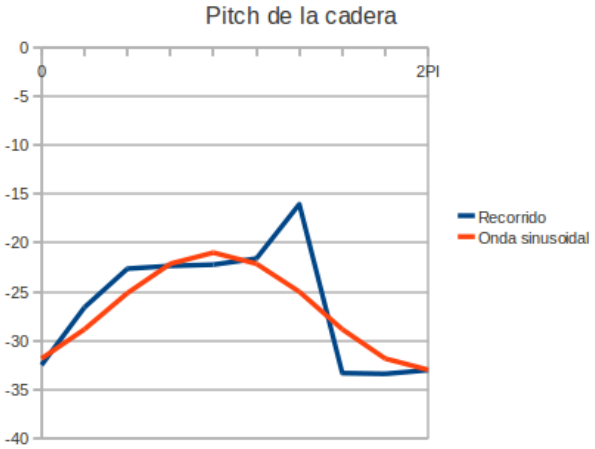


Figura 5.7: Gráfica con el recorrido del actuador pitch de la cadera

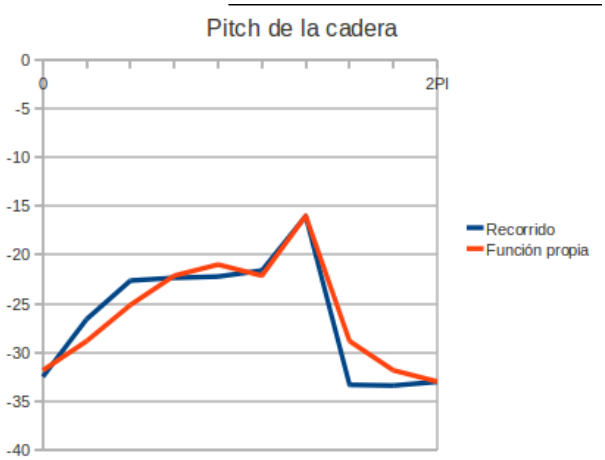


Figura 5.8: Gráfica con la comparación de recorridos del actuador pitch de la cadera (izda/dcha)

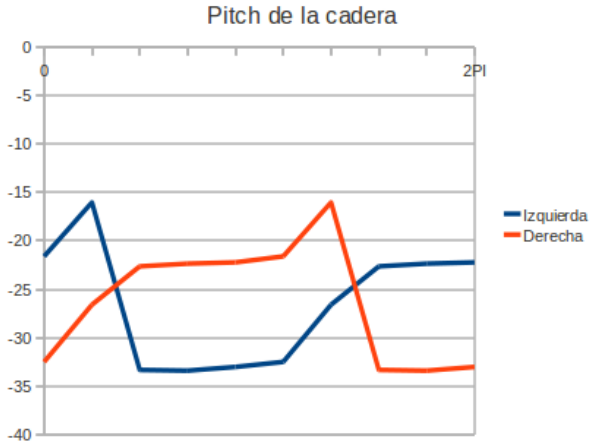


Figura 5.9: Gráfica con la comparación de recorridos del actuador pitch de la cadera

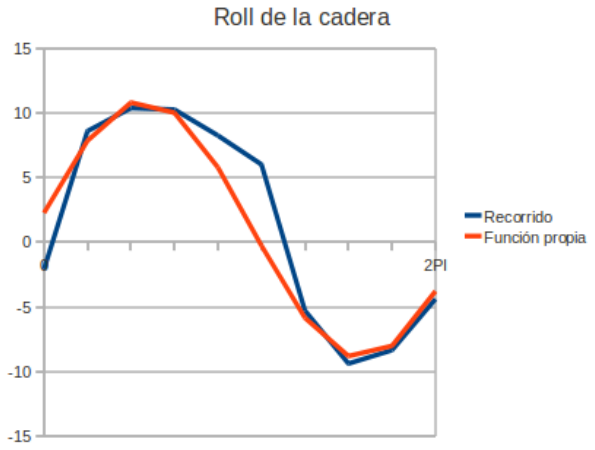


Figura 5.10: Gráfica con el recorrido del actuador roll de la cadera

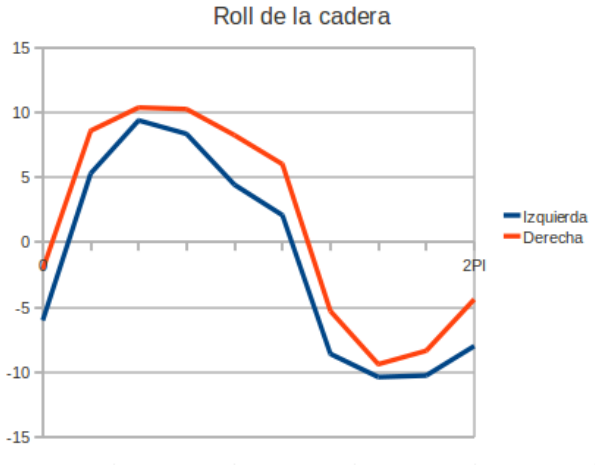


Figura 5.11: Gráfica con el recorrido del actuador roll de la cadera (izda/dcha)

- **Roll de la cadera** Según refleja la trayectoria del movimiento de este actuador en la gráfica de la figura 5.10, su comportamiento es muy similar al de una onda sinusoidal. Por este motivo hemos decidido usar la función seno sin necesidad de realizar ningún cambio para modelizar este actuador.

  Tal y como se aprecia en la gráfica de la figura 5.11 el desfase entre el actuador izquierdo y el derecho es 0. Así mismo podemos ver que, aunque el recorrido de los dos actuadores es prácticamente el mismo, existe una cierta variación del desplazamiento (γ). Esto sucede porque los rangos de actuación de los motores no es el mismo, mientras que el motor izquierdo va de $[-45, 25]$ el derecho va de $[-25, 45]$, esta variación hace que el movimiento esté compensado.
  Hemos estimado que esta variación es de $5\,°$, por ello la función utilizada para el actuador derecho será $f(t) = A_0 sin(\omega t + \beta_0) + \gamma$ y para el izquierdo será $f(t) = A_1 sin(\omega t + \beta_1) + \gamma - 5$

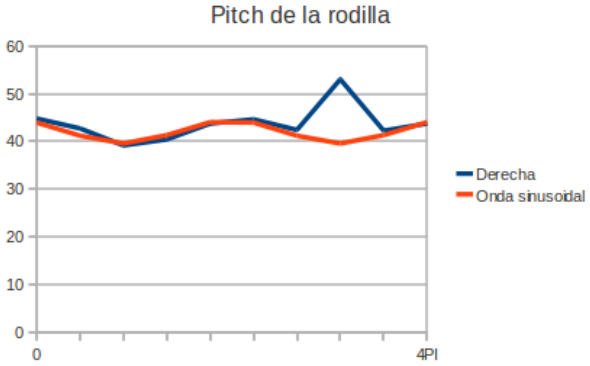


Figura 5.12: Gráfica con el recorrido del actuador pitch de la rodilla

- **Pitch de la rodilla** Tal y como se aprecia en la gráfica de la figura 5.12, el recorrido que realiza el actuador es muy similar al de una onda sinusoidal, sin embargo existe un intervalo en el que no coinciden estas dos gráficas, justamente en el segundo mínimo de la función. A diferente del resto, esta función no estará definida entre $[0, 2\pi]$ sino entre $[0, 4\pi]$ por lo que su frecuencia será el doble que la del resto de los actuadores. Esto no supone mayor problema ya que sólo debemos multiplicar la frecuencia del sistema por una constante entera (en este caso 2). Como el dominio es distinto debemos cambiar también el algoritmo que busca el valor correspondiente en el intervalo definido de la función (código 5.9).

```
my_frequency= (float)4*PI/(frequency); // obtenemos el valor de la
    //frecuencia seleccionada en Hz (frequency es la duración total
    // del paso en segundos)
x = (my_frequency * time_pos) + phase; // calculamos el valor en radianes
    //que vamos a evaluar

while (x < 0){
    x = (4* PI) + x;
}
while (x > 4*PI){
    x = x -(4*PI);
}
```

Listing 5.9: Bucle desarrollado para convertir cualquier valor del dominio al intervalo definido para el pitch de la rodilla. De esta manera x sustutuirá a $\omega t + \beta$.

La función elegida para modelar el recorrido de este actuador es la siguiente:

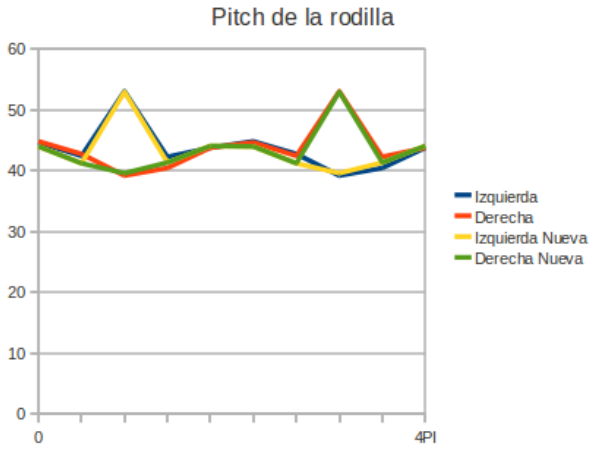


Figura 5.13: Gráfica la comparación de recorridos del actuador pitch de la rodilla

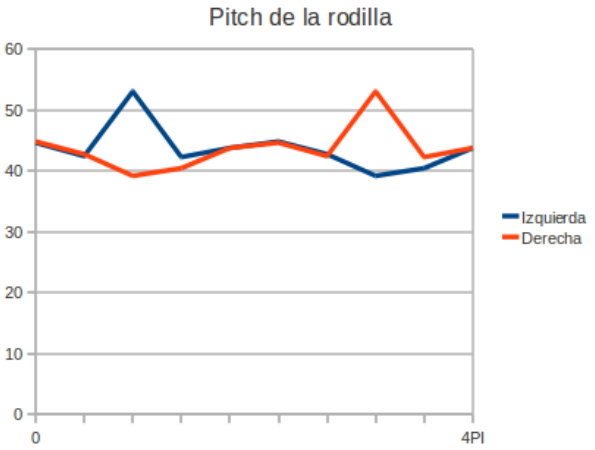


Figura 5.14: Gráfica con el recorrido del actuador pitch de la rodilla (izda/dcha)

$$f(t) = \begin{cases} 0{,}22Asin(x) + \gamma & si \quad x \leq \frac{7\pi}{2} - \frac{\pi}{4} \\ \frac{x - \frac{13\pi}{4}}{\frac{\pi}{4}}((A+\gamma) - (0{,}22Asin(\frac{13\pi}{4}) + \gamma)) + \\ +0{,}22Asin(\frac{13\pi}{4}) + \gamma & si \quad \frac{13\pi}{4} < x < \frac{7\pi}{2} \\ \frac{x - \frac{7\pi}{2}}{\frac{\pi}{4}}(0{,}22Asin(\frac{15\pi}{4}) + \gamma - (A+\gamma)) + \\ +A + \gamma & si \quad \frac{7\pi}{2} \leq x < \frac{15\pi}{4} \\ 0{,}22Asin(x) + \gamma & si \quad x \geq \frac{7\pi}{2} + \frac{\pi}{4} \end{cases}$$

Podemos ver la comparación entre el recorrido original y los valores fijados por nuestra función en la gráfica de la figura 5.13

Tal y como refleja la gráfica de la figura 5.14, el desfase entre en actuador izquierdo y el derecho es la mitad de la duración del paso, como en este caso la función esta definida entre $[0, 4\pi]$ el valor del desfase será 2π.

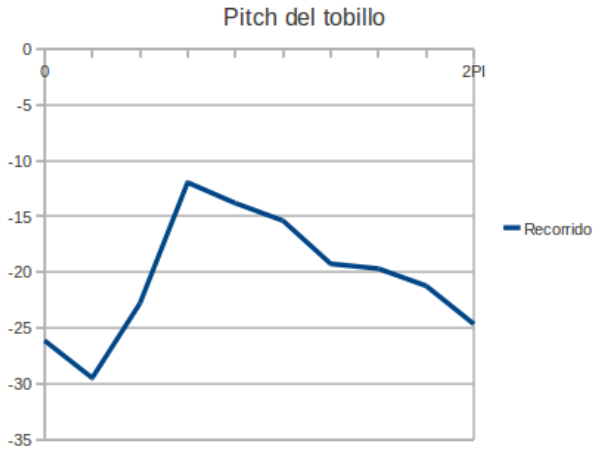


Figura 5.15: Gráfica con el recorrido del actuador pitch del tobillo

- **Pitch del tobillo** El pitch del tobillo es un caso particular ya que, como podemos observar en la gráfica de la figura 5.15, su recorrido no se parece al de una onda sinusoidal. Por ello hemos definido la función dividiéndola en dos partes y usando ecuaciones de rectas. Para que se ajusten al funcionamiento de una onda estas funciones deben depender de los mismos parámetros que las funciones anteriores (A, $\beta, \omega y \gamma$), por ello la función elegida para modelizar el recorrido de este actuador es la siguiente:

$$f(t) = \begin{cases} \frac{x}{\frac{2\pi}{5}} 2A + A + \gamma & si \quad x \leq \frac{2\pi}{5} \\ \frac{x-(\frac{2\pi}{5})}{\frac{8\pi}{5}}(-2A) + (\gamma + A) & si \quad \frac{2\pi}{5} < x < \pi \end{cases}$$

  Al igual que ocurre con las funciones anteriores, está definida en el intervalo $[0, 2\pi]$, para que esta función sea cíclica necesitamos aplicar también el bucle (código 5.8 para encontrar el valor correspondiente en el intervalo definido y poder aplicar el desfase β).

  Podemos ver el resultado de esta función en la gráfica de la figura 5.16, donde aparecen el recorrido de los actuadores siguiendo el movimiento base y el recorrido utilizando nuestra función. Así mismo, también podemos observar en esta misma gráfica que el desfase entre el actuador izquierdo y el derecho es 2π.

- **Roll del tobillo** El roll del tobillo es un caso exactamente igual al del roll de la cadera. El recorrido de los actuadores es muy similar al de una sinusoidal (figura 5.17), el actuador izquierdo y el derecho están en fase y existen una cierta variación en el desplazamiento vertical de la función (γ) entre ambos. Esta variación se produce, al igual que en el roll de la cadera, porque los actuadores izquierdo y derecho no tienen el mismo rango de actuación, de hecho tienen los mismo rangos que el roll de la cadera, $[-45, 25]$ para el actuador izquierdo y $[-25, 45]$ en el caso del derecho. Como era de esperar el valor de esa variación es el mismo que en el caso de la cadera, $5\,°$.

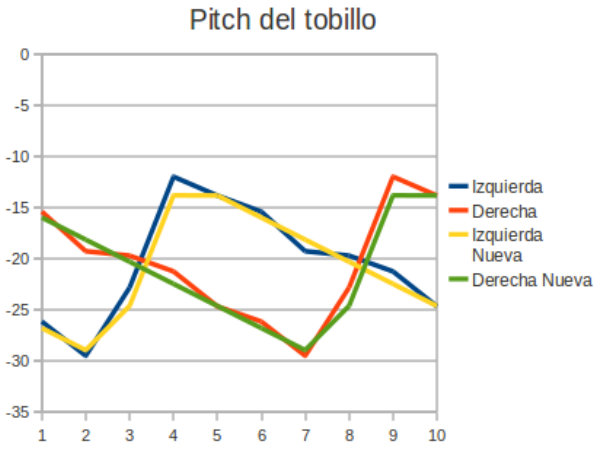


Figura 5.16: Gráfica la comparación de recorridos del actuador pitch del tobillo (izda/decha)

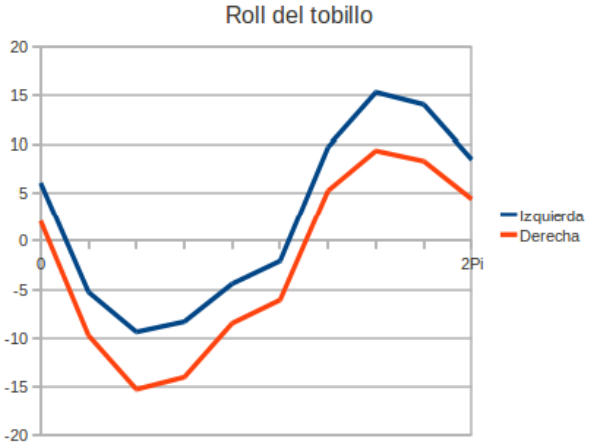


Figura 5.17: Gráfica con el recorrido del actuador roll del tobillo

Por lo explicado en el párrafo anterior las funciones que describen el movimiento serán:

- En el caso del actuador derecho: $f(t) = A_0 sin(\omega t + \beta_0) + \gamma$
- En el caso del actuador izquierdo: $f(t) = A_1 sin(\omega t + \beta_1) + \gamma - 5$

Podemos ver los resultados de aplicar esta función en la gráfica de la figura 5.18.

Una vez con el recorrido de todos los actuadores modelizados (6.4) tenemos una primera forma de andar parametrizada con un total de *16 parámetros:*

1. Frecuencia (γ), común para todos los actuadores.

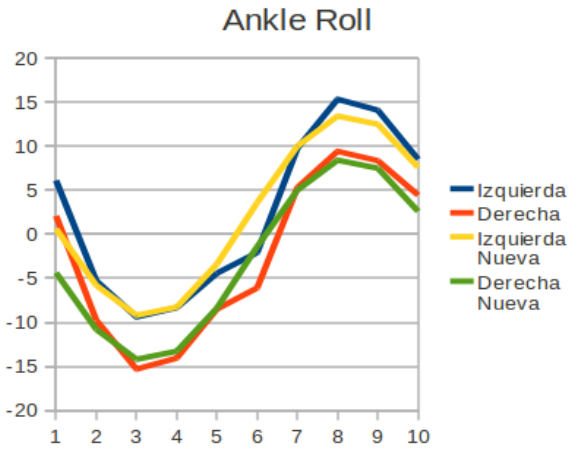


Figura 5.18: Gráfica la comparación de recorridos del actuador roll del tobillo

2. Amplitud del pitch de la cadera.
3. Fase (β) del pitch de la cadera.
4. Desplazamiento vertical (γ) del pitch de la cadera.
5. Amplitud del roll de la cadera.
6. Fase (β) del roll de la cadera.
7. Desplazamiento vertical (γ) del roll de la cadera.
8. Amplitud del pitch de la rodilla.
9. Fase (β) del pitch de la rodilla.
10. Desplazamiento vertical (γ) del pitch de la rodilla.
11. Amplitud del pitch del tobillo.
12. Fase (β) del pitch del tobillo.
13. Desplazamiento vertical (γ) del pitch del tobillo.
14. Amplitud del roll del tobillo.
15. Fase (β) del roll del tobillo.
16. Desplazamiento vertical (γ) del roll del tobillo.

Hasta ahora tenemos modelizada la caminata de un robot bípedo mediante 5 funciones con las que, debidamente configuradas, conseguimos que el robot ande. Uno de los parámetros de estas funciones es la fase inicial de la onda (β), que indica el momento inicial de cada uno de los actuadores. Según lo hemos modelado estas fases son totalmente independientes, por ejemplo, para conseguir la forma de andar que hemos

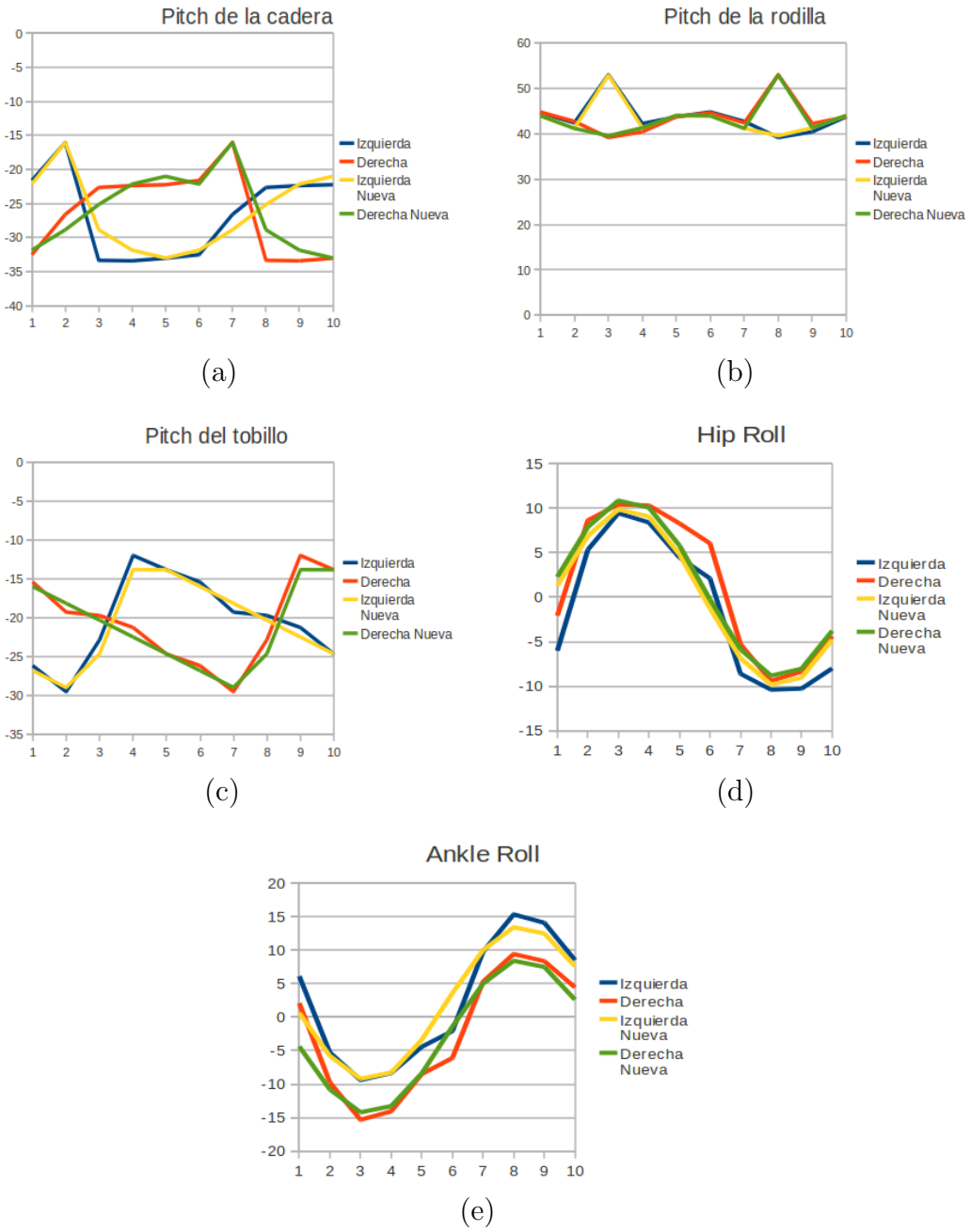


Figura 5.19: Gráficas de los actuadores que intervienen en la marcha

usado como base (la caminata en la que coinciden el movimiento inicial utilizando secuencias fijas, capturando la forma de andar de NaoQi, y el modelizado por nosotros) las fases deben ser las mostradas en la tabla 5.3.1.

Motor	**Fase**
Pitch de la cadera	$-\frac{\pi}{2}$
Roll de la cadera	$\frac{\pi}{3}$
Pitch de la rodilla	$\frac{12\pi}{5}$
Pitch del tobillo	$\frac{11\pi}{10}$
Roll del tobillo	$\frac{2\pi}{3}$

Cuadro 5.2: Tabla con los valores de las fases para el movimiento inicial

Fases relativas

Estos parámetros siguen siendo demasiados para comenzar a hacer una búsqueda teniendo en cuenta el elevado número de parámetros y sus rangos, por ello debemos encontrar alguna forma de reducirlos o crear algún tipo de dependencia para poder simplificarlos.

Esta forma de definir los movimientos con fases independientes es muy poco práctica ya que, de esta manera, conseguir un movimiento sincronizado y coordinado es muy complicado, por ello hemos decidido hacer que todas las fases dependan de una previamente fijada. El actuador elegido para ser el punto de partida es el pitch de la cadera. Para crear esta dependencia debemos hacer un pequeño cambio en las funciones anteriores y es que a la fase de cada función le debemos sumar también la fase del pitch de la cadera ($\beta' = \beta + \beta_{pitch-cadera}$), por lo que ya no serán fases independientes sino *fases relativas con respecto al pitch de la cadera*. Esta fase va a ser la que determine la posición inicial del movimiento, es decir, podemos determinar en qué posición queremos que empiece a andar el robot simplemente cambiando el valor esta fase. Precisamente por eso es uno de los parámetros que podemos obviar a la hora de buscar nuevos movimientos ya que no encontraremos ningún cambio significativo en el movimiento al variarlo, lo único que conseguiremos es que cambie la posición inicial. Tras la eliminación de la fase independiente tendremos *15 parámetros*.

Balanceo

Una de las simplificaciones que podemos hacer de forma bastante sencilla es fijar una dependencia directa en los actuadores que realizan el balanceo en la marcha del robot. Llamamos balanceo al movimiento que realiza el robot para compensar el peso a la hora de levantar una de las patas durante la marcha, podemos ver este movimiento en la figura 5.3.1. el balanceo intervienen únicamente el roll de la cadera y el roll del tobillo y son justamente los dos actuadores que hemos modelizado directamente son una onda sinusoidal. Variar las fases de estos movimientos significa que la marcha no

este acompasada y sea descoordinada por lo que podemos fijar una dependencia directa con la fase de la cadera de forma que sólo varíen las fases de los roll cuando varíe la fase de la cadera. Para conseguir un balanceo más suave o más pronunciado podemos fijar un movimiento base con unas amplitudes determinadas (por ejemplo, los valores que corresponden a la forma de andar con secuencias fijas) y multiplicar el valor de estas amplitudes por una constante. Con esto parametrizamos todo el movimiento del balanceo con una sola variable, la constante por la que vamos a multiplicar nuestro movimiento base, a este valor le hemos llamado *amplitud de balanceo*. De esta manera conseguimos eliminar las dos amplitudes, que dependerán del nuevo parámetro, las dos fases, que serán directamente dependientes de la fase del pitch de la cadera para que el balanceo sea siempre acompasado y los dos desplazamientos, ya que al igual que las amplitudes serán fijas y situarán a los actuadores en la base del balanceo. Tras la aplicación de esta idea volvemos a reducir el número de parámetros de nuestra caminata, quitando 2 actuadores, de esta manera nos quedan *10 parámetros*:

Parámetro	**min**	**max**
1. Frecuencia (γ), común para todos los actuadores.	0	-
2. Amplitud del pitch de la cadera.	0	62
3. Desplazamiento (γ) del pitch de la cadera.	-100	25
4. Amplitud del pitch de la rodilla.	0	65
5. Fase (β) del pitch de la rodilla.	-2π	2π
6. Desplazamiento (γ) del pitch de la rodilla.	0	130
7. Amplitud del pitch del tobillo.	0	60
8. Fase (β) del pitch del tobillo.	$-\pi$	π
9. Desplazamiento vertical (γ) del pitch del tobillo.	-75	45
10. Amplitud de balanceo	0	100

Cuadro 5.3: Tabla con los parámetros finales de la modelización del movimiento

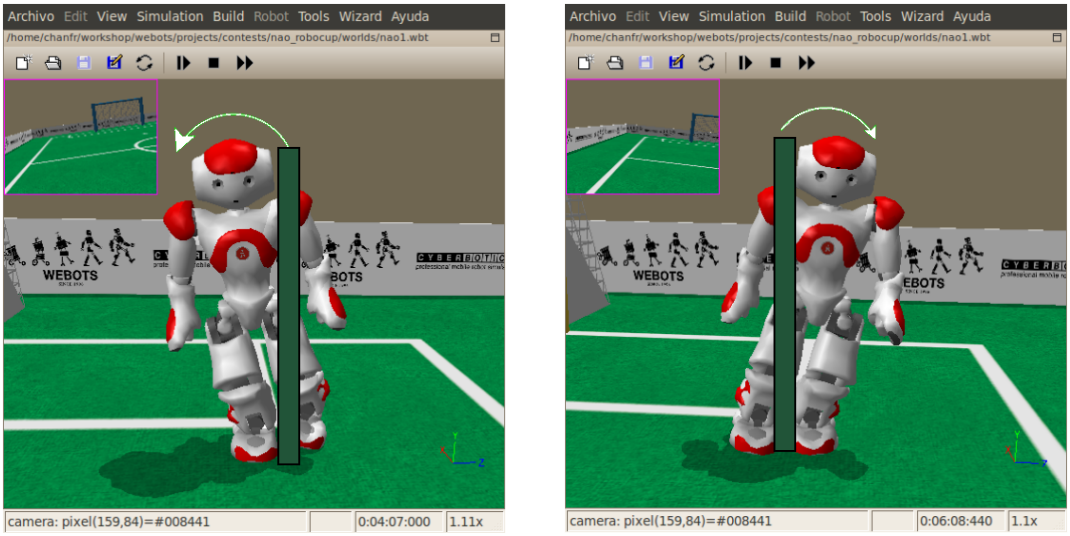
De esta forma con *únicamente 10 parámetros* tenemos caracterizada la caminata del robot humanoide Nao.

5.3.2. Movimiento de los brazos en la caminata parametrizada

Para incrementar la estabilidad del robot y que el movimiento sea más vistoso hemos parametrizado también el movimiento de los brazos del humanoide Nao. Para ello hemos seguido la misma idea que en el resto de actuadores pero esta vez sin tener ningún movimiento previo sobre el que basarnos sino que lo definimos nosotros mismos. Los actuadores que hemos decidido que intervengan en el movimiento de los brazos en la locomoción del Nao son el pitch y el roll del hombro y el yaw del codo. En la tabla 5.3.2 podemos encontrar los datos referentes a estos actuadores.

Para modelizar el movimiento de estos actuadores hemos utilizado directamente ondas sinusoidales, cada una de ellas con sus propios parámetros. Al igual que hicimos con el balanceo este movimiento lo fijaremos directamente sobre la fase del actuador pitch de la cadera. Para simplificar aún más este movimiento lo que haremos será crea un movimiento base con unas amplitudes, fases y desplazamientos fijados, de esta manera simplificamos el número de parámetros a fijar para realizar este movimiento

[H]



(a) (b)

Figura 5.20: Balanceo en la forma de andar parametrizada

Motor	min	max	Velocidad Nominal	Par nominal
LShoulderPitch	-120	120	351.77 °/s (7.03°/20ms)	0.57 Nm
RShoulderPitch	-120	120	351.77 °/s (7.03°/20ms)	0.57 Nm
LShoulderRoll	0	95	412.19 °/s (8.24°/ 20ms)	2.61 Nm
RShoulderRoll	-95	0	412.19 °/s (8.24°/ 20ms)	2.61 Nm
LElbowYaw	-120	120	351.77 °/s (7.03°/20ms)	0.57 Nm
RElbowYaw	-120	120	351.77 °/s (7.03°/20ms)	0.57 Nm

Cuadro 5.4: Tabla descriptiva de los actuadores

de brazos. Si queremos variar el recorrido realizado los brazos durante la marcha sólo tendremos que multiplicar cada una de las amplitudes de los brazos por una constante al igual que sucede con la amplitud de balanceo, en este caso esa amplitud la llamaremos *amplitud de recorrido de los brazos.*

A continuación detallaremos las funciones con las cuales hemos modelado cada uno de los actuadores que intervienen en el movimiento de los brazos durante la marcha del robot.

1. Pitch del hombro

   - Actuador izquierdo:
     $f(t) = A_{pitch\_hombro} \cdot A_{recorrido} \cdot sin(\frac{2\pi}{T} \cdot t + (\beta_{pitch\_hombro} + \beta_{pitch\_cadera}) + \gamma_{pitch\_hombro}$
   - Actuador derecho:
     $f(t) = A_{pitch\_hombro} \cdot A_{recorrido} \cdot sin(\frac{2\pi}{T} \cdot t + (\beta_{pitch\_hombro} + \beta_{pitch\_cadera} + \pi) + \gamma_{pitch\_hombro}$

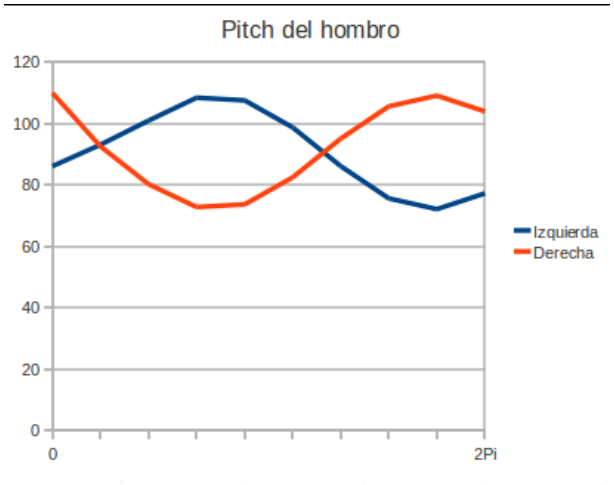


Figura 5.21: Gráfica con el recorrido del actuador pitch del hombro

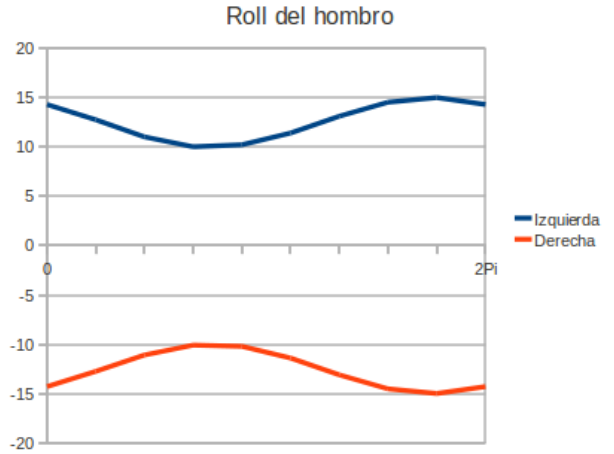


Figura 5.22: Gráfica con el recorrido del actuador roll del hombro

2. Roll del hombro

   - Actuador izquierdo:
     $f(t) = A_{roll\_hombro} \cdot A_{recorrido} \cdot sin(\frac{2\pi}{T} \cdot t + (\beta_{roll\_hombro} + \beta_{pitch\_cadera}) + \gamma_{roll\_hombro}$
   - Actuador derecho:
     $f(t) = -A_{roll\_hombro} \cdot A_{recorrido} \cdot sin(\frac{2\pi}{T} \cdot t + (\beta_{roll\_hombro} + \beta_{pitch\_cadera}) + \gamma_{roll\_hombro}$

3. Yaw del codo

   - Actuador izquierdo:
     $f(t) = A_{yaw\_codo} \cdot A_{recorrido} \cdot sin(\frac{2\pi}{T} \cdot t + (\beta_{yaw\_codo} + \beta_{pitch\_cadera}) + \gamma_{yaw\_codo}$
   - Actuador derecho:
     $f(t) = -A_{yaw\_codo} \cdot A_{recorrido} \cdot sin(\frac{2\pi}{T} \cdot t + (\beta_{yaw\_codo} + \beta_{pitch\_cadera}) + \gamma_{yaw\_codo}$

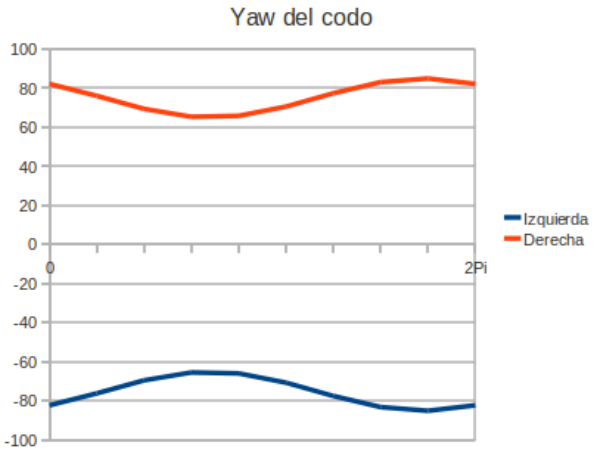


Figura 5.23: Gráfica con el recorrido del actuador yaw del codo

Los parámetros de amplitud, fase y desplazamiento son valores fijos no son variables, esta función depende de la amplitud de recorrido de los brazos y la fase del pitch de la cadera. Los valores de los parámetros fijos están reflejados en la tabla 3.

Parámetro	**valor**
A_{pitch_hombro}	0.37
β_{pitch_hombro}	π
γ_{pitch_hombro}	90.5
A_{roll_hombro}	0.05
β_{roll_hombro}	0
γ_{roll_hombro}	-12.5
A_{yaw_codo}	0.2
β_{yaw_codo}	π
γ_{yaw_codo}	75

Cuadro 5.5: Tabla con los valores de los parámetros fijos en los movimientos de los brazos durante la marcha.

Para la comprobación dinámica de todo lo desarrollado en este capítulo hemos utilizado algunas de las funcionalidades que nos ofrece la aplicación NaoOperator 5.24 que dan soporte a estos modos de andar parametrizados. De esta manera directamente desde el interfaz gráfico podemos variar los parámetros que deseemos para modificar la caminata del robot Nao, activar o desactivar los movimientos de los brazos durante la marcha del robot, variar el número de fotogramas en los que se va a dividir nuestra caminata, la frecuencia principal de la marcha (en segundos)...

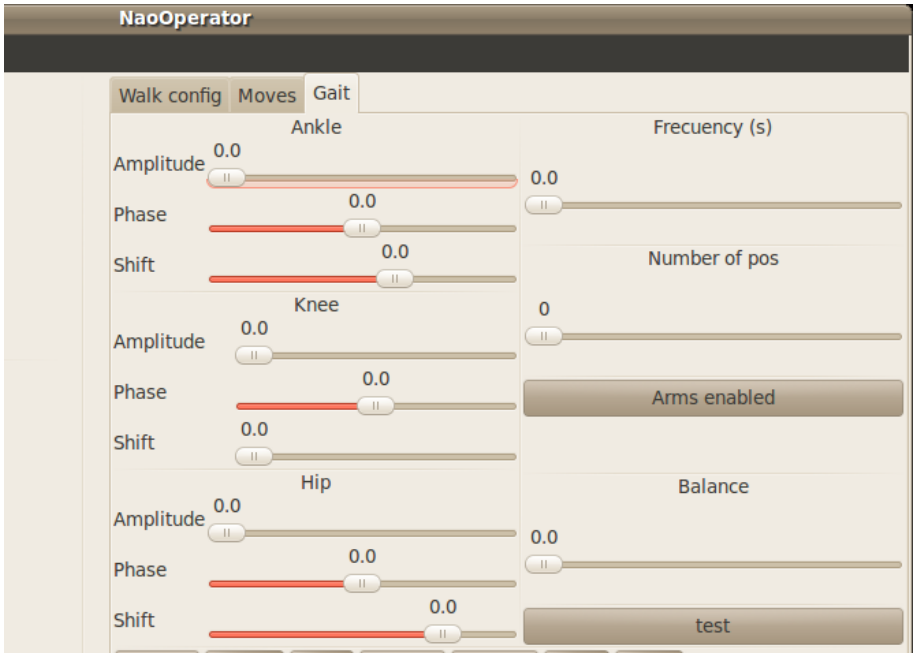


Figura 5.24: Interfaz gráfico del NaoOperator con soporte para los movimientos parametrizados

Capítulo 6
Búsqueda de caminata óptima

Con la modelización explicada en el capítulo anterior hemos conseguido caracterizar un gran número de formas de andar, todas las que somos capaces de crear. Es obvio que esto no cubre todas las posibles caminatas del Nao, pero sí nos ofrece una gran variedad donde buscar. La mejor forma de realizar la búsqueda de la marcha óptima es realizando una búsqueda sistemática que recorra todas las posibilidades, sin embargo esta opción no es factible debido al gran número de caminatas que es capaz de generar nuestro modelo.

En este capítulo explicaremos el espacio de búsqueda de nuestro modelo, los métodos de evaluación de caminatas utilizados, los estudios realizados sobre los movimientos encontrados y los mejores resultados obtenidos.

6.1. Espacio de búsqueda

Como hemos explicado en el capítulo anterior, hemos conseguido modelar la forma de andar del Nao con 11 parámetros (tabla 5.3.1, aunque a la hora de realizar búsquedas con estos parámetros podemos eliminar uno de ellos y es justamente la pase del pitch de la cadera. Todo el movimiento depende de él y es el parámetro que fija en qué punto comienza el movimiento, por ello el movimiento obtendrá la misma puntuación independientemente de en qué punto se empiece a mover ya que el resto de movimientos estarán en las mismas condiciones. Por ello tendremos realmante un espacio de búsquedas con *10 parámetros*, donde también están reflejados los rangos de cada uno de ellos.

La modelización que hemos utilizado para reducir el espacio de parámetros parte de un conjunto infinito de posibles movimientos realizados directamente accediendo a cada uno de los motores del robot, puesto que no buscamos cualquier movimientos sino una caminata para el Nao hemos reducido las posibilidades a un subconjunto de caminatas modelizadas con secuencias de movimientos intermedios, aplicando también la idea de repeticiones en la locomoción (pasos). Como este espacio sigue siendo demasiado grande para poder encontrar algún resultado de forma sencilla seguimos reduciendo el espacio aplicando los movimientos modelizados con ondas acopladas. Con este último paso conseguimos reducir en gran medida el número de parámetros, por consiguiente el espacio de búsquedas será menor. Este procedimiento lo podemos ver de forma gráfica

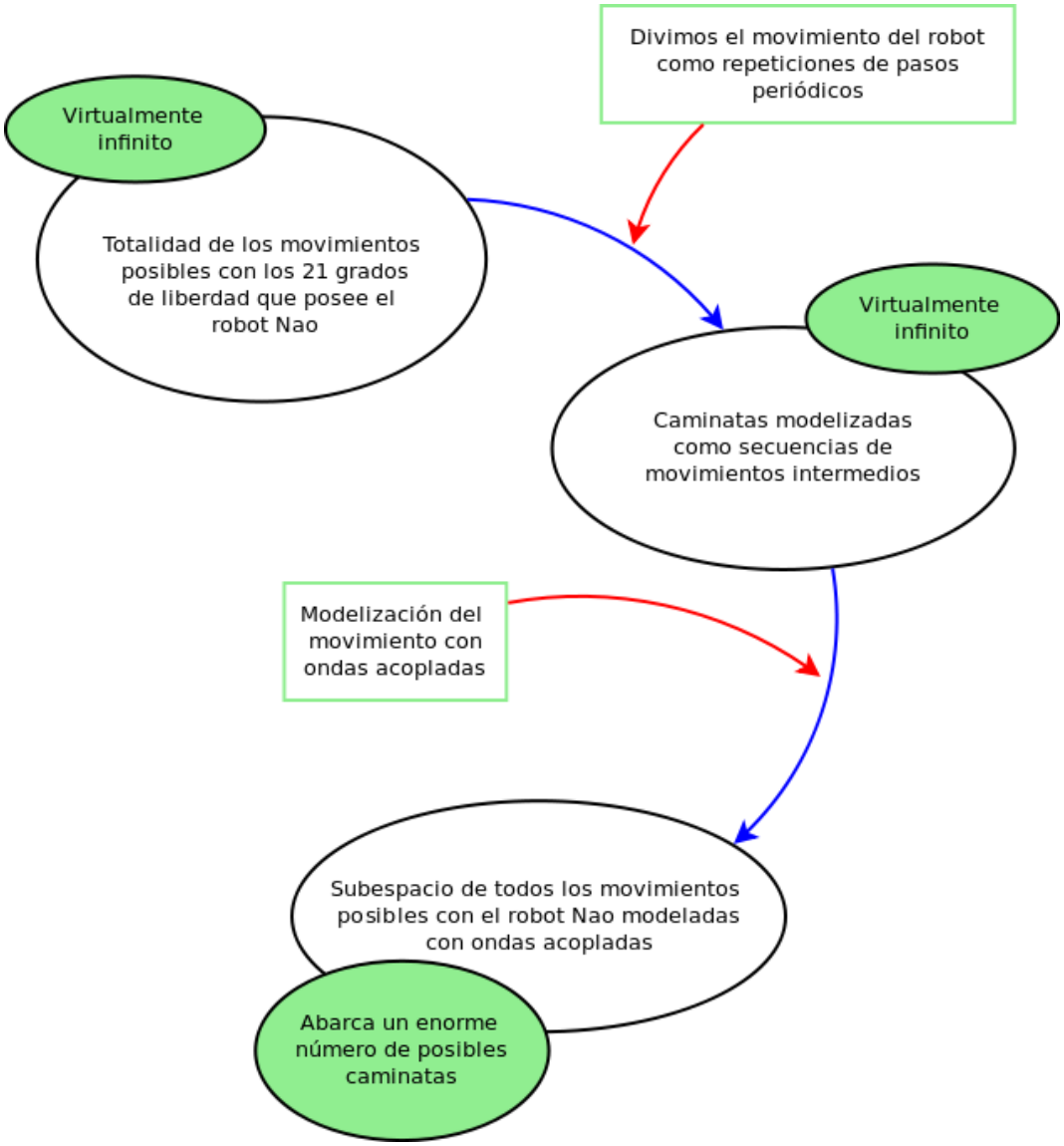


Figura 6.1: Gráfico resumen de la modelización de la caminata.

en la figura 6.1.

De todos las caminatas que describe nuestro modelo hay muchas de ellas que no tienen sentido, nos llevan a la caída del robot, otras generan pasos largos y lentos o rápidos con pasos cortos...

Para hacer una pequeña estimación del espacio en que vamos a trabajar discretizaremos todos los rangos de los parámetros a un rango acotado de valores como, por ejemplo, el siguiente:

- 62 niveles para la amplitud del pitch de la cadera.
- 125 niveles para el desplazamiento del pitch de la cadera.
- 65 niveles para la amplitud del pitch de la rodilla.
- 12 niveles para la fase del pitch de la rodilla.

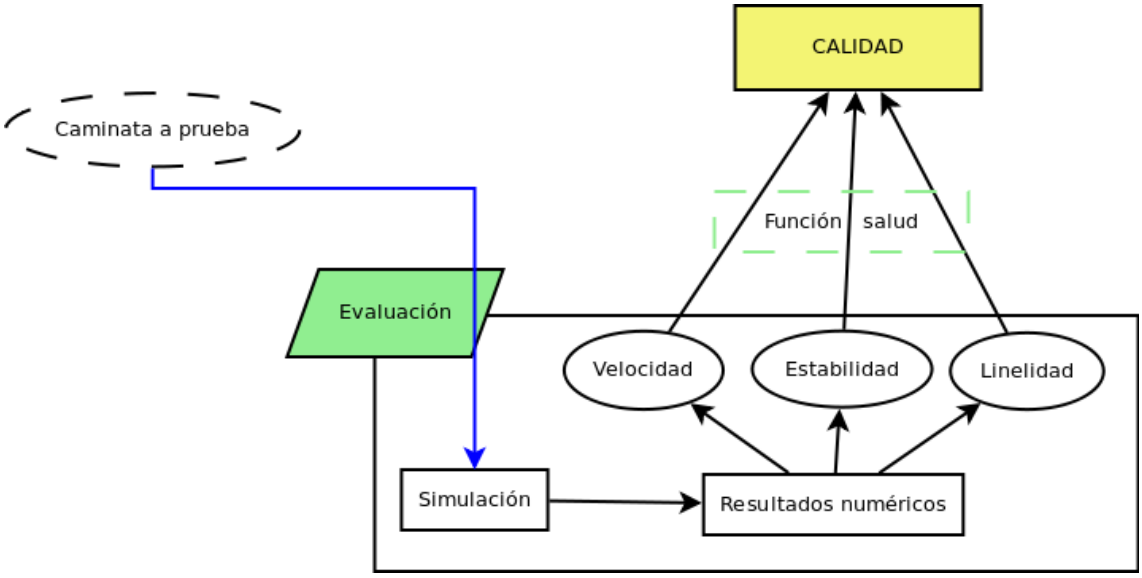


Figura 6.2: Gráfico resumen de la evaluación de la caminata.

- 130 niveles para el desplazamiento del pitch de la rodilla.
- 60 niveles para la amplitud del pitch del tobillo.
- 6 niveles para la fase del pitch del tobillo.
- 120 para el desplazamiento del pitch del tobillo.
- 100 niveles para la amplitud de balanceo.

Si hacemos una aproximación de las posibles combinaciones con estos parámetros tendríamos aproximadamente 2^{51} posibles caminatas y esto es fijando la frecuencia a un valor fijo. Si en hacer la evaluación de una caminata nuestra aplicación tarda unos 7 segundos, realizar una búsqueda sistemática significarían, aproximadamente, 71.404.103 años para lograr evaluarlas todas en el mismo ordenador con un solo simulador.

6.2. Evaluación de caminatas

La búsqueda en un espacio tan grande como estamos tratando es complicada, por ello es necesario tener unas ciertas funcionalidades sobre las que basar nuestra búsqueda, necesitamos herramientas para la evaluación de movimientos que nos proporcionen la información necesaria para determinar si una caminata es válida o no lo es. Para la evaluación de caminatas no utilizaremos ningún método analítico debido a la gran complejidad que supone, de modo que utilizaremos un simulador en el que desplegaremos el modelo y caracterizaremos numéricamente si la caminada realizada es buena o mala observando cualidades como estabilidad, velocidad... En la figura 6.2 se puede apreciar una representación gráfica de este procedimiento.

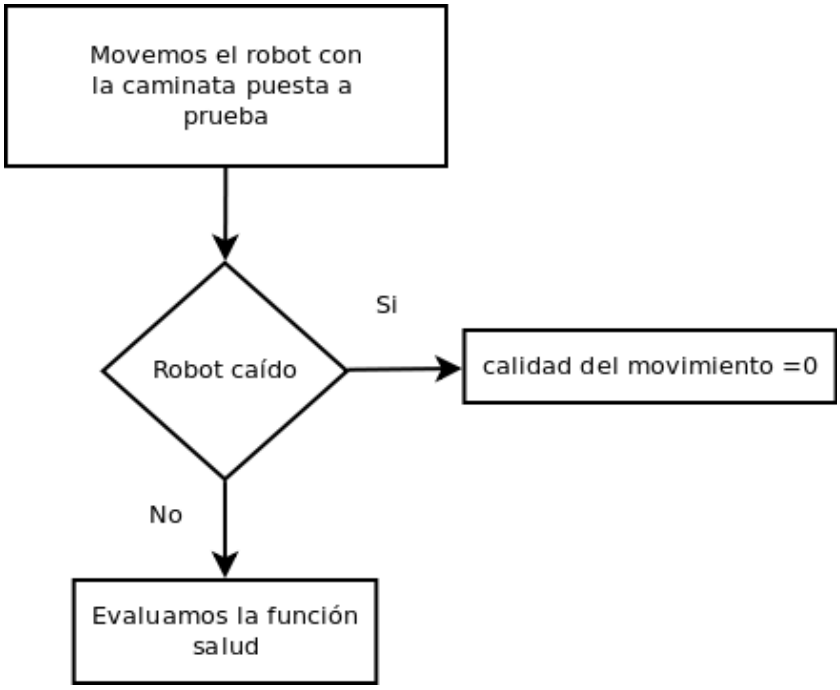


Figura 6.3: Gráfico resumen de la evaluación de la caminata con comprobación de caídas.

Para comprobar el funcionamiento de las posibles caminatas dentro de este espacio de parámetros es necesario realizar una evaluación de movimientos. Como habíamos introducido anteriormente lo que vamos a premiar en una marcha es la velocidad y la estabilidad, otro de los valores a tener en cuenta a la hora de desplazarse con un robot bípedo es la linealidad del movimiento. Es importante para la locomoción de un robot saber si la caminata que está realizando genera algún tipo de desviación sobre su trayectoria, si no es así el robot tendrá que compensar estas variaciones realizando otro tipo de movimientos como, por ejemplo, giros. Para no tener que hacer este tipo de compensaciones haremos este parámetro sea unos de los valores fundamentales a la hora de decidir la calidad del movimiento.

La velocidad, la estabilidad y la linealiadad son lo suficientemente discriminantes como para desechar caminatas no válidas, debemos aclarar que si durante la realización del movimiento el robot cae no se evaluará la función salud y la calidad del movimiento será directamente 0 (figura 6.3).

Para poder realizar esta evaluación de forma iterativa hemos utilizado algunas de las funcionalidades que nos ofrece tanto el NaoBody como el NaoOperator. El conjunto de driver y esquema nos ofrecen una serie de valores como son: posición inicial, posición final, tiempo en realizar el movimiento, estabilidad... Teniendo estos datos podemos calcular la distancia recorrida de manera muy simple utilizando los valores de posición inicial y posición final:

$$distancia = \sqrt{((end\_pos[0] - init\_pos[0])^2) + ((end\_pos[1] - init\_pos[1])^2)}$$

Por otra parte el cálculo de la estabilidad se realiza de forma iterativa mientras la simulación progreasa a partir de la variación de los componentes (x,y) de la posición real del robot (obtenida desde el módulo *posición verdadera* de NaoBody). Para ello en cada iteración que se realice mientras dure el movimiento se realiza el siguiente cálculo, mostrado en el código 6.1:

```
distance=distance+sqrt(((pos_actual[0]-pos_anterior[0])*(pos_actual[0]
    -pos_anterior[0]))+((pos_actual[1]-pos_anterior[1])*(pos_actual[1]
    -pos_anterior[1]))+((pos_actual[2]-pos_anterior[2])*(pos_actual[2]
    -pos_anterior[2])));

pos_anterior[i]=pos_actual[i];
```

Listing 6.1: Iteración para el cálculo de la estabilidad

El cálculo de la linealidad se realiza a partir de la posición inicial y la posición final calculando el desplazamiento total tal y como se muestra en e código 6.2.

```
if ((init_pos[1] - end_pos[1])<0){
    direccionalidad=init_pos[1] / (test_movement->init_pos[1]
        + (end_pos[1] - init_pos[1]));
}
else{
    direccionalidad=init_pos[1] / (init_pos[1] + (init_pos[1]
        - end_pos[1]));
}
```

Listing 6.2: Cálculo de la linealidad del movimiento

Por todo lo anteriormente explicado hemos decidido utilizar la función salud presentada a continuación: $salud = \sqrt{\frac{distancia\_recorrida^3}{tiempo}} \cdot \sqrt{\frac{1}{estabilidad}} \cdot direccionalidad$

Finalmente nos hemos decantado por esta función salud tras muchas funciones probadas, este formato de combinación (multiplicación de factores) y exponentes elegidos se ha mostrado lo suficientemente discriminante en los experimentos. Premia la distancia recorrida frente a cualquier otro parámetro, conseguir una forma de andar en la que el robot de pasos largos y que sea capaz de moverse de forma rápida y estable es lo más complicado. Damos tanto peso a la distancia recorrida porque las diferencias de este valor entre unas caminatas y otras es mínimo (recordemos que el robot mide 58 centímetros). Otras de las funciones probadas ha sido:
$salud = (\frac{distancia\_recorrida}{tiempo} - \sqrt{estabilidad}) \cdot direccionalidad$

6.3. Estudio de movimientos

Una vez sentadas las bases de nuestra evaluación y comprobado que la búsqueda sistemática no es factible tenemos que buscar otras formas para evaluar los movimientos. Basándonos en condiciones de cæteris paribus, en el que se mantienen constantes todas las variables de una solución a excepción de aquella o aquellas que vayan a ser objeto de estudio. De esta manera podemos realizar búsquedas bidimensionales suponiendo que el resto de los parámetros son constantes, fijándolos

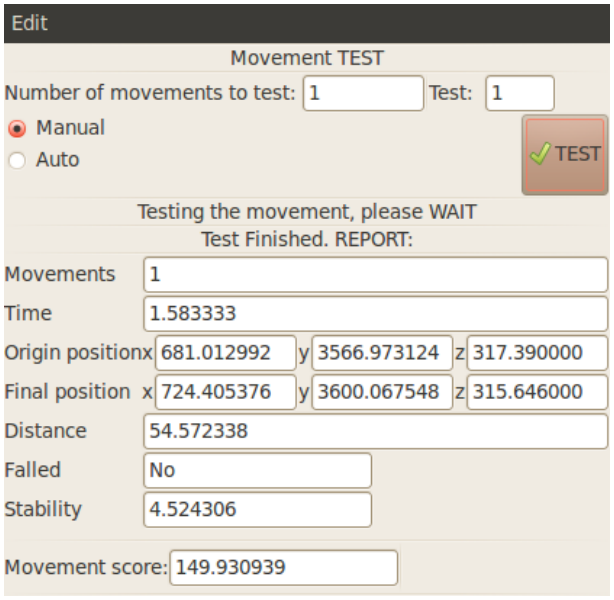


Figura 6.4: Interfaz gráfico del NaoOperator con soporte para la evaluación de movimientos.

con valores de movimientos válidos lo que nos permitirá conocer más la naturaleza del espacio de posibles caminatas y el efecto de estos parámetros variables sobre la calidad final del movimiento.

Para realizar esta búsqueda bidimensional hemos utilizando una funcionalidad que nos ofrece el NaoOperator (figura 6.5) dentro de la herramienta para evaluar movimientos en la que podemos elegir los dos parámetros que queremos explorar así como los rangos de búsqueda y el intervalo de cada uno. La búsqueda se realiza de modo totalmente automático, de forma que el simulador coloca al robot en una posición inicial (la misma para todos los movimientos), realiza el movimiento, lo evalúa y guarda los resultados en los movimientos en dos ficheros. El primer fichero guarda los parámetros usados para generar el movimiento (ejemplo en el código 6.3), que podemos cargar a posteriori para volver a recrear el movimiento; y un segundo fichero con los resultados obtenidos tras la evaluación, un ejemplo de este tipo de fichero es el mostrado en el código 6.4. Asimismo otra de las opciones para la presentación de resultados que nos aporta el NaoOperator es la posibilidad de mostrar de los resultados en una ventana como la mostrada en la figura 6.4.

A continuación explicaremos dos de los estudios realizados, los impactos que producen los cambios de los valores de amplitud y desplazamiento de la cadera y la rodilla en la calidad de la caminata. Podemos ver los grados de dos grados de libertad que controlar estos dos actuadores en la figura 6.6, que son la flexión de la rodilla y la elevación de la pierna con respecto al tronco.

```
right_ankle_pitch_a 5.000000
right_ankle_pitch_s -24.799999
right_ankle_pitch_p 0.000000
right_hip_pitch_a 7.000000
right_hip_pitch_s -29.900000
right_hip_pitch_p 0.000000
right_knee_pitch_a 8.000000
right_knee_pitch_s 38.000000
right_knee_pitch_p 0.000000
arms_amplitude 50.000000
balance 29.900000
n_pos 18
w 1.500000
```

Listing 6.3: Fichero con los parámetros usado para la generación de un movimiento.

```
Name test-movement/test-1-results
n_moves: 3
time: 4.750000
distance: 197.780182
origin: 3567.300000 318.022000 879.837968
end: 3568.074180 314.021000 0.000000
lateral_displacement: 1904.917960
balance: 4.738944
salud: 586.191406
```

Listing 6.4: Fichero con los resultados de la evaluación de un movimiento.

Figura 6.5: Interfaz gráfico del NaoOperator con soporte para la búsqueda bidimensional.

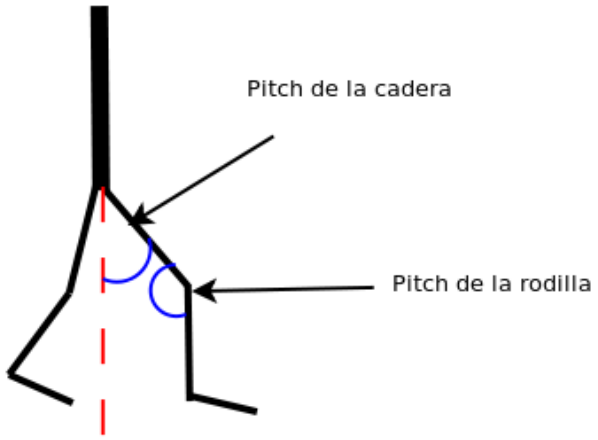


Figura 6.6: Esqueleto con los grados pitch de cadera y pitch de rodilla señalados.

Impacto de los cambios de amplitud y desplazamiento de la cadera en la calidad de la caminata

El pitch de la cadera supone uno de los actuadores fundamentales a la hora de la locomoción del robot, es el actuador encargado de fijar la verticalidad del robot y de elevar cada una de las piernas al comenzar una zancada. Como podemos ver en la gráfica de la figura 6.7(a), la calidad del movimiento disminuye si el desplazamiento es superior a 27 o inferior a 24. Ya que este actuador fija la verticalidad del Nao es normal que el rango para este actuador en el que la caminata es óptima sea muy reducido, si nos salimos de ese rango el robot andará inclinado hacia atras o hacia delante, tal y como queda reflejado en la figura 6.7(b). Al igual que sucede con el caso anterior esta verticalidad puede ser compensada por los desplazamientos de los otros dos actuadores que intervienen en la marcha, por ejemplo, podemos compensar una inclinación frontal con una mayor flexión de la rodilla.

Con la amplitud sucede lo mismo, si realizamos una caminata con valores muy grandes el robot, o bien se caerá, o bien no conseguirá un buen desplazamiento. Si la amplitud es pequeña el robot no es capaz de levantar las piernas por lo que no se caerá pero tampoco conseguirá avanzar, con una amplitud grande el robot sí que conseguirá levantar la pierna pero la levantará en exceso por lo que realizará una caminata con grandes desequilibrios o se caerá.

Impacto de los cambios de amplitud y desplazamiento de la rodilla en la calidad de la caminata

Tal y como podemos apreciar en la gráfica figura 6.8(a), no tiene sentido la evaluación de caminatas de este tipo con un valor mayor a, aproximadamente 41. Esto sucede porque a partir de este valor el Nao flexiona mucho las rodillas (figura 6.8(b)). Como los motores que pueden compensar este desfase aumentado son el pitch de la cadera y el pitch del tobillo no varían ya que son tomados como valores fijos. El

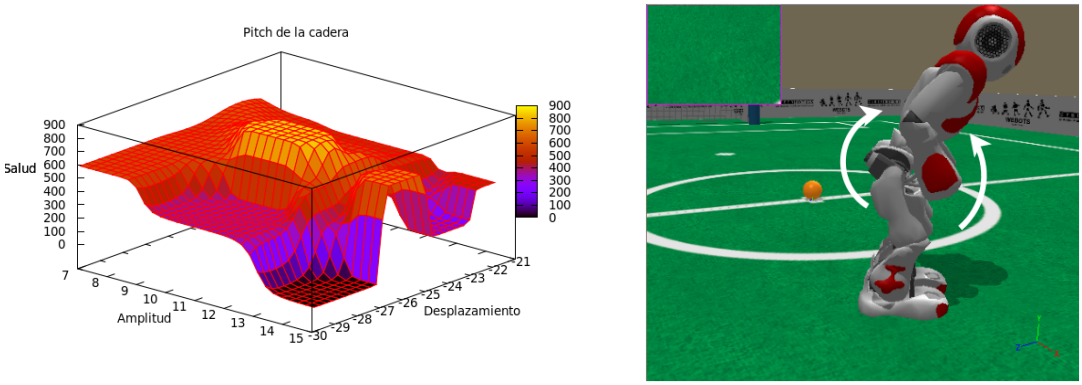


Figura 6.7: Gráfica con variaciones en la cadera (amplitud/desplazamiento) (a). Nao con valores altos en el pitch de la cadera(b).

robot al comenzar el movimiento se caerá en la mayoría de los casos. Probablemente en una forma de andar donde estos valores estén ajustados si tendría sentido buscar con un desplazamiento mayor, con lo que conseguiríamos una caminata en la que el robot anduviera agachado.

En lo referente a la amplitud de este actuador podemos ver que tanto para valor bajos como altos la calidad de la locomoción disminuye. Este hecho es razonable ya que la amplitud de oscilación del movimiento debe estar ajustada al tipo de locomoción, si existe muy poca amplitud el robot no será capaz de realizar el movimiento necesario para desplazar el peso del robot mientras que si este movimiento es excesivo perderá estabilidad por la elevada flexión de la rodilla y acabará en el suelo.

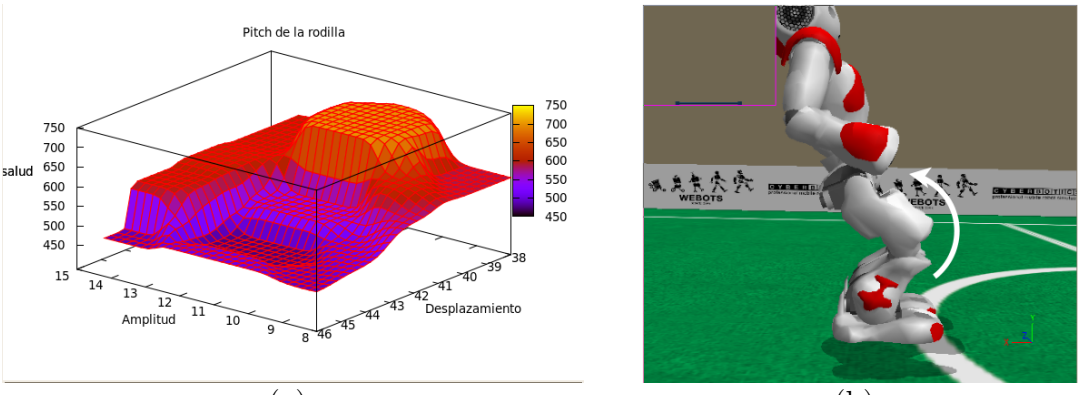


Figura 6.8: Gráfica con variaciones en la cadera (amplitud/desplazamiento) (a). Nao con valores altos en el pitch de la rodilla(b).

6.4. Búsqueda sistemática

La búsqueda en un espacio bidimensional fijando el resto de parámetros a valores fijos es útil para ver cómo se comporta el movimiento al variar unos parámetros en concreto, pero no es suficiente para encontrar la mejor forma de andar, ya que hay parámetros que influyen directamente en otros. Por ejemplo, si fijamos el desplazamiento de la rodilla a un valor muy alto hará que la rodilla esté muy flexionada por lo que la cadera y el tobillo tendrán que compensar este movimiento para mantener un buen equilibrio y las amplitudes también se deben ajustar.

Para reducir el espacio de búsquedas hemos diseñado un algoritmo que efectúa un recorrido sistemático *acotado* en el espacio de búsqueda con una serie de restricciones previamente fijas como los máximos, mínimos e incrementos de cada uno de los parámetros. De esta forma nos permite probar caminatas con pocos valores para cada parámetro. Para ello el NaoOperator nos ofrece un componente con este fin y cuyo interfaz gráfico, con el que tendremos que interactuar, lo vemos en la figura 6.9. En este ejemplo hemos dejado fijos algunos de los parámetros y hacemos una exploración sobre 3 de ellos, con un total de 189 combinaciones, lo que suponen unos 22 minutos de búsqueda, al acortar cada uno de los actuadores que van a ser incluidos en la búsqueda de manera dinámica y gráfica vemos en el interfaz la cantidad de combinaciones posibles con la selección que hemos realizado, pudiendo modificar las acotaciones tantas veces sean necesarias, antes de empezar la búsqueda, para ajustarse a nuestras necesidades.

Como hemos explicado en el apartado 6.1, el espacio de búsquedas es inmenso debemos restringir las búsquedas para que tarden lo menos posible. Como realizar las evaluaciones con el robot real es un trabajo costoso y peligroso para el robot, recordemos que muchas de las posibles caminatas basadas en nuestro modelo harán que el robot se caiga, hemos decido utilizar el simulador Webots para este fin. Webots nos aporta un entorno virtual con un modelo de robot Nao simulado bastante fiel, con el cual podemos automatizar las evaluaciones de movimientos. El procedimiento es bastante sencillo: una vez fijados los criterios de nuestra búsqueda sistemática acotada se cargará el primer movimiento en el NaoOperator, colocará el robot en una posición inicial y hará que el robot repita la secuencia (figura 6.10) n veces (previamente fijadas en el evaluador de movimientos). De esta manera todas las caminatas empezarán con las mismas condiciones iniciales. este procedimiento se realizará tantas veces como combinaciones tenga nuestra espacio acotado. Repetimos cada movimiento n veces para tener la certeza de que el movimiento se ha evaluado de forma correcta y no se ha producido ningún error. A lo largo de todo este proyecto, en las pruebas realizadas, el número de repeticiones ha sido el mismo, 3.

Uno de los problemas que nos hemos encontrado durante la realización de largas evaluaciones automáticas de caminatas es Webots es un error relacionado con el alto consumo de cpu, tras varios minutos de ejecución Webots empieza a consumir muchos recursos 6.11 y las evaluaciones empiezan a no ser válidas, el controlador del robot no recibe las órdenes a tiempo y en la mayoría de los casos acaba por

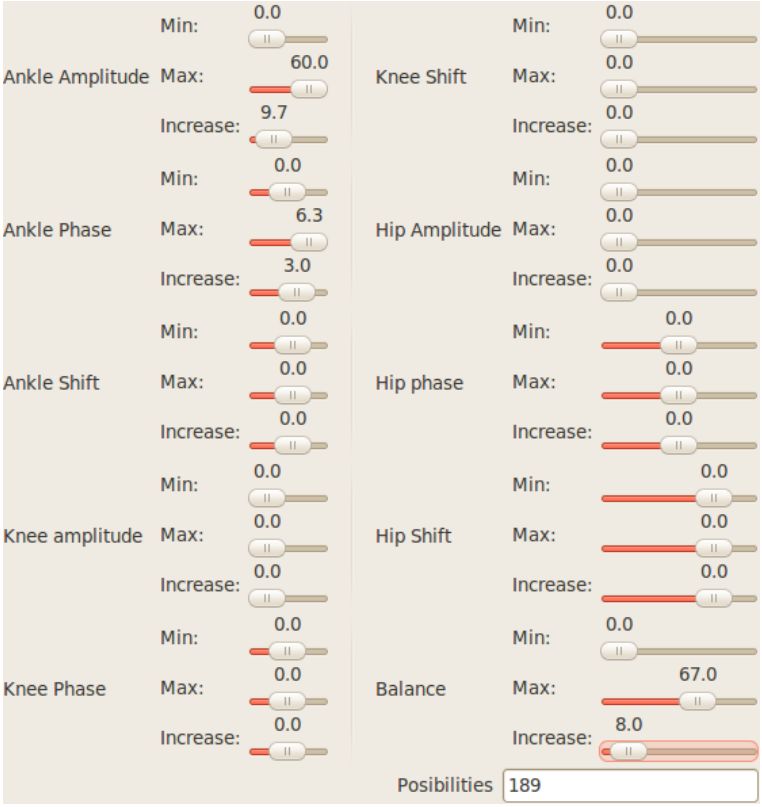


Figura 6.9: Interfaz gráfico del NaoOperator con soporte para la búsqueda multidimensional.

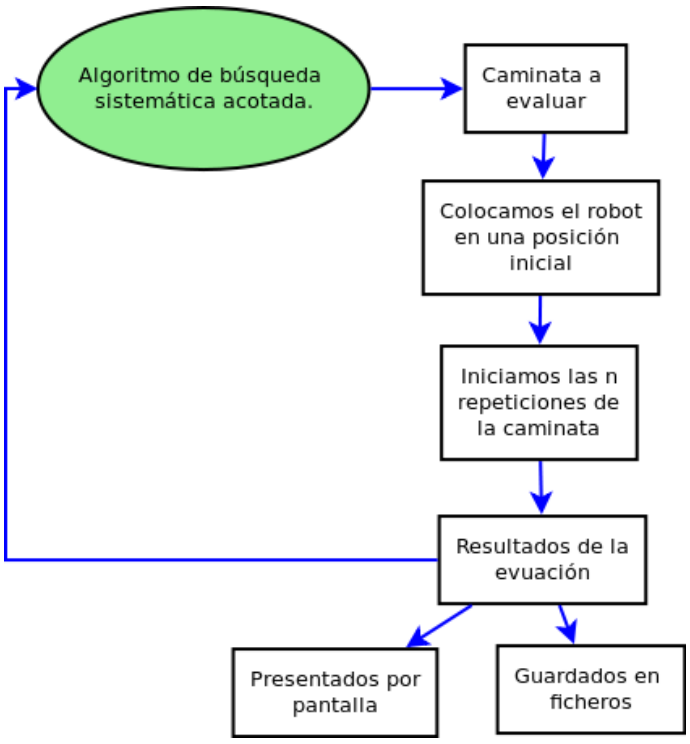


Figura 6.10: Secuencia de evaluación de caminatas en el simulador.

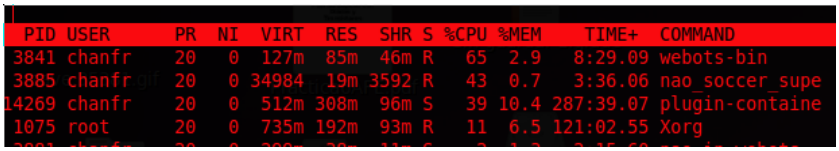


Figura 6.11: Pantallazo del comando top mostrando el consumo de cpu por parte de webots.

caerse, esto supone que sólo se puedan evaluar unos 50 o 60 movimiento de forma secuencial. Este problema se soluciona tras el reinicio del simulador pero para que las evaluaciones sean completamente automáticas debemos detectar el problema cuando se produzca sin necesidad de controlarlo desde el exterior. Para ello hemos utilizado dos de las funcionalidades ofrecidas por las herramientas desarrolladas en este proyecto. El esquema se encarga de, cada 10 evaluaciones, comprobar el estado de la cpu (utilizando el *script* del código 6.5) y en caso de que éste sea elevado activa un *flag* de reinicio. El driver al detectar esta nueva situación, el driver desactiva todos los componentes conectados al simulador y le manda mediante uno de los *sockets* conectados a Webots una señal para que se reinicie y el driver se queda en espera. Una vez que Webots se ha reiniciado de forma correcta éste informa de este estado al driver para que pueda continuar la evaluación de movimientos. De esta manera la secuencia de evaluación de caminatas de forma automática queda como muestra la figura 6.12

```
#!/bin/bash
#top -bn2 | grep Cpu | cut -d"%" -f1 | cut -d" " -f2- | cut -d" " -f2-
    | tail -n 1
cat /proc/loadavg | cut -d" " -f1
```

Listing 6.5: Script para la comprobación del estado de la cpu.

Otro de los problemas encontrados a raíz de la solución anterior es que tras numerosos reinicios del simulador para paliar el problema del alto consumo de cpu es simulador termina por cerrarse. Este problema se produce justo cuando Webots se está reiniciando por lo que no afecta a nuestra aplicación de JDErobot ya que tiene todos los componentes desactivados, el problema es que seguirá en este estado hasta que Webots le informe de que se ha reiniciado correctamente y como Webots está cerrado esa situación nunca se producirá. Por ello hemos tenido que desarrollar otro script (código 6.6) que se encargue de arrancar el simulador en caso de que se haya cerrado de forma inesperada. Este scripts se ejecuta de forma independiente a nuestras herramientas de JDErobot y su funcionamiento es simple, comprueba cada 10 segundos si el simulador se esta ejecutando, si no es así éste se encargará de volver a arrancar con el mundo que estamos utilizando para nuestras evaluaciones.

Un concepto importante a la hora de realizar búsquedas en espacios tan grandes es el de la división o paralelización de búsquedas. Tenemos a nuestra disposición los equipos del laboratorio de robótica de la Universidad Rey Juan Carlos (figura **??** para realizar pruebas, compuesto por un total de 10 equipos, muchos de ellos con equipamientos muy potentes. El gran inconveniente es que las pruebas a realizar serán independientes en

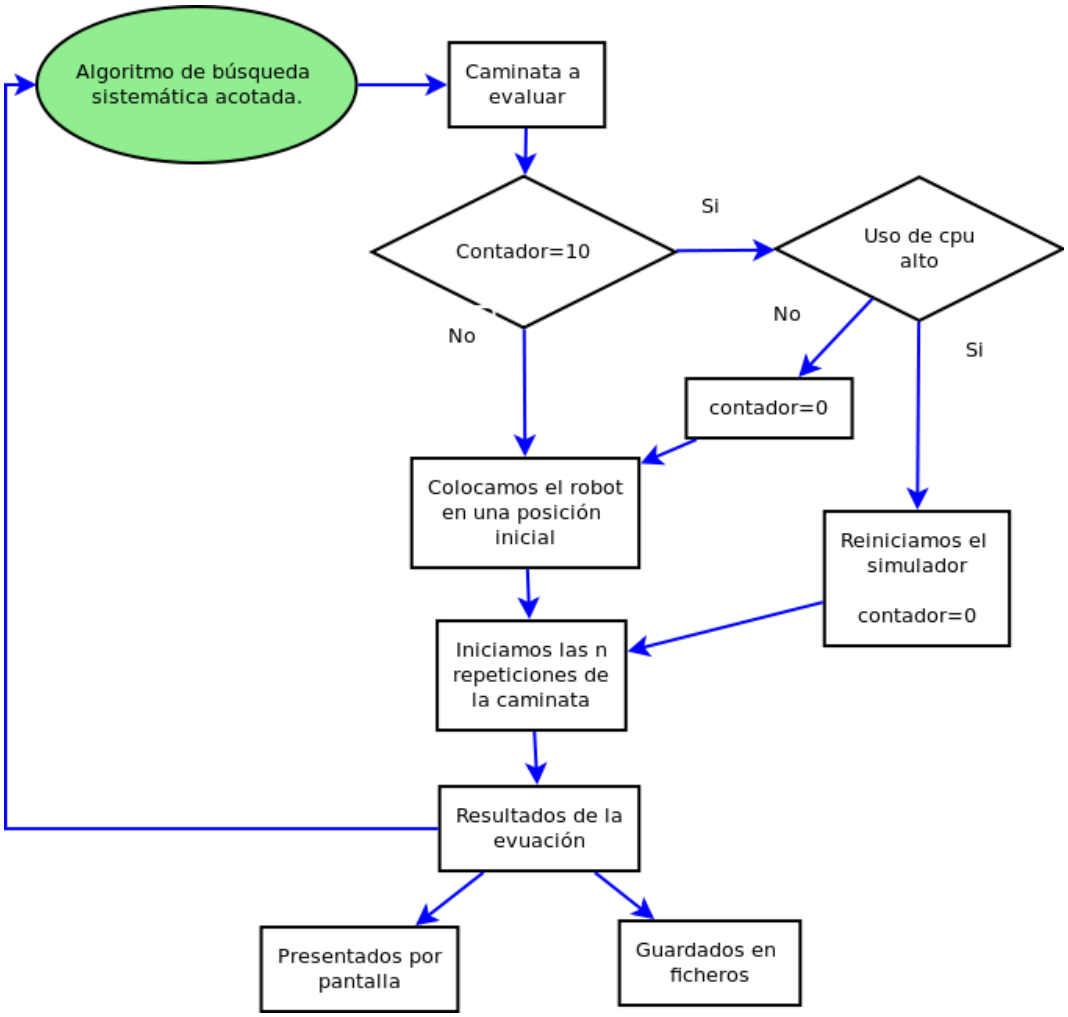


Figura 6.12: Secuencia de evaluación automática con comprobación del estado de la cpu

```
#!/bin/bash

while true
do
    numero=‘ps -eaf | grep webots | wc -l‘
    if [ $numero -eq 1 ]
    then
        $WEBOTS_HOME/webots/webots projects/contests/nao_robocup/worlds/nao1.wbt &
    fi
    sleep 10
done
```

Listing 6.6: Script para la comprobación del estado de Webots.

cada uno de los equipos lo que supone una tarea complicada a la hora de interpretar los resultados. La mejor solución para realizar una búsqueda conjunta sería, una vez fijada la discretización a utilizar en nuestra búsqueda, dividir el espacio en partes iguales para que cada equipo evalúe una parte equitativa y una vez termina la búsqueda juntar todos los resultados e interpretarlos de forma conjunta. Para cubrir esta necesidad hemos incorporado al evaluador de movimientos esta funcionalidad que nos da como resultado una serie de ficheros con los parámetros correspondientes a su búsqueda, este fichero puede ser cargado directamente al NaoOperator y se podrá comenzar a realizar la búsqueda con las especificaciones fijadas en dicho fichero.

FOTO DEL LABORATORIO!!

Tras el estudio de los movimientos y la búsqueda exhaustiva en nuestro espacio de búsquedas la mejor forma de andar encontrada corresponde a los valores presentados en la tabla 6.4. Este movimiento nos permite gran versatilidad ya que se comporta correctamente a diferentes frecuencias, con lo que podemos conseguir caminatas con diferentes velocidades sin tener que modificar ningún otro parámetro, es estable y por definición del tipo de movimiento enlazable. Este movimiento tiene una velocidad de *0.18 m/s*, su puntuación en estabilidad es bastante bajo, 26 (recortemos que el valor de la estabilidad aumenta si se producen desequilibrios) y la linealidad es prácticamente perfecto 0.997, lo que significa que en todo el recorrido efectuado durante la evaluación solo se ha desviado 3 milímetros.

Parámetro	**Valor**
Frecuencia	0.9
Amplitud del pitch de la cadera	11
Desplazamiento del pitch de la cadera	-27
Amplitud del pitch de la rodilla	11
Fase del pitch de la rodilla	0
Desplazamiento del pitch de la rodilla	42.1
Amplitud del pitch del robillo	7.5
Fase del pitch del tobillo	0
Desplazamiento del pitch del tobillo	-21.4

Cuadro 6.1: Tabla los valores correspondientes al mejor movimiento encontrado.

Una vez encontrada una buena caminata (rápida, estable y lineal) y tras realizar varias pruebas en el simulador utilizamos el robot real para la evaluación de la caminata sobre un soporte real. Realizar el cambio de simulador a robot real desde el punto de vista software es muy fácil, ya que como hemos explicado en el capitulo 4.3 sólo tenemos que cambiar la IP del robot y especificar en el fichero de configuración de JDErobot que el robot que vamos a utilizar es un robot real. Frecuentemente el funcionamiento de la caminata no se comporta exactamente como en el simulador, por ello en numerosos casos es necesario afinar esta caminata de forma manual. Esta diferencia de comportamiento con la misma caminata se produce porque aunque la simulación de Webots es muy buena, no es perfecta. En las figuras 6.4 y **??** tenemos unas capturas del movimiento del robot para poder comprar las posibles diferencia entre ambas caminatas.

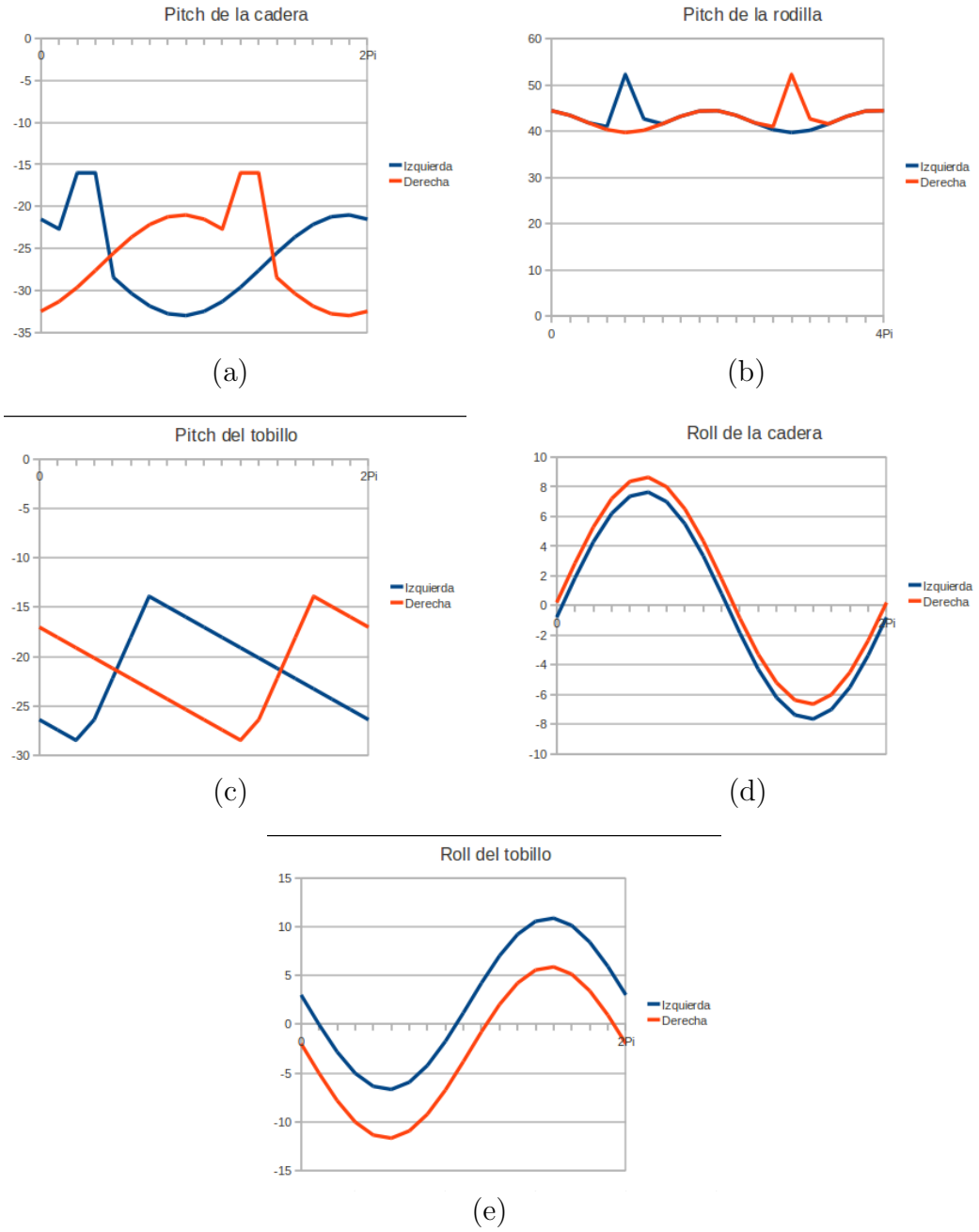


Figura 6.13: Gráficas de los actuadores del mejor movimiento encontrado

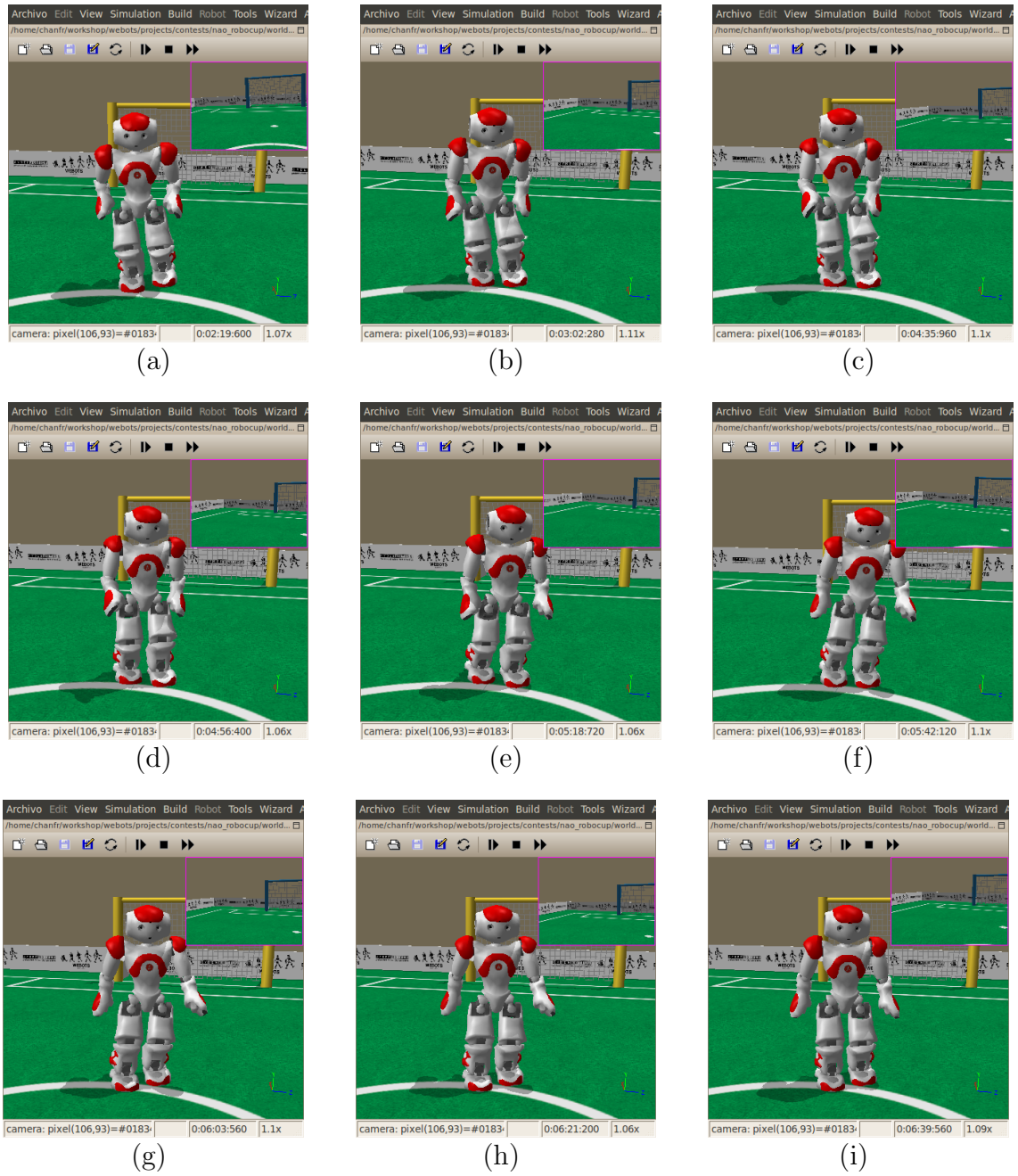


Figura 6.14: Secuencia de la mejor caminata en el robot simulado.

Capítulo 7
Conclusiones y trabajos futuros

En los capítulos anteriores hemos hecho una amplia descripción del problema que hemos abordado y las soluciones que hemos desarrollado así como alguno de los resultados obtenidos. En este capítulo se presentarán las conclusiones principales del proyecto y se propondrán una serie de líneas futuras de investigación que pueden continuar a partir de este proyecto fin de carrera.

7.1. Conclusiones

Tras un desarrollo de cerca de 26.000 líneas de código divididos en la aplicación NaoOperator y en el driver NaoBody hemos conseguido alcanzar los objetivos planteados en el capítulo 2.1, consiguiendo una forma de andar propia para el humanoide Nao que es veloz y estable.

A continuación repasaremos los distintos subobjetivos planteados para conocer la solución a cada uno de ellos:

1. El primer subobjetivo a alcanzar era la creación de un driver que permita de acceder a los componentes del humanoide Nao. Tal y como hemos explicado en al apartado 4.2, hemos desarrollado el driver NaoBody nos ofrece un soporte completo de todos los componentes del robot, así como varias funcionalidad extra en caso de tratarse de un robot simulado en Webots.

2. El siguiente subobjetivo era crear un esquema para JDErobot capaz de controlar el robot, por ello hemos desarrollado la aplicación NaoOperator, explicada en el apartado 4.3, que es capaz de interactuar con el driver del robot para controlar cada uno de sus componentes. Este esquema nos ofrece un interfaz gráfico muy intuitivo con el que poder realizar cualquier acción sobre el robot: mover actuadores, encender leds, mover al robot, sintetizar texto...
   Con el fin de crear otra aplicación adicional para el control del robot Nao hemos desarrollado también una aplicación para el sistema iOS4 con el que podemos controlar el robot Nao desde cualquier dispositivo que utilice esta arquitectura como, por el ejmplo, el iPhone de Apple.

3. Con todo lo anteriormente desarrollado conseguir una forma de andar totalmente parametrizada. Tras el estudio de la forma de andar ofrecida por el fabricante hemos modelizado este movimiento basándonos en ondas sinusoidales. Finalmente

el modelo conseguido queda parametrizado con 10 valores:

a) Frecuencia (γ), común para todos los actuadores.
b) Amplitud del pitch de la cadera.
c) Desplazamiento vertical (γ) del pitch de la cadera.
d) Amplitud del pitch de la rodilla.
e) Fase (β) del pitch de la rodilla.
f) Desplazamiento vertical (γ) del pitch de la rodilla.
g) Amplitud del pitch del tobillo.
h) Fase (β) del pitch del tobillo.
i) Desplazamiento vertical (γ) del pitch del tobillo.
j) Amplitud de balanceo.

4. El último subobjetivo era lograr un procedimiento de búsqueda capaz de ayudarnos a encontrar el mejor movimiento modelado con nuestra forma de andar parametrizada. Debido a la complejidad encontrada a la hora de realizar búsquedas en el espacio acotado por los parámetros de nuestra marcha, explicado en el capítulo 6, hemos tenido que aplicar diferentes estrategias de búsqueda para lograr una forma de andar que se ajuste a nuestros requisitos.

Después de realizar numerosas búsquedas hemos conseguido encontrar una forma de andar que se ajusta a nuestras especificaciones, es estable, rápida, enlazable y lineal.

Requisitos

Los requisitos que han tenido que cumplir la aplicaciones desarrolladas en este proyecto han estado marcadas por lo especificado en el capítulo 2.2.

Como ya habíamos establecido en los requisitos, la arquitectura sobre la que se base el desarrollo de esta aplicación es JDErobot, siendo finalmente los lenguajes C, C++ y Objective C los utilizados. El hecho de utilizar esta plataforma nos ha facilitado, en gran medida, el desarrollo de nuestras aplicaciones y driver ya que JDErobot es el encargado de la interconexión de todos los componentes.

Las aplicaciones desarrolladas para JDErobot han sido el driver NaoBody, el esquema NaoOperator, al que además de la funcionalidad de teleoperador completo del robot Nao le hemos incorporado todas las funcionalidad necesarias para el trabajo con nuestro modelo de caminata parametrizada, así como diversas herramientas para la evaluación y búsqueda.

Otro de los requisitos que cumple nuestro proyecto en la búsqueda de movimientos es que la evaluación, basada en valores como velocidad, estabilidad y linealidad.

Novedades en este proyecto

Las novedades presentadas en este proyecto con básicamente dos: la creación de un soporte completo a JDErobot para el robot Nao, desde un driver completo que nos ofrece acceso a todos los componentes del robot como un esquema mediante el cual, en forma de interfaz gráfico, podemos controlar el Nao los actuadores del robot, crear movimientos, coreografías, etc..

La segunda novedad que aportamos es el estudio realizado sobre la locomoción con robots bípedos y la modelización de este movimiento mediante una serie de parámetros. Con ello conseguimos una forma totalmente nueva para este robot humanoide que podrá usarse en futuros proyectos o competiciones robóticas.

Conocimientos adquiridos

Los conocimientos adquiridos durante el desarrollo de este proyecto han sido muy numerosos. Desde el punto de vista de programación, el uso de C++ para la comunicación con el robot mediante el middleware de NaoQi, las librerías de apra e inet para la creación de sockets en los componentes de red, OpenGL para la creación de los visores en 3D y los objetos que hay en ellos, Glade y GTK+ para el desarrollo de interfaces gráficas, Objective-C para el desarrollo de la aplicación del iPhone, etc.

En lo que se refiere a la locomoción de un robot bípedo los conocimientos han sido muy diverso, desde estudio de la marcha y patrones en el movimiento hasta los conocimientos matemáticos necesario para el modelado de las trayectorias de los motores hasta diversas estrategias necesarias para aumentar la estabilidad y mejorar el movimiento.

También hemos adquirido numerosos conocimientos con respecto al uso del simulador Webots y la programación de controladores para este simulador, ya que hemos tenido que hacer algunos cambios en el controlador del simulador para simplificar algunas cosas en este proyecto.

7.2. Trabajos futuros

Este proyecto fin de carrera asienta las bases para futuros trabajos con el robot Nao desde la plataforma JDErobot. Utilizando las aplicaciones desarrolladas en este proyecto, que aporta un soporte completo de los componentes, se podrán realizar cualquier tipo de tareas con este robot. Algunas de las líneas con más interesantes relacionadas con la locomoción desarrollada en este proyecto serán detalladas a continuación.

Podemos usar la locomoción utilizada en este proyecto y aplicarla a numerosas tareas. Una línea futura podría ser aplicar las ideas presentadas por Julio Vega en su proyecto de robot guía [1] y usarlas con este humanoide.

[1] http://jderobot.org/index.php/Jmvega_guide_robot

Otra línea de investigación sería enfocar la funcionalidad desarrollada en este proyecto para locomoción lineal para encontrar formas óptimas de movimientos arqueados, laterales y giros con lo que conseguiríamos una locomoción completa con un modelo parametrizado de todos los movimientos posibles del robot.

Una tercera línea de investigación directamente relacionada con este proyecto sería ampliar las búsquedas de nuestra caminata modelada utilizando búsquedas genéticas con algoritmos evolutivos basados en nuestra función salud o a partir de una nueva.

Otro proyecto interesante sería modificar las funciones base de la modelización de este proyecto para comprobar si se consiguen mejoras importantes. Podemos basar el movimiento en otra forma de andar de NaoQi con parámetros modificados o directamente a partir de una forma creada totalmente desde 0 siguiendo patrones de marcha de humanos.

Bibliografía

[Godoy Míguez, 2007] Manuel Francisco Godoy Míguez. Diseño de sistema distribuído sobre robot móvil autónomo para caracterización de terrenos. 2007.

[Gómez, 2008] Juan González Gómez. Robótica modular y locomoción: aplicación a robots ápodos. *Tesis doctoral, Universidad Autónoma de Madrid*, 2008.

[Ltd., 2010a] Cyberbotics Ltd. Webots reference manual (release 6.2.4). 2010.

[Ltd., 2010b] Cyberbotics Ltd. Webots user guide (release 6.2.4). 2010.

[macprogramadores.org, a] macprogramadores.org. El lenguaje objective-c para programadores c++ y java.

[macprogramadores.org, b] macprogramadores.org. Programación cocoa con foundation framework.

[Muelas Sahagún, 2008] David Muelas Sahagún. Visualizador 3d interactivo para laboratorios de análisis de marcha. *Proyecto Fin de Carrera. ITIS - Universidad Rey Juan Carlos*, 2008.

[Perdices García, 2009] Eduardo Perdices García. Autolocalización visual en la robocup basada en detección de porterías 3d. *Proyecto fin de carrera - Ingeniería en informática superior - Universidad Rey Juan Carlos*, 2009.

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

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

[Robotics, 2010] Aldebarán Robotics. Documentación y guía de referencia del robot nao (versión 1.3.17. 2010.

[www.robocup.es, 2009] www.robocup.es. Robocup soccer humanoid league rules and setup. 2009.