Ingeniería Informática Superior

Curso académico 2014/2015

Trabajo Fin de Carrera

Caminatas basadas en ondas acopladas para el humanoide Nao en Gazebo-5 y JdeRobot.

Autor: Francisco Pérez Salgado

Tutor: José María Cañas Plaza

Una copia de este proyecto, las fuentes del programa y vídeos de los experimentos están disponibles en la siguiente dirección:

http://jderobot.org/Fperez

Agradecimientos

Más de un año ha pasado desde que comencé esta aventura robótica que pone punto y final mi vida como estudiante. He de decir que no ha sido nada fácil y que ha habido momentos tanto muy buenos, como muy malos. No obstante, no hay recompensa sin sacrificio, así que sólo puedo decir que mereció la pena.

Quiero dar mi agradecimiento más sincero a Jose María Cañas, tutor de este proyecto y sobretodo, gran mentor. Nada de esto habría llegado a buen puerto sin los conocimientos, la sabiduría y lo más importante, el apoyo de este gran maestro. Gracias a él he aprendido más que en cualquier otro sitio en el que hubiera estado.

Por supuesto a toda mi familia, que se ha volcado conmigo dándome ánimos cuando los he necesitado, y apoyándome siempre tanto en los buenos momentos como en los malos.

Cómo no a mis dos grandes amigos Emilio y Marcos, que me han dado fuerza y energía para concluir este proyecto, con sus irreverencias y sus frases graciosas, que nunca pasan de moda para nosotros.

Por último, dar las gracias a todas aquellas personas que, de un modo u otro han sido participes de esta andadura, porque todas ellas influyen en lo que eres y en lo que haces, por lo tanto, gracias.

Muchos baches, muchas idas y venidas, pero lo que predomina, por encima de todo... la satisfacción.

Resumen

Una parte relevante del desarrollo creciente de la robótica reside en los simuladores, los cuales, son cada vez más potentes, fiables y ofrecen muchas posibilidades a la hora de estudiar, desarrollar y probar robots en multitud de situaciones, entornos y condiciones diferentes. Además de ofrecer gran flexibilidad en términos de testeo, los simuladores también permiten añadir ese factor de seguridad tan necesario a la hora de probar nuevos comportamientos que pueden ser, en ocasiones, inesperados.

Uno de los aspectos más complicados a día de hoy a la hora de programar robots humanoides es dotarles de estabilidad y de un movimiento natural. Dicha dificultad reside en la gran cantidad de factores de los que dependen estos comportamientos, por lo que algo tan simple para las personas como caminar puede resultar una tarea muy laboriosa cuando se trata de robots bípedos.

Este proyecto aborda estos dos frentes aplicándolos sobre el robot humanoide de *Aldebaran Robotics*: Nao, en el simulador Gazebo. Por un lado migraremos el modelo existente en JdeRobot a la última versión del simulador, retocando tanto el modelo (con sus parámetros tanto físicos como visuales) como los *plugins* que dotan de movilidad al robot. Por otro lado, hemos modelizado una forma de caminar para el Nao basándonos en el acoplamiento de ondas diferentes para cada articulación. Esta forma de caminar parametriza la caminata y se van generando valores para dichos parámetros mediante una búsqueda rastrillada en un espacio de búsqueda.

Para el desarrollo de este proyecto se ha utilizado como plataforma JdeRobot 5.3. Además como lenguaje de programación se ha utilizado C++, ICE para las interfaces de comunicación y QT como librería gráfica para la interfaz de usuario.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

En este capítulo introductorio se describirá a vista de pájaro el contexto en el que se enmarca este proyecto. Para ello, comenzaremos con una breve descripción de la robótica, a la que le seguirá una explicación general del tipo de robots que ocupa este proyecto: los humanoides, para finalizar con los aspectos más importantes del uso de simuladores en robótica.

1.1. Robótica

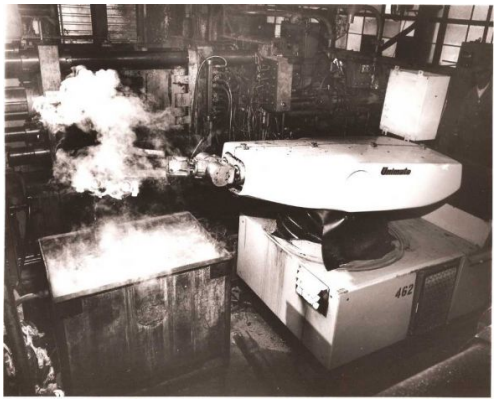
La robótica es la disciplina que engloba tanto el diseño como la construcción y programación de robots. Esta disciplina no es aislada, ya que con ella se combinan muchas otras como la mecánica, electrónica, informática, inteligencia artificial o ingeniería de control. Citando a *Robot Institute of America*, se puede decir que:

"*un robot es un dispositivo multifuncional reprogramable diseñado para manipular y/o transportar material a través de movimientos programados para la realización de tareas variadas.*"

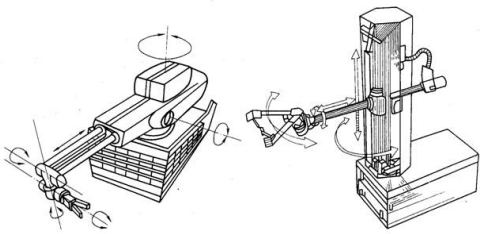
En esencia, un robot se compone de tres ingredientes fundamentales: sensores, actuadores y uno o más computadores. Podemos considerar un robot a toda aquella máquina o sistema informático que disponga de sensores: encargados de obtener la información del entorno que les rodea; actuadores: encargados de interactuar con dicho entorno; y uno o más computadores: encargados de ejecutar el software que analiza la información capturada por los sensores y genera señales correspondientes en los actuadores. Un último ingrediente que no se ha citado es el software. A pesar de no ser un elemento hardware, es la parte más importante de un robot, ya que es el encargado de dotar de

un determinado comportamiento al mismo a través del procesamiento de información procedente de los sensores y la comanda de órdenes a través de los actuadores, utilizando como soporte un computador.

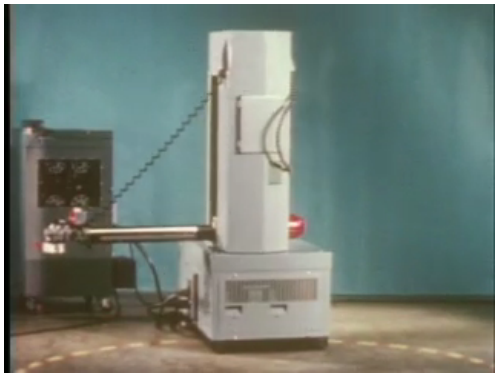
El primer robot programable y dirigido de forma digital data del año 1961. Diseñado por *Unimate*[1] éste primer robot realizaba tareas potencialmente peligrosas para humanos como levantar piezas calientes de metal de una máquina de tinte y colocarlas después correctamente.



(a)



(b)



(c)

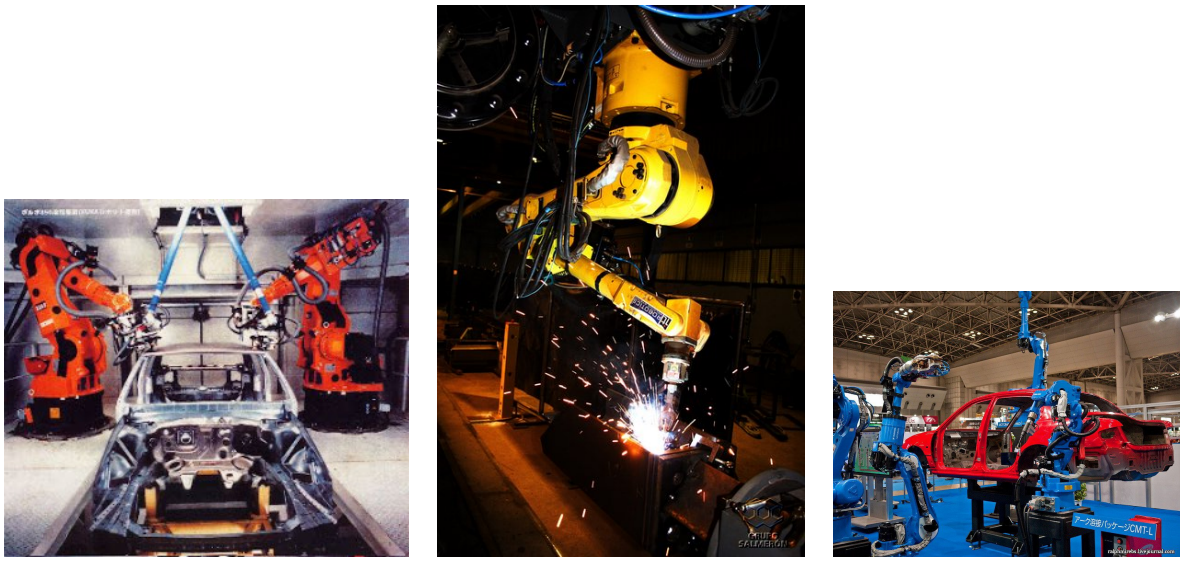
Figura 1.1: (a) Robot *Unimate*, (b) esquemas del *Versatran* y del *Unimate* y (c) Robot *Versatran*.

Esa fue la piedra angular que abrió el desarrollo masivo de diversos tipos de robot para todo tipo de usos. En los años siguientes, la robótica fue perfilándose hacia entornos más industrializados, debido a la creciente evolución del sector industrial. Es por ello que comenzaron a surgir robots orientados a la automatización de tareas repetetitivas,

[1] `http://www.prsrobots.com/unimate.html`

peligrosas para los humanos e incluso para tareas complejas que requieren de gran precisión. Algunos de los ejemplos más significativos de la época en cuanto a robots industriales puede ser el *Versatran* (Figura 1.1(c)), el cual se desarrolló para el transporte de cargas, como su propio nombre indica *Verstaile transfer* dentro de la fábrica de Ford en 1962.

Ya en 1969 se lanzan los primeros robots más característicos del mundo de la robótica: los robots soldadores. Estos robots surgen de la necesidad de precisión, y velocidad a la hora de realizar las soldaduras de los automóviles en las cadenas de producción de las fábricas. Además, cubrían aspectos tan importantes como la seguridad de los trabajadores, al considerarse el trabajo de soldadura como peligroso y dañino para las personas. Este tipo de robots aumentan en gran medida la productividad de las cadenas de producción debido a su alta precisión y velocidad. Un operario humano no podría desarrollar la misma tarea que desarrolla un robot soldador de forma tan fiable, tan rápida y tan segura.



(a) (b) (c)

Figura 1.2: (a),(b) y (c) robots soldadores en plantas de producción industrial.

Poco a poco los robots han ido evolucionando y a día de hoy tenemos robots con aptitudes muy diferentes y capaces de desarrollar tareas más complejas, no sólo en el campo de la investigación, sino en escenarios domésticos o industriales. Estos están muy presentes en tareas que resultan repetitivas, aburridas o peligrosas para los seres humanos. Buen ejemplo de ello son los robots que trabajan en la construcción de coches, pero también hay ejemplos en otros sectores como el de la bollería industrial con robots que se utilizan para embalar magdalenas (Figura 1.3(a)). Además, siguiendo la línea del entorno industrial,

a parte de los ya mencionados robots han aparecido infinidad de robots para diferentes propósitos. Ya sea levantar cargas pesadas de forma autónoma, transporte de cargas y herramientas en un entorno controlado, o la simple tarea de embalar productos, como se acaba de mencionar. Uno de los ejemplos más claros a la hora de automatizar su funcionamiento es la empresa *Amazon*, la cual cuenta con una flota de vehículos no tripulados encargados del transporte de mercancías dentro de sus naves industriales [2]. Esta misma empresa está además contemplando la posibilidad de realizar entregas de paquetes en zonas de acceso complicado, como pequeños pueblos de montaña, mediante el uso de Drones, que se describen tres párrafos más abajo.

Actualmente, en el entorno doméstico, se comercializan robots muy útiles como el *Roomba* de la empresa iRobot (Figura 1.3(b)). Este robot es un robot aspirador para las casas, pudiendo aspirar sobre cualquier superficie (suelo liso, azulejos, parqué, alfombras, etc.). Es un robot que posee diferentes sensores, de contacto e infrarrojos, y con un algoritmo de navegación se pasea por el suelo recogiendo el polvo, suciedad y residuos que encuentre a su paso.

(a) (b) (c)

Figura 1.3: (a) Robot embalador de magdalenas, (b) *Roomba* de iRobot y (c) robot aéreo no tripulado.

El uso de *UAVs* (*Unmanned Aerial Vehicle* o vehículo aéreo no tripulado), o más comúnmente denominados *drones* (Figura 1.3(c)), cada vez está más extendido. En Alemania quieren usar este tipo de robots para defender los trenes de los grafiteros y el vandalismo, usándolos como medio de vigilancia en las cocheras. Con esta medida esperan ahorrar dinero, pues se estima en 10 millones de dólares anuales el gasto que supone limpiar los grafitis. Actualmente, en la Comunidad de Madrid se está estudiando el uso de robots de este estilo equipados con cámaras y sensores que detectan gases y agentes radiactivos para ayudar a los servicios de emergencia (bomberos, SAMUR y policía municipal) en labores de rescate, reduciendo no sólo costes sino también riesgos humanos.

[2] `http://www.youtube.com/watch?v=tMpsMt7ETi8`

1.2. Robots humanoides

En cuanto a la investigación dentro de la robótica, uno de los tipos más interesantes de robots son los humanoides (o bípedos). Este tipo de robot es el modelo que más interés despierta en la comunidad robótica debido a la infinidad de posibilidades que ofrecen. Al tratarse de robots con morfología humana, es mucho más complejo crear comportamientos asociados a las formas de actuar de las personas (el cual es el fin que se busca con el desarrollo de este tipo de robots) ya que, si los comparamos con los robots que hemos visto hasta ahora en este capítulo, sus funciones no son triviales. Prototipos como el *ASIMO* de Honda, el Fujitsu *HOAP-3* o el propio *Nao* de Aldebaran son la base de muchos esfuerzos en investigación, varios de ellos diseñados para replicar tanto la inteligencia como la maniobrabilidad humanas. Los mayores avances vistos hasta la fecha en robots humanoides se deben al desarrollo efectuado con *ASIMO*, el cual ya puede realizar autónomamente tareas humanas tales como, servir bebidas, subir y bajar escaleras (Figura 1.4(a)), e incluso correr.

(a) (b)

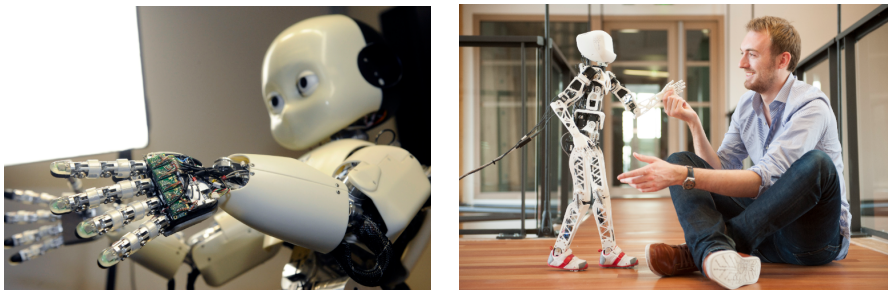
Figura 1.4: (a) *ASIMO* de Honda y (b) *Nao* de Aldebaran Robotics.

Algunos de los últimos lanzamientos en cuanto a robots humanoides son: Poppy (figura 1.5(b)), Romeo e iCub. Lo más destacable del primero, Poppy, es que está construido casi integramente a partir de piezas obtenidas de una impresora 3D. Este robot, además, tiene la característica de su avanzada forma de caminar, ya que, como sus desarrolladores afirman, está inspirada en la forma biológica que utilizan los humanos, permitiendo una mejor interacción entre los robots y las personas. Romeo, desarrollado por Aldebaran Robotics (al igual que el Nao), se perfila como uno de los mejores asistentes robóticos personales, dedicados al cuidado y atención de las personas. Está fabricado con fibra de carbono y plástico para reducir el riesgo de lesiones a las personas por contacto. Además, tiene una constitución como la de un niño de en torno a los ocho años de edad. Se estima que para

el año 2017-2019 será completamente operativo en entornos domésticos.

Por último, tenemos el nuevo robot llamado iCub (figura 1.5(a)). iCub ha sido desarrollado por el consorcio de varias universidades europeas denominado RobotCub. El nombre de este robot es un acrónimo de las palabras *Cognitive Universal Body*, o en español: Cuerpo Universal Cognitivo. La motivación tras el diseño tan humano que posee el robot está en la hipótesis de la cognición corpórea o encarnada, la cual defiende que la interacción con otros humanos o humanoides (en este caso robots con aspecto humano) juega un papel fundamental en el desarrollo cognitivo de las personas. Uno de los principios que sostiene esta hipótesis es que los bebés adquieren más habilidades cognitivas cuando interactúan con su entorno, y en especial con otros humanos utilizando sus extremidades y sus sentidos, lo que por ende, acaba en una percepción más desarrollada de que el mundo está formado en su mayor parte por figuras humanas. De tal modo que una de las finalidades de este robot es ponerse a disposición de los humanos, en especial bebés y personas con graves enfermedades de deterioro psicológico, como el alzheimer, para ayudarles tanto a desarrollar sus capacidades cognitivas, como para ayudar a los médicos en los tratamientos para dichas enfermedades a través de la interacción con el humano.

Como robot que vamos a estudiar y sobre el que gira todo este proyecto tenemo el *Nao* (Figura 1.4(b)), desarrollado por Aldebaran Robotics. Desde el año 2008 es el robot oficial de la *Standard Platform League* en la competición robótica internacional de fútbol *RoboCup*. La *RoboCup* es un proyecto internacional cuyo objetivo es promover la investigación robótica a través de diferentes competiciones creadas específicamente para cada tipo de robot.



(a) (b)

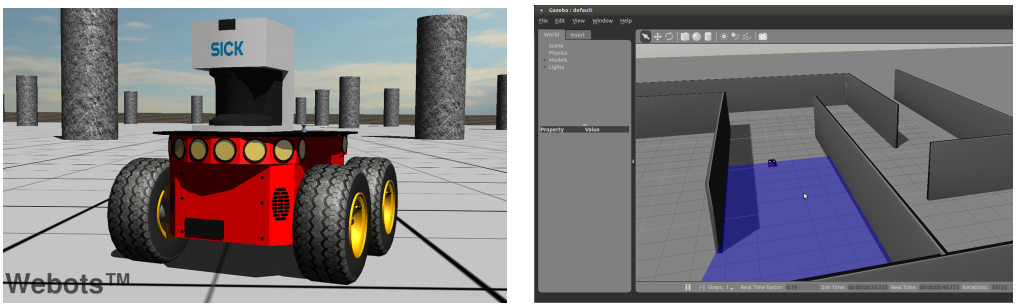
Figura 1.5: (a) *iCub* de RobotCub y (b) *Poppy*.

1.3. Simulación en robótica

Uno de los elementos más importantes en el desarrollo de comportamientos para robots en la actualidad son los simuladores. Los simuladores están diseñados para ayudar en la depuración del desarrollo software de los robots ofreciendo un entorno virtual realista y totalmente controlado por el usuario en el que se simulan tanto las propiedades físicas de los objetos (incluído el robot) como las acciones y reacciones resultantes de los diferentes elementos del entorno con el propio robot. De las muchas ventajas que ofrecen los simuladores, las más destacables pueden ser: la repetibilidad, la seguridad, el bajo coste y la posibilidad de trabajar con grupos de robots simultáneamente sin riesgo para los propios robots físicos. Conseguir un entorno controlado en la realidad puede llegar a ser una tarea muy compleja y costosa, tanto en tiempo invertido como en dinero. Factores como la luz, la humedad, la temperatura,... influyen de manera significativa en el entorno del robot en determinadas situaciones. Es por ello que los simuladores ofrecen una gran ventaja a la hora de probar y depurar software robótico.

A pesar de todas las ventajas que ofrecen los simuladores, en ocasiones es complicado emular de forma exacta las propiedades físicas de los modelos virtuales tanto de los robots como del entorno. Los simuladores trabajan con motores de físicas que realizan cálculos constantes de todas las variables del mundo. Factores como la gravedad, fuerzas presentes en el modelo y el entorno, rozamientos, masas, inercias, ... Todos ellos influyen de manera muy significativa a la hora de conseguir una simulación realista y fiable, por lo que hay que tener muy controladas las propiedades físicas tanto del robot que queremos desarrollar como de los objetos que iteractúan con él. Pequeños cambios en estas variables pueden suponer la pérdida total de realismo en la simulación, y hay que saber muy bien qué valores otorgar a cada elemento presente en la simulación. No obstante, los principales simuladores en la actualidad ofrecen infinidad de facilidades y documentación para hacer de nuestras simulaciones algo de lo que poder fiarnos.

Históricamente los simuladores no han sido muy bien acogidos por la comunidad robótica, debido a la imprecisión de los mismos. No obstante, la complejidad de este tipo de herramientas ha crecido tanto que a día de hoy disponemos de simuladores que son muy leales a la realiad. Con el tiempo se han ido mejorando aspectos como los motores de físicas, los problemas de visión de los robots, aumento de la precisión de los sensores, etc. Por lo que en la actualidad, ya no se piensa en desarrollar un robot sin utilizar un simulador para depurar y probar software destinado a las máquinas. Existen muchos simuladores hoy en día, desde aquellos comerciales como *Webots* (Figura 1.6(a)) o *Microsoft Robotics Developer*



(a) (b)

Figura 1.6: (a) Simulador *Webots*, y (b) *Gazebo*.

Studio [Jackson, 2007] a otros desarrollados por la comunidad de software libre. Un reciente simulador interesante es *MORSE*[3] (Modular OpenRobot Simulation Engine) [Echeverria *et al.*, 2011], con propósito educativo. Quizá los más extendidos son el simulador multi-robot bidimensional *Stage* [Gerkey *et al.*, 2003] y *Gazebo*[4]. En particular, *Gazebo* [Koening and Howard, 2004] es un simulador 3D que ofrece un rico entorno para el desarrollo y test de sistemas multi-robot con capacidad de simular prácticamente cualquier sensor como cámaras, láseres, sónares o incluso GPS, micrófonos, WiFi y altavoces en sus últimas versiones. *Gazebo* nació dentro del proyecto *Player/Stage/Gazebo* [Vaughan and Gerkey, 2007] (Figura **??**), pero recientemente *Willow Garage* y la *Open Source Robotic Foundation* (OSRF) han promovido su desarrollo como software independiente. Además, hay que tener en cuenta que *Gazebo* (Figura 1.6(b)) ha sido usado por la *DARPA* como plataforma de simulación para sus competiciones de programación para robots (DRC).

1.4. Robótica humanoide en la URJC

En la Universidad Rey Juan Carlos, el grupo de robótica, en el departamente de *Sistemas telemáticos y de computación* se lleva a cabo un proyecto de desarrollo de una plataforma robótica llamada JdeRobot. Esta plataforma provee las herramientas y mecanismos necesarios para la programación de robots y sensores de diversos tipos tanto reales como simulados. Actualmente, JdeRobot da soporte a dos vehiculos: pioneer y turtlebot, a sensores RGBD como el Kinect de Microsoft o el Xtion de Asus, haciendo mención especial al proyecto de final de carrera de Daniel Yagüe este mismo año, quien

[3] `http://www.openrobots.org/wiki/morse`
[4] `http://gazebosim.org`

ha dado soporte completo al cuadricóptero ArDrone de Parrot, diseñando y elaborando tanto el modelo y los plugins como herramientas para controlarlo en un entorno real y simulado. Además, de la aportación tanto de Borja Menendez (2013) como de Francisco Rivas (2010) para mejorar el soporte existente para el robot humanoide Nao en JdeRobot y que se va a ampliar en este proyecto. A la par se están desarrollando diversos proyectos encaminados a la ampliación de funcionalidades de la plataforma, como a su asentamiento y estabilidad. Un ejemplo claro es el desarrollo de herramientas para el control de sensores RGBD en entornos domésticos para el seguimiento de personas, o la migración hacia las librerías más actualizadas de terceros de las que hace uso JdeRobot tales como OpenCV, el propio Gazebo, Ice, ...

1.4.1. Nao en JdeRobot

El robot *Nao* simulado en *Gazebo* ha tenido soporte en JdeRobot gracias al Proyecto Fin de Carrera de Jorge Bermejo [Bermejo, 2010]. En él se desarrolló dicho soporte para que otros desarrolladores pudieran crear componentes JdeRobot que hicieran uso de dicho robot humanoide. De esta forma se podrían probar en simulación dichos componentes para luego llevarlos al robot real. El soporte fue realizado para la versión 0.9 de *Gazebo*, actualmente obsoleta.

También se ha usado el robot *Nao* dentro de la plataforma JdeRobot, en concreto en el proyecto fin de carrera de Francisco Rivas[5]. En este proyecto se desarrolló una forma de andar propia para el robot que fuera totalmente parametrizable con el fin de encontrar la marcha más eficiente posible, como ya logró Francisco Rivas con su proyecto de fin de carrera [Rivas, 2011] con Webots. Para lograrlo se diseñó una herramienta con la que poder controlar cualquier componente del robot, pudiendo así reproducir cualquier movimiento del mismo. Esta herramienta se llamó *NaoBody*, el primer servidor que podía comandar acciones a los actuadores del robot y recoger información proporcionada por los sensores. Corría encima de NAOqi, pero ejecutaba fuera del robot, de modo que la interacción con él se hacía a través de la red. Además, esta arquitectura daba la posibilidad de conectar este servidor tanto al robot real como a uno simulado con el simulador oficial del *Nao*, *Webots*.

Por último, cabe destacar el aporte realizado por Borja Menéndez con su trabajo de fin de master [Menéndez, 2013] el cual refactorizó el modelo existente del Nao creado por Jorge Bermejo, implementando además un componente basado en visión artificial que permitá que el Nao siguiera una pelota con la mirada utilizando filtros de color para aislar

[5] `http://jderobot.org/Frivas-pfc-itis`

la pelota en el espacio y técnicas de visión artificial para localizarla. Además desarrolló un componente llamado VisualHFSM el cual permite programar el comportamiento del robot a partir de autómatas de estado finito con un interfaz gráfico intuitivo. Esta técnica de programación es bastante popular cuando se quiere introducir el desarrollo sobre robots a niños o a adolescentes, ya que permite de forma muy intuitiva y visual, arrastrando módulos que encapsulan un comportamiento como: saludar, andar, bailar, ... a un canvas como si de un IDE se tratara, de tal modo que la interconexión entre esos módulos sea trivial.

Soporte para el robot humanoide Nao en Gazebo 0.7

El primer soporte que se dió en JdeRobot del robot Nao fue llevado a cabo por Jorge Bermejo [Bermejo, 2010] e incluía como novedad la creación de un modelo del robot humanoide para el simulador Gazebo en su versión 0.7. En ese proyecto se desarrolló todo el modelo del robot desde cero modelizando a mano las mallas del robot simulado y generando todos los cuerpos sólidos del nao manualmente. Además se desarrolló un driver genérico para controlar todas las articulaciones utilizando una librería escrita en C específica para gazebo llamada *libgazebo*. A pesar de que aquí nace el soporte del Nao en JdeRobot, las limitaciones del momento no permitían movimientos complejos del robot al no ser el modelo descrito un modelo exacto, por lo que hubo que esperar un año, a la aportación de Francisco Rivas para que esto fuera posible. De los frutos de ese proyecto nacieron los siguientes desarrollos sobre el humanoide Nao, y que actualmente tiene un soporte muy poco maduro, pero bastante estable y completo.

Soporte para el robot humanoide Nao en Webots y métodos de caminar

Un año más tarde, en 2011, tuvo lugar al aportación que m ás similitudes tiene este proyecto, la de Francisco Miguel Rivas [Rivas, 2011]. En su proyecto de fin de carrera, Francisco dió soporte al robot Nao para el simulador Webots, además de desarrollar herramientas muy completas diseñadas para la locomoción del robot, tanto real como simulado en la plataforma JdeRobot. La aportación de Francisco es una de las más importantes en cuanto a robots humanoides en la URJC, ya que además de desarrollar dicho soporte, diseñó el modelo matemático que describe las formas de caminar del Nao, y a partir del cual nace este proyecto. Actualmente en JdeRobot se conservan las herramientas desarrolladas en el proyecto de Francisco, que van a ser refactorizadas próximamente con los cambios realizados por el presente proyecto.

Soporte para el robot humanoide Nao en Gazebo 1.2 y generación de comportamientos mediante autómatas

En 2013, Borja Menéndez [Menéndez, 2013] presenta su proyecto de fin de máster, en el cual se desarrolló la refactorización del modelo propuesto por Jorge Bermejo en 2010. Esta refactorización se realizó para dar soporte a la versión de Gazebo 1.8, que ha sido la versión oficial de JdeRobot hasta hace pocos meses, cuando se dio el salto a Gazebo 5. La propuesta de Borja fue remodelar por completo el modelo existente utilizando los nuevos formatos descriptivos para los robots simulados en Gazebo, así como la refactorización de los plugins, separando la funcionalidad de cada articulación en un plugin independiente. De esa base parte este proyecto. Además como aportación extra, Borja implementó un sistema de programación de comportamientos para el robot humanoide mediante autómatas jerárquicos de estado finito, incluyendo un interfaz gráfico para facilitar esta tarea.

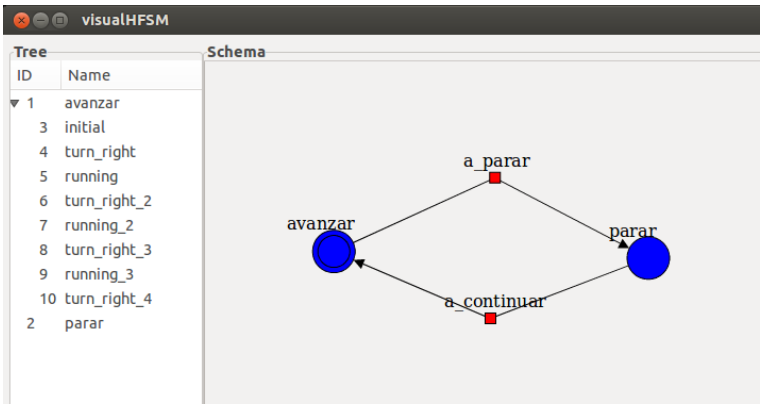


Figura 1.7: Diseño del autómata de ejemplo hecho con VisualHFSM.

Artículos publicados

Fruto de estos tres proyectos antes mencionados ([Bermejo, 2010], [Rivas, 2011] y [Menéndez, 2013]), se han publicado algunos artículos en relación con la robótica humanoide. Algunos ejemplos son el artículo publicado para *Proceedings of the XIV Workshop en Agentes Físicos (WAF-2013)* en 2013, en el que se expone lo desarrollado en el trabajo de fin de máster de Borja ([Menéndez, 2013]). El WAF (*Workshop de Agentes Físicos*) es un foro de encuentro en el que se intercambia información y experiencias relacionadas con la aplicación del concepto de agente en entornos físicos, especialmente en el control y coordinación de sistemas autónomos: robots, robots móviles, procesos industriales o sistemas complejos. Este *Workshop* se lleva celebrando por toda España desde el año 2000.

Otro de los artículos más relevantes publicados en relación a los robots humanoides fue el de RoboCity2030 en 2013 [Rivas *et al.*, 2013]. En él se describía la programación de un robot humanoide con comportamientos sociales para fines terapéuticos utilizando JdeRobot y la arquitectura BICA, además de los desarrollos realizados por Francisco Rivas y Borja Menéndez. El proyecto RoboCity2030 tiene como objetivo desarrollar e innovar en la integración de aplicaciones para robots destinados al servicio, para aumentar así la calidad de vida de los ciudadanos en áreas metropolitanas. El proyecto RoboCity2030 lleva dando conferencias por todo el mundo desde 2004, habiéndo celebrado desde entonces hasta la fecha más de 120 de ellas. Otro ejemplo es el artículo publicado en 2011 por Francisco Rivas [Rivas *et al.*, 2011] para el *Proceedings of Robot2011 III Workshop de Robótica: Robótica experimental*, donde se exponía también un resumen de su proyecto de fin de carrera.

El presente documento está estructurado de la siguiente forma: priemro se detallan los objetivos y el plan de trabajo seguidos para el desarrollo de este proyecto (Capítulo 2); acto seguido se detalla el contexto tecnológico en el que se enmarca el proyecto y las herramientas utilizadas para llevarlo a cabo (Capítulo 3); lo que da paso a la exlicación de las soluciones aportadas al problema planteado en este proyecto (Capítulos 4 y 5); para finalizar con las conclusiones a las que se ha llegado tras el desarrollo y los problemas encontrados durante el mismo, además de presentar algunas líneas de investigación que abre la consecución de los objetivos marcados en el proyecto (Capítulo 6).

Capítulo 2

Objetivos y plan de trabajo

Una vez comprendido el contexto en el que se va a desarrollar este proyecto, a continuación se plantea y describe el problema abordado, así como la y metodología aplicada al desarrollo software de las soluciones programadas.

2.1. Descripción del problema

El objetivo principal de este proyecto es realizar la mejora del soporte en JdeRobot del robot humanoide Nao en Gazebo-5, incluyendo la generación de formas de caminar (utilizando ondas acopladas para cada articulación del robot). Todo esto está diseñado para funcionar en simulación.

Para realizar este desarrollo de foma más eficiente y cómoda, se han dividido los dos objetivos principales en varios subobjetivos: los tres primeros relacionados con la mejora del soporte para JdeRobot y los dos últimos relacionados con la capacidad de caminar:

1. Migrar el modelo existente a la última versión del simulador elegido para este proyecto: Gazebo. La última versión de este simulador (v5.1) incluye cambios importantes en los motores de físicas ODE y BULLET, los cuales simulan el comportamiento de los objetos en un escenario virtual. Dichos cambios hacen necesaria la refactorización tanto del modelo como de los plugins que lo controlan. Por otra parte, el salto a esta versión permite más versatilidad a la hora de simular escenarios, ya que, entre otras cosas, se incluye un editor de objetos y otro editor de mundos, que amplían la funcionalidad del simulador en gran medida.

2. Mejorar el modelo existente del robot dentro del entorno de simulación. El Robot actual dispone de un modelo de simulación con valores físicos estandar para su funcionamiento por articulaciones individuales. El objetivo es refactorizar este modelo con valores físicos precisos, tales como los tensores de inercia de cada articulación que definen el movimiento del robot al aplicársele una fuerza interna o externa, o estando en estado de reposo. Además, se ha propuesto mejorar el robot visualmente para que se parezca aún más al robot real, ya que las mallas de visión actuales, a pesar de tener un aspecto similar al del Nao, no son del todo exactas.

3. Refactorizar los plugins que controlan el modelo y dotan de movimiento al robot, generando así un control de bajo nivel para el robot humanoide. Como se ha especificado en el primer punto, es necesaria la refactorización de todos los plugins implicados en el movimiento debido a los cambios realizados en los motores de físicas. Con este cambio de API, algunas de las instrucciones de los plugins actuales han quedado obsoletas, generando comportamientos extraños en el robot, tales como movimientos periódicos infinitos, saltos cuando no actúa ninguna fuerza sobre el robot y movimientos residuales en estático. Se han incluído todos los plugins que había en el modelo original, mejorando cada uno por separado.

4. Basándonos en el proyecto de fin de carrera de Francisco Rivas, remodelizaremos la generación de caminatas para el Nao, de forma que de soporte para Gazebo en su última versión, lo que genera un control de alto nivel para el humanoide. Actualmente, la generación de caminatas no está implementada en JdeRobot, ya que el simulador que se utiliza no es Webots, sino Gazebo. Por lo tanto se realizará un generador de caminatas basado en ondas acopladas que será compatible con los cambios expuestos en los puntos anteriores.

5. Desarrollar un interfaz gráfico que de control sobre la búsqueda de caminatas de forma sencilla. Dicho editor deberá permitir generar candidatos a caminata realizando una búsqueda rastrillada en un espacio de búsqueda determinado. Esta automatización deberá además evaluar cada movimiento candidato y medir cuán bueno es en función de una función salud que tendrá en cuenta varios aspectos relevantes a la hora de caminar.

Todos estos puntos serán validados mediante pruebas exhaustivas de modo que se asegure el correcto funcionamiento de todos los componentes implicados en el proyecto. Además, se asegurará que todos los componentes desarrollados sean compatibles con el

entorno JdeRobot para su futura incorporación a la plataforma, de modo que puedan ser utilizados, estudiados y mejorados por otras personas dado su carácter de código libre.

2.2. Requisitos

Además de los objetivos arriba listados se querrá satisfacer una serie de requisitos para considerar válido el desarrollo del proyecto, a saber:

1. El robot simulado deberá ser funcional en la plataforma Gazebo, versión 5.1 y deberá ofrecer como requisito mínimo el acceso y la manipulación de todas las articulaciones presentes.

2. El robot Nao simulado tendrá que ofrecer acceso a los mismos interfaces que el robot real.

3. El robot deberá ofrecer estabilidad a la hora de mover cada articulación por separado, ajustándose a cómo se comportaría si se aplicaran los mismos movimientos al robot real.

4. El interfaz gráfico desarrollado deberá ofrecer de forma lo más sencilla posible la generación, evaluación y almacenamiento de caminatas para el robot. Además de ofrecer información actualizada de cada candidato evaluado en tiempo real.

5. El proyecto entero estará desarrollado bajo la plataforma JdeRobot 5.3, programando los componentes en C++ y utilizando bibliotecas gráficas lo más modernas posibles (como Qt4).

2.3. Metodología

Para la realización de este proyecto se ha optado por seguir un modelo de desarrollo en espiral basado en prototipos, ya que se ha considerado que se ajusta más al tipo de proyecto que se iba a realizar. Este modelo de desarrollo se basa en la idea fundamental de repetir las mismas tareas iterativamente de forma que en cada ciclo se aumenta la complejidad del proyecto, permitiendo la generación de prototipos funcionales al final de cada uno. Debido a que se esperaban cambios de requisitos constantes, comunes en este tipo de proyectos de investigación, se optó por este modelo de desarrollo ya que permitía libertad a la hora de

reajustar funcionalidades específicas en cada ciclo, además de hacer el proyecto escalable tanto en funcionalidad como en complejidad sin afectar al tiempo de desarrollo.

Todo esto sumado a la posibilidad de llevar un control por hitos al final de cada iteración además de las reuniones semanales realizadas durante todo el proceso de desarrollo hacen de este modelo el ideal en nuestro caso, dado que cada reunion ofrecería realimentación para lo hecho teniendo así en todo momento el proyecto bajo control.

Como herramienta asociada al proyecto, se ha dispuesto de un mediawiki en la página de JdeRobot que serviría como cuaderno de bitácora donde plasmar todo lo hecho durante el proceso de desarrollo. Dicho mediawiki se puede econtrar [1]

El modelo en espiral se realiza por ciclos que corresponden a las diferentes fases del proyecto software. Cada ciclo se compone de cuatro fases principales, en las que se realizan diferentes tareas. En la figura 2.1:

- **Determinar los objetivos**. Se definen las necesidades que debe cumplir el software a desarrollar en cada iteración, contemplando los objetivos finales. Esto hace que la complejidad del proyecto y el coste del ciclo avancen en función del tiempo
- **Evaluar alternativas**. Se deben tener en cuenta las diferentes formas de llegar al objetivo proponiendo alternativas a los distintos elementos del proyecto. Además se deben considerar los riesgos que puedan existir e intentar reducirlos al máximo.
- **Desarrollar y verificar**. Teniendo en cuenta las alternativas propuestas en la fase anterior, se debe elegir la mejor de ellas y desarrollarla para llegar a los objetivos propuestos, para que finalmente el producto de ese desarrollo sea verificado con las pruebas pertinentes.
- **Planificar**. Con los resultados de las pruebas realizadas en la fase anterior, se ha de planificar la siguiente iteración revisando los posibles errores cometidos durante el ciclo actual y se comienza con un nuevo ciclo.

Todas estas fases se corresponden con lo realizado a lo largo de este proyecto, coincidiendo cada ciclo con los puntos que a continuación se desarrollan. Para cada ciclo se han marcado una serie de hitos a cumplir modularizando así el trabajo a realizar.

[1] `http://jderobot.org/Fperez`

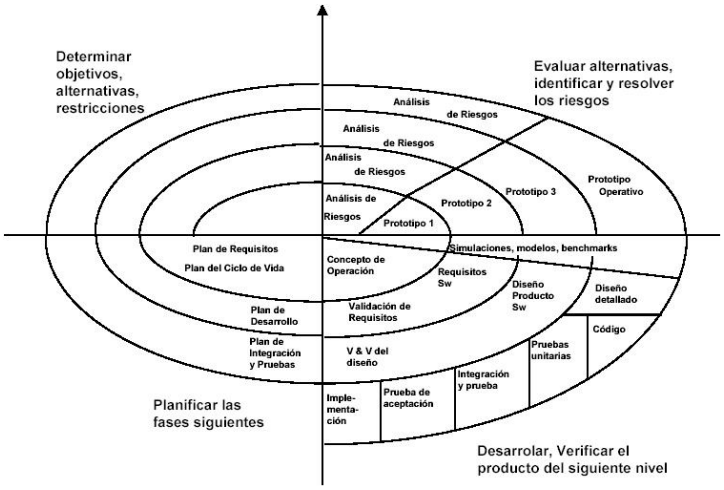


Figura 2.1: Modelo de desarrollo en espiral.

2.4. Plan de trabajo

En cada ciclo del desarrollo se han ido proponiendo nuevos requisitos que aumentaban la funcionalidad del producto a desarrollar hasta conseguir los objetivos finales marcados al principio del proyecto. Esta fase de especificación de requisitos y de cumplimiento de hitos se ha visto realimentada por las reuniones semanales con el tutor del proyecto, quien ha guidao el desarrollo ciclo a ciclo.

1. **Familiarización con el entorno de programación y el simulador**. Esta primera fase de familiarización consiste en tratar de aterrizar sobre el entrono de programación de JdeRobot instalando todo el software disponible y tratando de compilarlo y ponerlo en funcionamiento. Para este fin disponíamos de un componente básico, que a modo de ejemplo trata de explicar tanto la estructura interna de los componentes desarrollados en JdeRobot, como los estándares de programación y de comunicaciones utilizados en la plataforma. Adicionalmente, habrá que estudiar de las librerás gráficas que permiten desarrollar interfaces de usuario asociadas a los componentes que se vayan a desarrollar. Para ello existen dos alternativas principales usadas en JdeRobot: GTK y QT.Como último objetivo de esta fase, habrá que entender cómo funciona el simulador, en nuestro caso: Gazebo-5. Para ello ya hay ejemplos desarrollados en la plataforma que sirven como buen punto de partida para tomar contacto con todo el entorno simulado, ya que la estructura interna de los plugins es prácticamente la misma para todos los robot simulados de la plataforma. De esta forma. el objetvio al salir de esta etapa es ser capaz de desarrollar un modelo simulado asociado a un

plugin que dictará su comportamiento.

2. **Refactorización del modelo físico y el control humanoide**. En esta fase se estudiará el modelo existente del humanoide, además de toda la documentación oficial del robot para la corrección de errores y la inclusión de mejoras en los parámetros físicos del modelo. Además se han de conseguir unas mallas más precisas para el humanoide que lo doten de realismo visual y físico respetando tamaños y propiedades físicas del modelo. Acto seguido, habá que refactorizar el soporte existente para hacerlo compatible con la última versión del simulador. Para ello hay que estudiar los cambios realizados en el API de Gazebo, incorporarlos al soporte existente y porbar su funcionamiento en el simulador articulación por articulación hasta conseguir una simulación estable y realista. Estos plugins ofrecen las interfaces de comunicación entre el simulador y la plataforma de desarrollo, de forma que se pueda tener control total sobre el robot simulado con la creación de componentes específicos para ese fin de forma sencilla.

3. **Estudio de la parametrización de las caminatas e implementación de la búsqueda de caminatas**. El objetivo es conseguir entender la parametrización de una caminata en un robot bípedo a partir de la reducción de posibles parámetros involucrados en la acción de caminar. Para ello han de estudiarse los diferentes grados de libertad de cada articulación, eliminar los que no afecten a una caminata estable, e intentar reducir todo lo posible la parametrización de una caminata creando dependencias entre los grados de libertad de las articulaciones involucradas. Una vez que se disponga de control total sobre todas las articulaciones del robot simulado y se comprenda la parametrización de las caminatas en un robot bípedo, se ha de desasrrollar un nuevo plugin que sirva de interfaz de comunicacion entre el componente generador de caminatas que se va a desarrollar, y el robot simulado. Este plugin deberá ser capaz de recibir las caminatas parametrizadas generadas por el componente JdeRobot, procesarlas, generar las ondas pertinentes para cada articulación, acoplar dichas ondas y comandar a cada articulacion un valor en el tiempo a partir de los valores obtenidos. Una vez conseguido todo lo anterior, el último ciclo consistirá en crear una herramienta para JdeRobot, que permita al usuario interactuar con las nuevas funcionalidades que se hayan desarrollado para entonces para el humanoide. De este modo, el objetivo será generar una herramienta con un interfaz gráfico asociado, que permita de forma sencilla generar los parametros de una caminata utilizando un algoritmo de búsqueda rastrillada. Una vez generado el candidato a caminata, el componente deberá entregar dichos parámetros al plugin

de caminatas que se haya generado en el ciclo anterior y esperar la evaluación de este obteneiendo la información que el simulador le ofrezca para cada candidato, deshechando los peores.

4. **Validaciones experimentales**. El objetivo de esta fase final es dejar en funcionamiento el algoritmo de búsqueda de modo que evalue un gran número de candidatos a caminata y probando así el funcionamiento de todo lo que se desarrolle durante el proyecto.

Capítulo 3

Infraestructura

Este capítulo recoge toda la tecnología utilizada para el desarrollo de este proyecto *software*, así como una descripción superficial del robot humanoide con el que se ha trabajado en simulación. Esto sitúa el contexto tecnológico de este proyecto.

3.1. Robot Nao

El robot Nao[1] es un robot autónomo, programable y de mediana estatura desarrollado por la empresa Aldebaran Robotics, con sede en París. En el año 2008 el Nao reemplaza al Aibo de Sony como la plataforma estándar para la RoboCup Standard League, una competición internacional con robots autónomos que juegan al fútbol, siendo esta versión del robot la primera estable, denominada V2. En 2009, Aldebaran Robotics pone a disposición de las universidades y de la RoboCup una nueva versión del robot con las correcciones de los problemas de fiabilidad descubiertos durante el desarrollo de la RoboCup de 2008 denominada V3. Finalmente en 2012 aparece la versión V4, que es la versión utilizada para el desarrollo de este proyecto. Actualmente la versión del nao es la V5 o Nao Evolution que se puede diferenciar por la forma ligeramente diferente de su cabeza.

Se puede ver una imagen del robot con la descripción de su cuerpo en la Figura 3.1. Sus principales características son:

- 58 cm de altura.
- 5.4 kg de peso.

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

- Autonomía de 90 minutos (60 minutos caminando).
- Grados de libertad: de 21 a 25.
- CPU: Intel ATOM Z530 a 1.6 GHz.
- 1 GB de RAM.
- 10 GB de almacenamiento.
- Plataforma: Linux.
- Lenguajes de programación: C++, C, Python.
- Ethernet, WiFi, USB.
- Sensor de inercia, infrarrojos.
- Sensores de ultrasonidos: cuatro.
- Sensor de infrarrojos.
- Multitud de LED's.
- Cuatro micrófonos, dos altavoces, dos cámaras.
- Sintetizador de voz.

Para programar el robot Nao el fabricante ofrece un API y una serie de herramientas que permiten la creación de módulos programables de manera muy intuitiva. El robot Nao funciona bajo la plataforma Linux y puede usarse para programar el robot un kit de desarrollo de software (o SDK) llamado NAOqi, el cual ofrece la posibilidad de programarlo en C, C++, Ruby, Python y Urbi. NAOqi está disponible tanto para Windows como para Linux.

Este SDK está basado en una arquitectura cliente-servidor donde es el propio NAOqi quien actúa como servidor, los diferentes módulos son incorporados como bibliotecas o como *Brokers* y la comunicación se hace a través de IP con NAOqi. Estos nuevos Brokers se conectan al Broker principal, llamado `MainBroker`, como se muestra en la Figura 3.2. Gracias a esta arquitectura se tiene la posibilidad de ejecutar el código tanto directamente sobre el robot como desde una máquina remota.

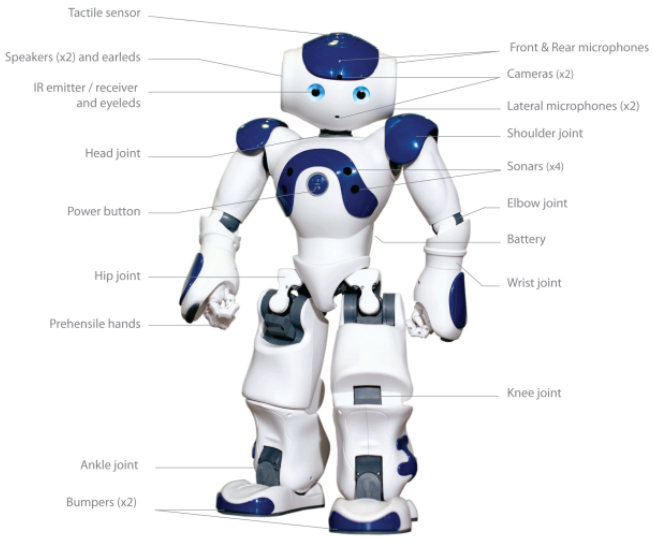


Figura 3.1: Descripción del robot humanoide Nao.

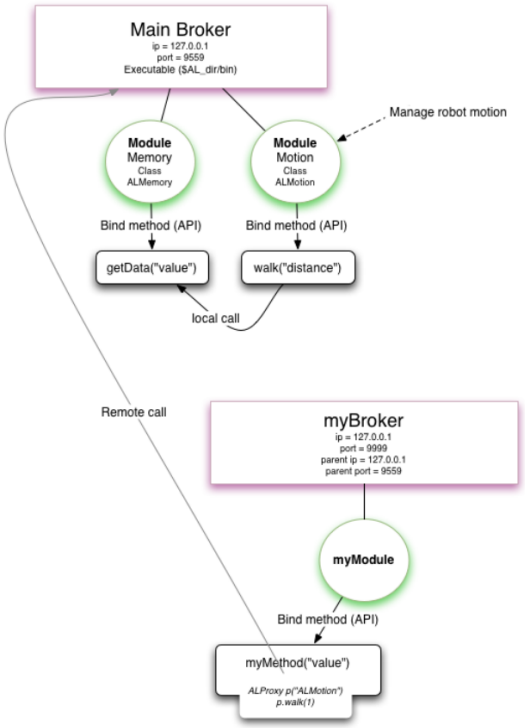


Figura 3.2: Arquitectura de NAOqi usando Brokers.

3.2. Qt y QtCreator

Qt[2] es un conjunto de bibliotecas gráficas multiplataforma para desarrollar interfaces gráficas de usuario (GUI). En nuestro proyecto se ha hecho uso de la versión 4 que es la más estable hasta el momento. Debido a su caracter multiplataforma y multilenguaje dispone de diferentes API's en función del lenguaje de desarrollo que se utilice. Así por ejemplo diponemos de la versión para C++ llamada Qt, la versión para Java llamada QtJambi o la versión para python llamada PyQt, entre otras. Dispone tanto de versión gratuita como de pago y ambas están bajo la licencia *GNU LGPL*[3].

Qt ofrece infinidad de opciones a la hora de desarrollar interfaces gráficos divididas en módulos. Así por ejemplo, disponemos de funcionalidades tales como la creación de gráficas tanto en 2D como en 3D, módulos de comunicaciones inalámbricas como Bluetooth, modulos para administrar contenido multimedia (videos, fotos, radio, cámaras, ...) e incluso módulos para creación de *widgets* para la web, entre muchos otros tanto de alto como de bajo nivel.

Como la mayoría de bilbiotecas gráficas, Qt ofrece una gran cantidad de elementos de interfaz tales como: botones, *scrolls* verticales y laterales, cuadros de texto, separadores, paneles LCD, y un largo etc. La base de casi todos estos elementos se encuentra en el módulo QObject, del cual dependen casi todo el resto de elementos que ofrece Qt.

La forma de interactuar con estos elementos es mediante el uso de eventos. Cuando ocurre un evento, tal como presionar un botón o cerrar una ventana, la señal apropiada será emitida por el elemento en cuestión afectado. Esta señal podrá ser capturada y conectada a una función (un *callback*) mediante el manejador de señales. Esta función será llamada cuando la señal sea detectada. Así, para que un botón realice una determinada acción solo tendrá que conectar dicha señal a la función correspondiente.

Qt utiliza su propio hilo de procesamiento durante su ejecución. De tal forma que tiene un manejador de eventos propio que responde a cada solicitud realizada por el usuario a través de su interfaz. Esto supone una gran ventaja a la hora de realizar componentes asíncronos en JdeRobot, ya que no tendremos que esperar al gui para realizar otras tareas útiles, optimizando así el tiempo de ejecución de nuestros programas.

Además, Qt dispone de API para el desarrollo en dispositivos móviles y tablets gracias a la gran variedad de módulos que ofrece. Esto hace que la herramienta para el caso de JdeRobot sea muy potente al ofrecer la posibilidad de desarrollar las mismas interfaces para

[2] `http://www.qt.io`

[3] `http://www.gnu.org/licenses/old-licenses/lgpl-2.1.html`

Figura 3.3: Implementación multiplataforma de interfaces gráficas con Qt.

un componente en diferentes plataformas. Debido al carácter modular y exportable de los componentes de JdeRobot, el hecho de que se puedan implementar interfaces gráficos para diferentes plataformas facilita en gran medida el desarrollo de aplicaciones móviles para los componentes existentes y futuros. En la figura 3.3 se puede apreciar perfectamente.

Para un desarrollo más rápido y más cómodo de las interfaces gráficas realizadas en el proyecto, se ha utilizado como herramienta de ayuda el IDE (*Integrated Development Environment*) llamado *QtCreator*, el cual ofrece al desarrollador la posibilidad de crear interfaces de usuario con tan solo arrastrar elementos a un canvas. Todo esto es tan parametrizable y tan flexible como la programación manual, y además ofrece la comodidad de emplazar los elementos de forma visual e intuitiva para el usuario sin tener que perder tiempo compilando y ejecutando el programa cada poco tiempo para comprobar los resultados. Este IDE es muy configurable y muy potente, ya que además de poder emplazar los diferentes elementos con un *click*, permite editar las señales que emite o recibe cada elemento del interfaz de forma gráfica también. El código es autogenerado según se van añadiendo elementos a la interfaz, lo cual puede hacerse un poco engorroso de modificar, pero es bastante limpio si lo comparamos con otros IDE que realizan la misma tarea.

Todas estas ventajas citadas, sumadas al caracter multilenguaje de Qt, han sido claves a la hora de decidir la forma de codificación de la interfaz de usuario de nuestro componente. Esta decisión ha supuesto un ahorro considerable de tiempo de estudio de la API, así como de codificación del componente.

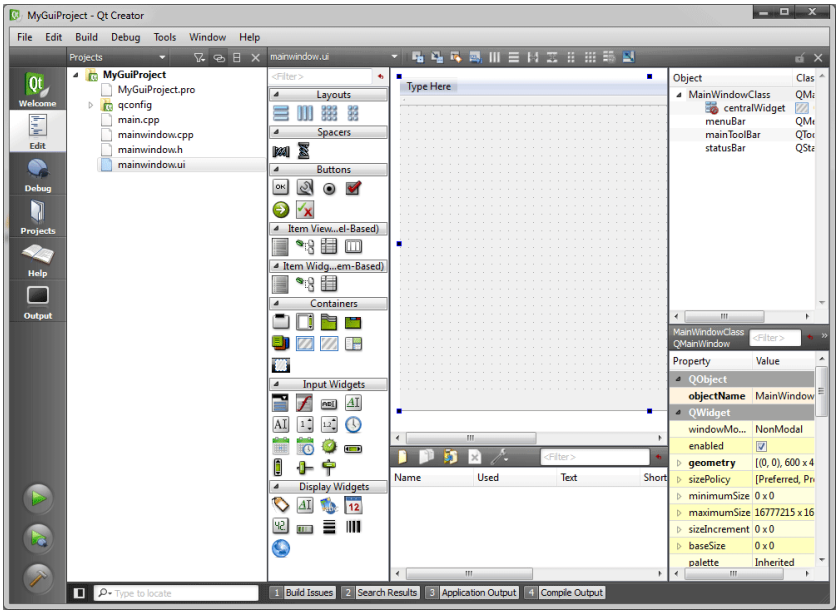


Figura 3.4: Interfaz de usuario del IDE QtCreator.

3.3. Gazebo

Gazebo[4] es un simulador 3D que ofrece un entorno para desarrollar y probar sistemas multirrobot de manera sencilla. Es capaz de simular de manera realista muchos tipos de sensores y actuadores, entre los que destacan cámaras, láseres, sónares, GPS, altavoces, etc., todo ello gracias, también, al motor de física que utiliza (*ODE*, *Open Dynamics Engine*) y a OpenGL (*Open Graphics Library*). Además, da soporte a multitud de robots comerciales como el Turtlebot o el Pioneer 2-DX. Es también software libre, ya que está licenciado bajo la licencia Apache[5].

Gazebo se ha convertido en uno de los simuladores más famosos entre la comunidad robótica debido a su completitud y precisión. En la actualidad es uno de los simuladores más completos en cuanto a características que ofrece, y al ser de código abierto, tiene detrás una comunidad que mejora el software poco a poco. El hecho de que sea un simulador multirrobot dota de una gran utilidad a Gazebo, ya que se pueden simular infinidad de situaciones en las que uno o varios robots se vean involucrados y estudiar su comportamiento simultáneamente. Además, Gazebo ofrece la capacidad de ejecutarse de forma distribuida, lo que supone una gran ventaja frente a otros simuladores, al poderse repartir las cargas de simulación (ya sabemos que es un proceso pesado debido a la gran

[4] `http://gazebosim.org`
[5] `http://www.apache.org/licenses/LICENSE-2.0.html`

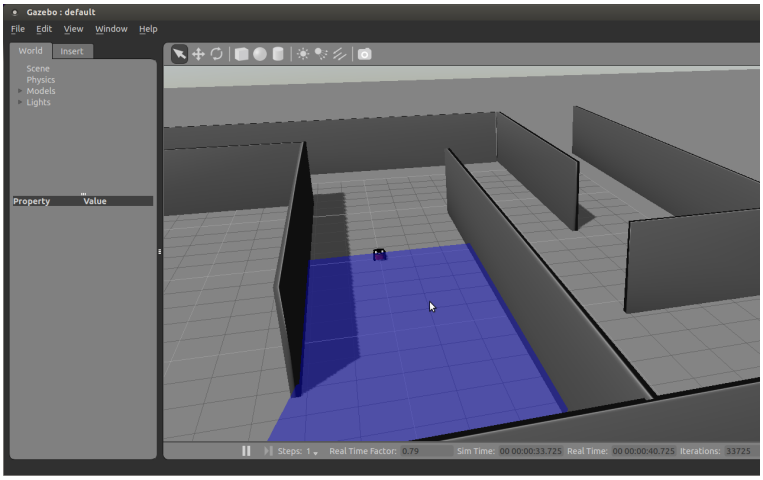


Figura 3.5: Ejecución de Gazebo simulando el robot Pioneer 2-DX con el láser y las cámaras.

cantidad de factores que influyen) entre varios computadores. Se puede ver un ejemplo de ejecución de Gazebo en la Figura 3.5.

La inclusión de robots en Gazebo es, en esencia, bastante sencilla. Basta con tener un fichero descriptivo del modelo robótico que se quiere simular en un formato de archivo específico: el formato SDF. Este formato es un formato XML que describe objetos y entornos robóticos para simulación. Originalmente fue creado como una parte de Gazebo, pero hoy en día se ha convertido en un formato independiente para este fin. Este formato de archivo describe elementos tales como iluminación, terrenos, objetos estáticos y dinámicos y hasta físicas. Debido a la alta facilidad de comprensión del formato SDF, la descripción de un robot o un entorno robótico está muy bien definida y estructurada. En el caso de este proyecto, se han tenido que especificar tanto el modelo del Nao en SDF, como un mundo simple sobre el que el robot pueda caminar. En el modelo del robot se contemplan propiedades como la masa de los cuerpos que lo componen, los valores de inercia de esos cuerpos, las articulaciones que el robot posee y algunos otros valores físicos tales como los rozamientos o las amortiguaciones de las articulaciones. Los modelos que acepta Gazebo son primitivas geométricas básicas como cubos esferas o cilindros, por lo que para conseguir un aspecto visual más cercano a la realidad, a esas primitivas hay que aplicarles una imagen que servirá como piel. Estas imágenes son imágenes COLLADA [6] (ficheros .dae).

Por último, el modelo del Nao especifica los plugins, o bibliotecas dinámicas, desarrolladas para dotar de control y movimiento al robot, y desarrollados utilizando la

[6] `https://collada.org`

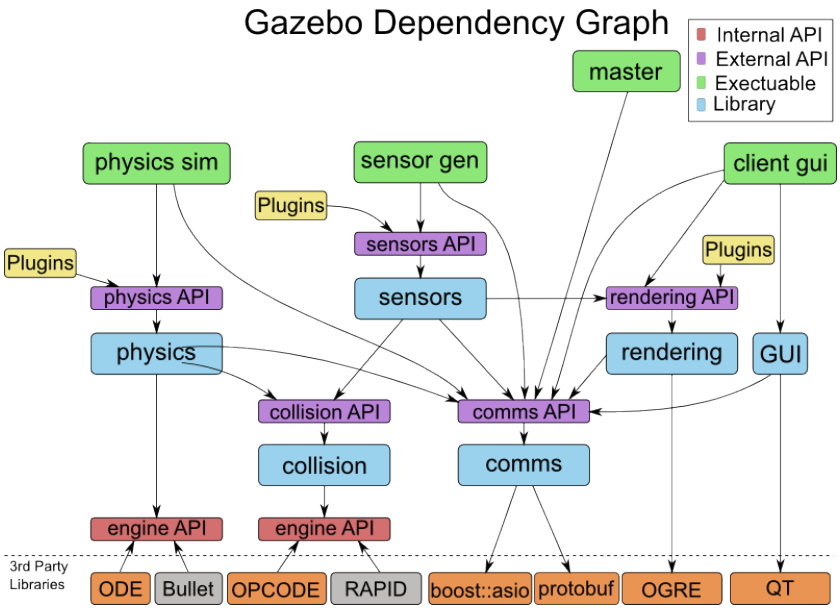


Figura 3.6: Diagrama de dependencias *software* en Gazebo.

API de Gazebo. Se puede ver un diagrama explicativo de la arquitectura de Gazebo en la Figura 3.6.

3.4. Blender

Blender [7] es un programa libre y de código abierto dedicado al modelado de figuras en tres dimensiones. Blender además permite modelar, texturizar, animar, renderizar, simular efectos físicos y hasta edición de audio y vídeo, además de su uso fundamental, el desarrollo de videojuegos; gracias a su motor interno.

Las principales características de Blender son:

- Multiplataforma. Funciona bajo Windows, MacOS, GNU/Linux, Solaris FreeBSD e IRIX.
- *Software* de código abierto y libre.
- Funcionalidades principales: modelado 3d, mapeado UV, texturizado, edición de gráficos rasterizados, animación esquelética, simulación de fluidos y humo, simulación de partículas, simulación de cuerpos suaves, escultura, animación, seguimiento de

[7] `http://www.blender.org`

cámara, renderizado, iluminación, topología dinámica, edición y composición de video, etc.

- Su versión más actual es la 2.74, estando ya la 2.75RC para su uso.

A pesar de ser una herramienta muy potente, la curva de aprendizaje de Blender es muy pronunciada debido a su interfaz de usuario poco amigable. Debido a la gran cantidad de opciones de que dispone este *software*, y la organización caótica de su interfaz de usuario, aprender a utilizar Blender requiere de un período de tiempo más largo que el resto de editores en 3D análogos como Maya o 3DStudio Max. La parte positiva de esto, es que el interfaz es altamente configurable, por lo que una vez que se domina el uso de las herramientas que ofrece, puede resultar realmente productivo si se configura una buena interfaz de usuario.

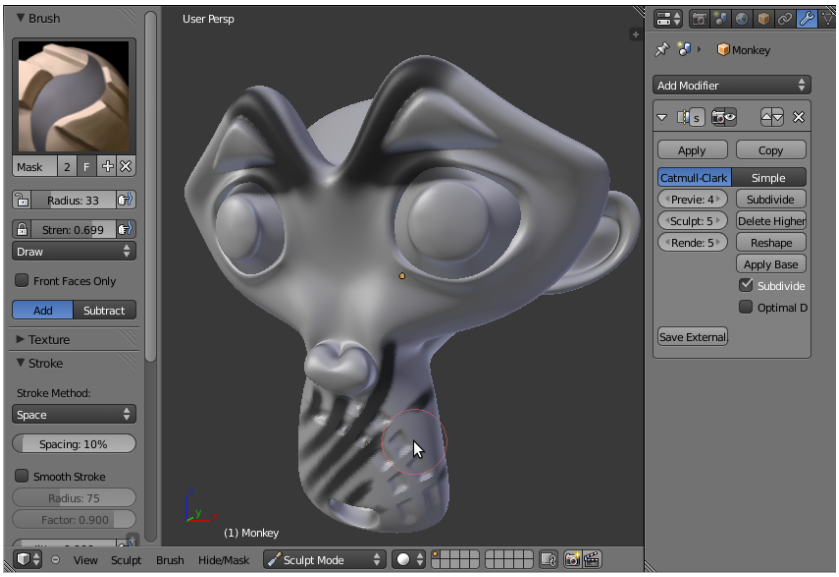


Figura 3.7: *Software* de modelado en 3D, Blender.

Para este proyecto en concreto, el uso de Blender se ha limitado a dividir el modelo en 3D del humanoide Nao que ofrece Aldebaran en las diferentes partes de las que se compone el Nao simulado, además de aplicarles color a dichas partes de forma manual utilizando las herramientas de texturizado que ofrece este *software*.

3.5. MeshLab

MeshLab [8] es un *software* avanzado de procesamiento de mallas en tres dimensiones. La funcionalidad que ofrece este *software* es el procesamiento y la gestión de mallas en 3D grandes y no estructuradas a priori (cuerpos no formados por primitivas geométricas 3D), ofreciendo para ello herramientas para editar, limpiar, sanear, renderizar y convertir este tipo de mallas. Además ofrece gran cantidad de información técnica sobre estas mallas como sus propiedades físicas, geométricas y visuales.

Las principales características de MeshLab son:

- Multiplataforma: funciona bajo Windows, MacOS, GNU/Linux y en iOs y Android con funcionalidad reducida.
- *Software* de código abierto y libre.
- Soporta diversidad de formatos como: PLY, STL, OFF, OBJ, 3DS, VRML 2.0, U3D, X3D, COLLADA e incluso nubes de puntos.

MeshLab ofrece un interfaz completo e intuitivo en el que se pueden procesar mallas en 3D con multitud de herramientas específicas para la manipulación tanto física como visual, así como para obtener información detallada de todas las propiedades de las mallas.

En este proyecto se ha utilizado este software para obtener las matrices de inercia de las diferentes mallas por las que está compuesto el humanoide Nao, previamente divididas y exportadas al formato COLLADA con Blender.

3.6. JdeRobot y ICE

La plataforma robótica utilizada en el proyecto es *JdeRobot*[9]. Es una plataforma de código libre para aplicaciones en robótica, domótica y visión artificial. JdeRobot es una plataforma orientada a componentes, esto es, en elementos independientes que funcionan en paralelo como procesos individuales. Estos componentes interactúan con sensores y actuadores gracias a interfaces de comunicación ICE. Esta plataforma sirve para programar la inteligencia de los robots. JdeRobot tiene licencia GNU GPLv3.

[8] `http://meshlab.sourceforge.net/`
[9] `http://jderobot.org`

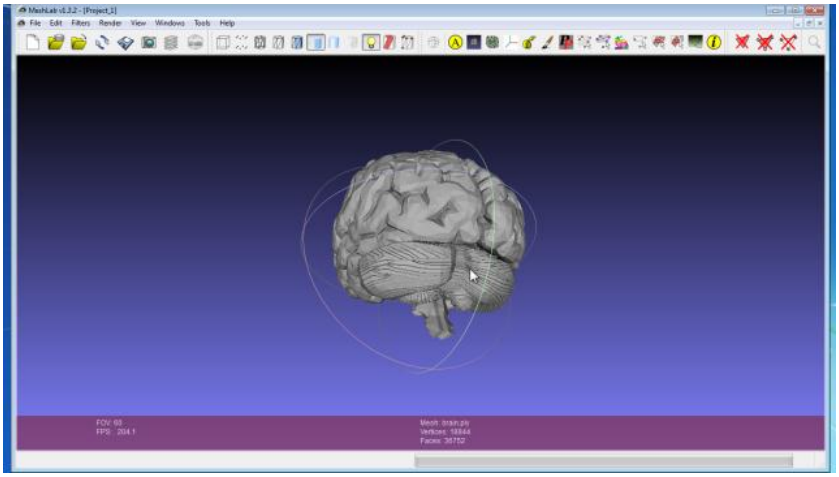


Figura 3.8: *Software* de procesado de mallas en 3D, MeshLab.

La arquitectura de JdeRobot está dividida en tres partes bien diferenciadas: la de los **componentes**, que son estos elementos independientes que se acaban de comentar; la de **interfaces**, que son los interfaces de comunicación que utilizan los componentes de JdeRobot para que interoperen entre sí; y la de las **bibliotecas** desarrolladas para el uso de componentes en diferentes áreas y que facilitan la programación de funcionalidad robótica.

Dentro de los componentes existen diferentes tipos: los *drivers*, que son aquellos componentes que actúan como servidores de datos ofreciendo información (como por ejemplo `cameraserver`, que ofrece acceso a las imágenes ofrecidas por una cámara); y las *herramientas*, que son todos aquellos componentes que interactúan entre sí y/o con los *drivers* para ofrecer una funcionalidad más completa (por ejemplo, `cameraview` accede a las imágenes servidas por `cameraserver` y las muestra en un pequeño cuadro informativo).

Para la intercomunicación entre componentes de JdeRobot, este *software* hace uso de un *middleware* de comunicaciones llamado *ICE*[10], que provee a JdeRobot de una capa de abstracción sobre la red y sus conexiones. Así, la comunicación entre los componentes la provee ICE a través de los interfaces definidos en el lenguaje *SLICE* (*Specification Language for ICE*), que ayuda a definir la manera de comunicación y parámetros usados entre componentes. Una vez definido el interfaz de comunicación, este se puede compilar para varios lenguajes de programación como C++, Python o Java, haciendo que JdeRobot se pueda considerar multilenguaje.

Actualmente JdeRobot da soporte a varios tipos de robots y sensores para los cuales existen variedad de herramientas que se pueden combinar entre sí y ofrecernos información

[10] `http://www.zeroc.com`

de todo tipo. Dispone de varios visualizadores de imágenes tanto de color como de profundidad, visores en 3D que ofrecen información de la posicion de un robot en el mundo, servidores de imágenes tanto de color como de profundidad, componenetes para la teleoperación de los robots a los que se dan soporte entre muchos otros.

3.6.1. Soporte del Nao en JdeRobot

El soporte para el Nao en JdeRobot incluye tanto herramientas como controladores para el robot Nao real y simulado, así como diferentes interfaces de comunicación para conectarse a cada uno de los sensores y actuadores del humanoide. Desde el comienzo del soporte para el Nao en JdeRobot a cargo de Jorge Bermejo [Bermejo, 2010], se han ido evolucionando estas herramientas hasta lo que son actualmente.

El robot cuenta con un *plugin* específico para cada sensor y actuador que lo compone. De tal forma que se puede controlar el robot completamente a través de componentes externos tanto al simulador, como al robot real implementados en JdeRobot. Los *plugins* que controlan al Nao simulado actualmente son:

- ***PoseNeck***, plugin encargado de controlar el cuello.
- ***Pose[Right,Left]Shoulder***: cada uno encargado de controlar un hombro del robot (*Right* para el hombro derecho y *Left* para el izquierdo).
- ***Pose[Right,Left]Elbow*** cada uno encargado de controlar un codo.
- ***Pose[Right,Left]Hip***: cada uno encargado de controlar una cadera.
- ***Pose[Right,Left]Knee***: cada uno encargado de controlar una rodilla.
- ***Pose[Right,Left]Ankle***: cada uno encargado de controlar un tobilo.

Todos estos *plugins*, ofrecen interfaces de comunicación ICE, que permiten que los diversos componentes de JdeRobot puedan acceder a ellos con una IP y un puerto específico (al que se haya atado el *plugin* previamente). Los insterfaces más importantes que ofrecen los plugins destinados al movimiento son básicamente cuatro:

- ***Pose3DEncoders***: este interfaz ofrece información de posición (x, y, z), orientación (**pan**, *tilt* y *roll* y velocidad (*panSpeed*, *tiltSpeed* y *rollSpeed*) de las articulaciones

- ***Pose3DMotors***: este interfaz ofrece la posibilidad de comandar acciones sobre posición (*x*, *y*, *z*), orientación (*pan*, *tilt*, *roll*) y velocidad (*panSpeed*, *tiltSpeed*, *rollSpeed*) a las articulaciones. El soporte actual solo hace uso del control sobre la posición (que se transforma en control en velocidad dentro del propio plugin).

- ***WalkerData***: este interfaz permite transmitir información sobre las caminatas parametrizadas. En concreto ofrece una estructura de datos donde se almacenan los 10 parámetros que componen una caminata como las diseñadas para este proyecto.

- ***StadisticsData***: este interfaz permite transimitir información estadística entre el *plugin* controlador de caminatas y los componentes JdeRobot. La información contenida aquí es, en esencia, información de posición, recorridos y tiempos, entre otros.

Capítulo 4

Nao simulado en Gazebo 5

Una vez claros los objetivos de trabajo, los requisitos que se deben cumplir y las herramientas a nuestro alcance, dedicaremos este capítulo a cubrir de forma extensa y precisa uno de los dos objetivos principales del presente proyecto: simulación del Nao en Gazebo-5. Para ello, se dará primero una visión general del diseño elegido para acto seguido describir todos los detalles de la implementación llevada a cabo, así como los problemas que han surgido y las soluciones aportadas a dichos problmas. Este capítulo estará dividido en tres secciones principales que abordarán todo lo relacionado con el modelo del robot simulado, el *plugin* desarrollado para dar soporte a la generación de caminatas (descrita en el capitulo 5) para concluir con las herramientas desarrolladas ad-hoc para el testeo de todo lo hecho previamente.

4.1. Diseño global

Como se ha expuesto en el Capítulo 2, el objetivo general del proyecto es migrar el soporte actual del robot humanoide Nao a la nueva versión del simulador utilizado en la plataforma JdeRobot: Gazebo 5, adem ás de dotar a este robot simulado de un comportamiento; en concreto hacer que camine de forma totalmente autónoma. Actualmente, el soporte existente para el Nao en JdeRobot ofrece un modelo de robot simulado que se asemeja al Nao real y para el cual se tiene control sobre todas las articulaciones por separado. La Figura 4.1 lo ilustra perfectamente. El propósito de este proyecto es mejorar el modelo existente en todas sus facetas tanto visuales como funcionales de modo que al final del mismo tengamos un robot simulado exactamente igual al robot real en apariencia e igualmente robusto en cuanto a comportamiento pudiendo incluso caminar por sí mismo.

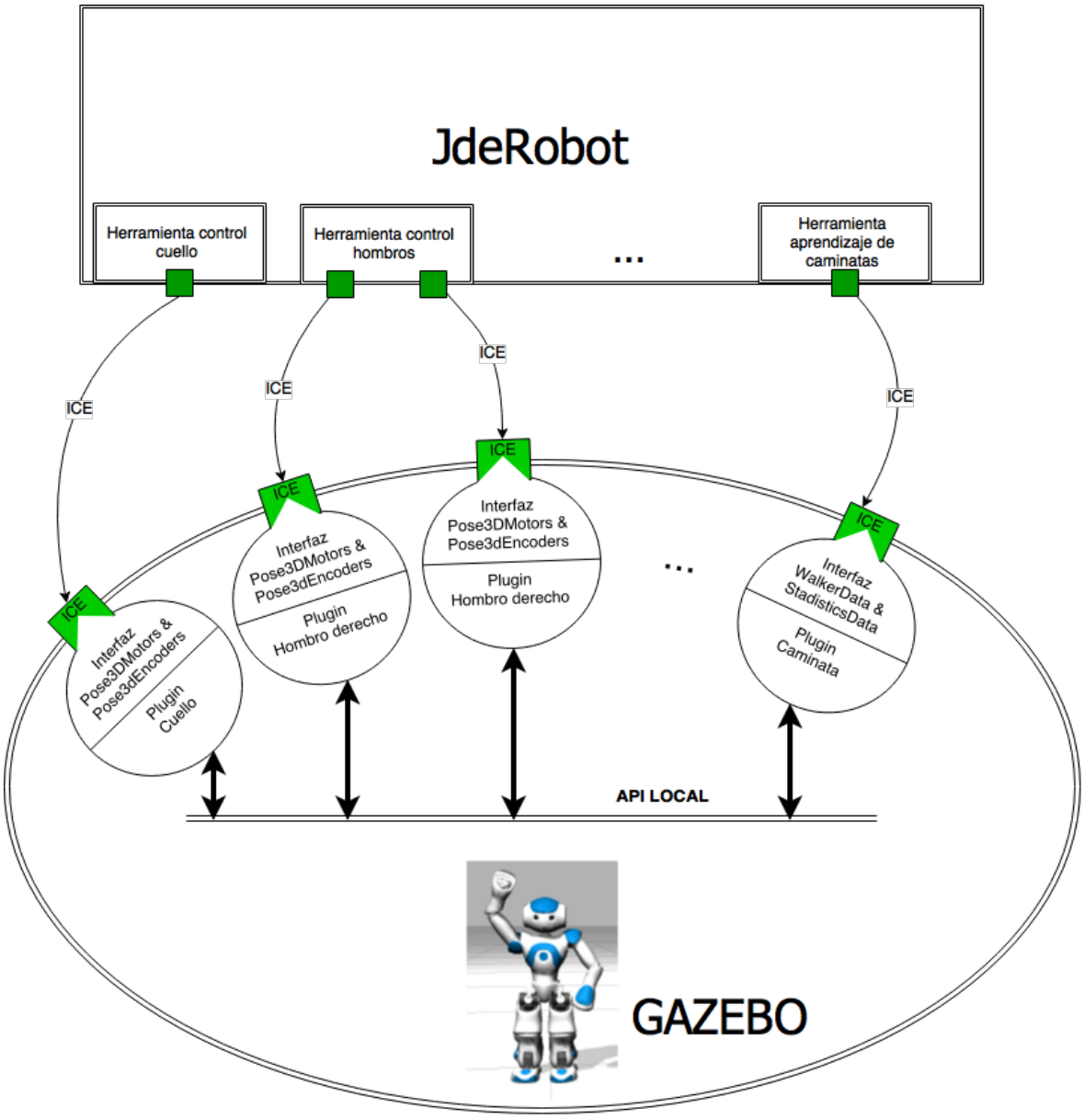


Figura 4.1: Arquitectura de JdeRobot para la simulación.

Cada sensor y actuador del robot simulado ofrece interfaces de comunicación ICE estandarizados por la plataforma JdeRobot. Gracias a estos interfaces cualquier componente JdeRobot que los tenga implementados se puede comunicar con el robot para obtener información o comandarle acciones. Estos interfaces de comunicación ofrecen toda la funcionalidad básica al robot y gracias a ellos se puede comprobar, por ejemplo, la posición de una articulación para moverla al punto deseado u obtener imágenes con las que después trabajar.

Con todos estos puntos en mente, la idea final es generar un comportamiento autónomo comandando acciones a todas las articulaciones simultáneamente de forma acompasada que nos permita realizar movimientos complejos con el robot tales como saludar, agacharse, levantarse, y el que nos ocupa en este proyecto: caminar.

Para este fin en concreto se ha desarrollado un *plugin* específico que ofrece toda la funcionalidad necesaria para que el robot consiga caminar. El soporte existente antes de la conclusión de este proyecto obligaba a conectarse a cada articulación por separado para realizar movimientos, de tal forma que había que abrir una conexión ICE para cada *plugin* al que quisiéramos conectarnos. Para hacer que el robot camine este diseño es válido pero no óptimo, ya que al saber a priori qué articulaciones están implicadas en ese movimiento, podemos crear una conexión única a un *plugin* especficamente diseñado para hacer caminar al robot, que es lo que se ha logrado con este proyecto. El *plugin* de caminatas desarrollado ofrece un único interfaz de comunicación ICE por el que solicita una serie de parámetros al componente JdeRobot para generar una caminata específica. Una vez que este *plugin* tiene calculadas las posiciones para todas las articulaciones que intervienen en el movimiento, envía acciones a cada articulación con las posiciones calculadas a través de punteros específicos a las mismas que este *plugin* posee. De esta forma, ahorramos unas 7 conexiones ICE y centralizamos la acción de caminar en un único *plugin* que contiene la lógica de este movimiento. Al interfaz que ofrece se pueden conectar los distintos componentes JdeRobot que cuya funcionalidad sea la de ordenar caminatas al robot simulado. Componentes como por ejemplo *NaoOperator*, existente en la plataforma JdeRobot y que ofrece control absoluto sobre el robot humanoide, podría hacer uso de este nuevo interfaz para añadir la funcionalidad de simular caminatas en el simulador Gazebo.

4.2. Modelo del Nao

Para lograr nuestro objetivo, hay que cumplir dos requisitos previos indispensables: tener un modelo preciso y realista del Nao; y tener los *plugins* necesarios que puedan

recoger datos y comandar acciones al robot simulado. El modelo como tal refleja todas las propiedades fśicas y visuales del robot en simulación, además de tener una referencia sobre qué *plugin* controla qué articulación. En cuanto a los *plugins*, son bibliotecas dinámicas (ficheros con formato *.so* en entornos Linux), los cuales son cargados junto con el modelo del robot al arrancar el simulador e incluir dicho modelo en el mundo. El conjunto de todos estos *plugins* se puede denominar *driver* del robot simulado, del cual hacen uso los componentes para acceder a los sensores y actuadores del robot.

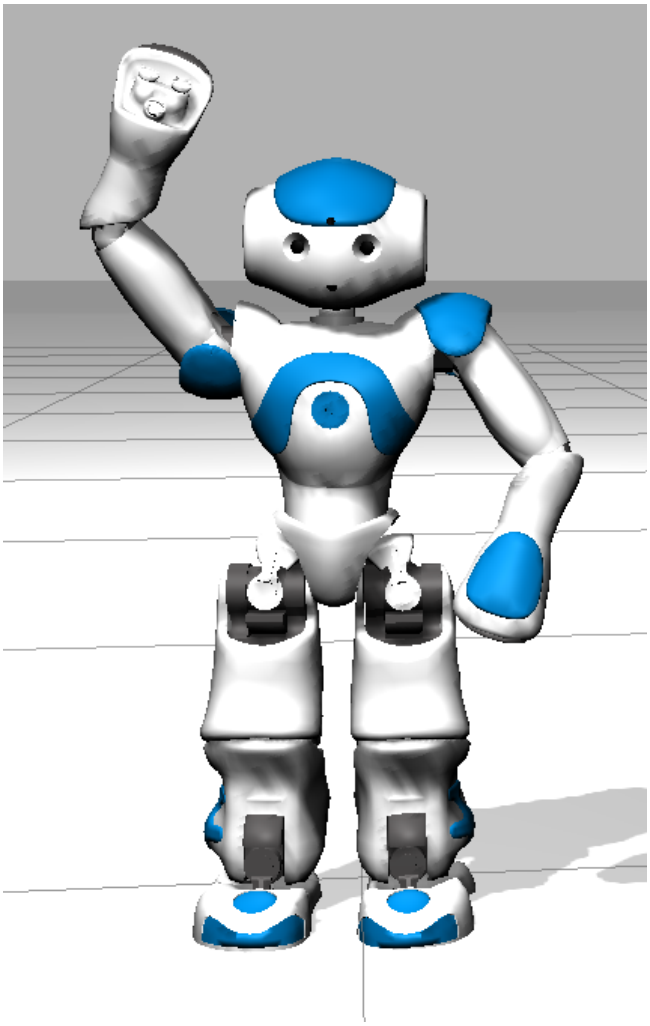


Figura 4.2: Modelo del Nao actual en Gazebo-5.

Se puede ver en la Figura 4.1 un esquema representativo de cómo está estructurado este *driver*. En ella se puede ver cómo se conecta un componente de JdeRobot al robot simulado mediante interfaces explícitos de ICE y qué interfaces ofrece cada uno de los sensores y/o actuadores del robot.

Para lograr una simulación lo más realista posible se ha accedido a la documentación

oficial del Nao en la página web de Aldebaran Robotics [1]. Hay que tener en cuenta que hay diferentes versiones del robot en función de las características técnicas y físicas del mismo. Actualmente existen tres versiones diferentes: la 3, la 4 y la 5. Para el desarrollo de este proyecto se ha simulado la versión 4 del robot, con lo cual tuvimos que tener muy presente las propiedades físicas que ofrece el fabricante, ya que entre las diferentes versiones del Nao, existen pequeñas variaciones, que, a la larga, podían causar deficiencias en nuestra simulación. Para poner de manifiesto este hecho, la Tabla 4.1 muestra la característica que se ha implementado en el modelo, la masa del cuerpo, para el torso, que es la parte del robot que más pesa, de cada una de las versiones del robot:

Versión	Masa (kg)
3.2	1.02628
3.3	1.03948
4.0	1.04956

Tabla 4.1: Masa del torso del Nao en sus diferentes versiones.

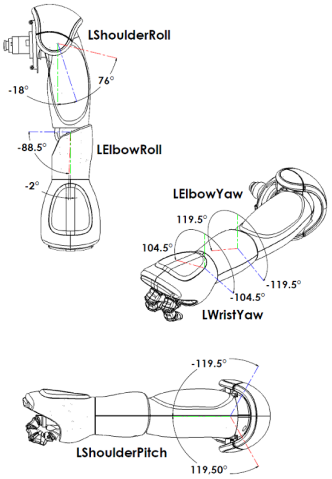
La elección de desarrollar el proyecto sobre la versión 4.0 del robot se debe a la cantidad de información que Aldebaran ofrece sobre este modelo, además de ser el más estable al inicio de este proyecto. La página web del fabricante está repleta de información de todo tipo acerca de todos los datos técnicos del robot, como puede verse en la Figura 4.3.

El anterior modelo tenía unas determinadas restricciones debido a la forma en la que se diseñó y a la carencia de mallas oficiales en aquel momento, por lo que las restricciones existentes, citadas textualmente del proyecto de Borja Menéndez, fueron:

- *El centro de masas de cada cuerpo no se ha modelado.* Esto es así debido a que se vio que algunos de ellos no parecían encajar de manera correcta con el modelo recreado. Gazebo pone por defecto el centro de masas en el centro físico del cuerpo y ese es el que finalmente modela su centro de masas.

- *La matriz de inercia de cada cuerpo tampoco se ha modelado.* La razón de ello viene derivada no sólo del anterior punto, puesto que depende directamente del centro de masas del cuerpo, sino también por su dependencia con la geometría del cuerpo en sí. Al no saber cómo ha calculado Aldebaran Robotics la matriz de inercia de los cuerpos, poner en el modelo los resultados que proporciona la empresa causa movimientos

[1] `https://www.aldebaran.com`



Joint name	Motion	Range (degrees)	Range (radians)
LShoulderPitch	Left shoulder joint front and back (Y)	-119.5 to 119.5	-2.0857 to 2.0857
LShoulderRoll	Left shoulder joint right and left (Z)	-18 to 76	-0.3142 to 1.3265
LElbowYaw	Left shoulder joint twist (X)	-119.5 to 119.5	-2.0857 to 2.0857
LElbowRoll	Left elbow joint (Z)	-88.5 to -2	-1.5446 to -0.0349
LWristYaw	Left wrist joint (X)	-104.5 to 104.5	-1.8238 to 1.8238
LHand	Left hand	Open and Close	Open and Close

Figura 4.3: Información acerca de las articulaciones del brazo en la web de Aldebaran.

inesperados o comportamientos extraños, por lo que se deja la responsabilidad de poner dicha matriz a Gazebo. El simulador pone por defecto la matriz identidad.

- *El movimiento de las muñecas y los dedos no se ha desarrollado.* Esto es debido a que dichas articulaciones, además de los dedos, ofrecen muy poca movilidad y la simulación de estos últimos no aporta gran valor a las aplicaciones actuales del Grupo de Robótica URJC con el Nao.

En el caso de este proyecto, las restricciones relacionadas con los tensores de inercia de los cuerpos y el centro de masas han sido solventadas gracias en parte a la obtención de las mallas oficiales provistas por Aldebaran. En cuanto al movimiento de articulaciones como las muñecas o las falanges, tal como apunta Borja en su proyecto, siguen siendo irrelevantes para éste, ya que no aportan un valor añadido a la caminata, más que visual.

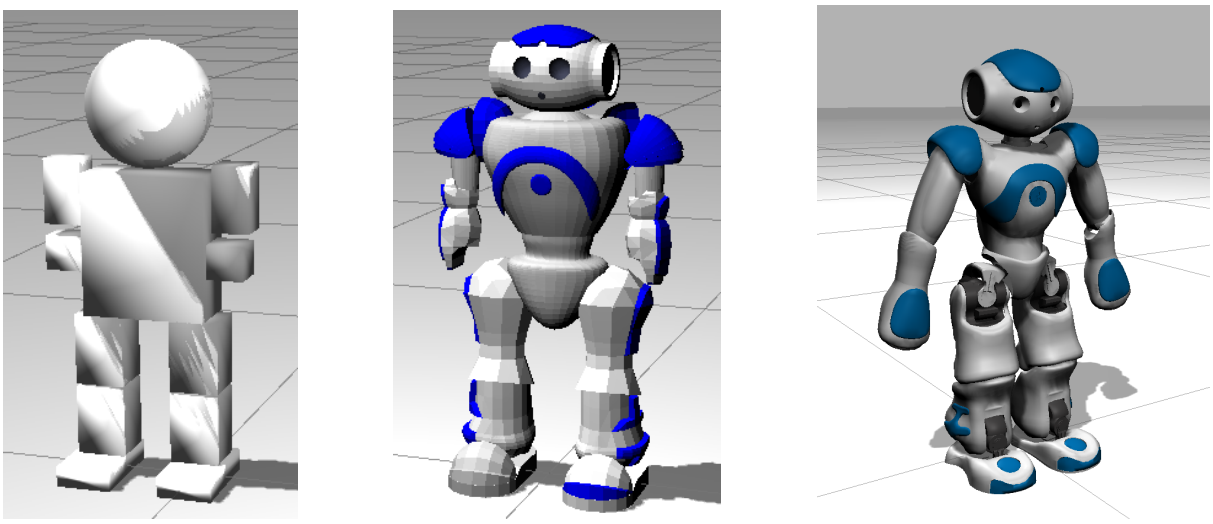
Para ponernos en el contexto de la estructura interna de un modelo robótico para la simulación en Gazebo, a continuación se explica detalladamente la estructura interna de los diferentese elementos que lo componen para introducir más adelante las propiedades físicas y visuales de esos elementos.

4.2.1. Estructura básica del modelo

El modelo de un robot para Gazebo tiene formato *SDF* (Simple Data File) al estilo *XML*, en el que se describen todos los elementos de un entorno de simulación. Dentro de

él se pueden definir multitud de características, no sólo del robot del que queramos crear un modelo, sino también del mundo que le rodea. Tal y como está estructurado Gazebo se puede crear, por un lado, el modelo del robot y, por el otro, una descripción del mundo que se quiere cargar, en la que tienen cabida diferentes modelos de robot, suelos, iluminación, etc. En esta sección se hablará únicamente del modelo del robot.

El modelo existente del Nao consta de varios cuerpos unidos entre sí mediante articulaciones. Los cuerpos se crean mediante formas geométricas básicas como son hexaedros, esferas y cilindros, entre otros. En un principio, para que el cuerpo esté creado de manera correcta hay que darle, como mínimo, una forma visual y una forma para las colisiones, pudiendo ser estas formas del cuerpo completamente diferentes entre sí. Además, hay que proporcionarle a Gazebo la posición relativa que va a tener este cuerpo con respecto a la posición del robot en el mundo, ya que por defecto estará posicionado a ras de suelo y en el centro de la estructura robótica. También se pueden dar muchas otras características como la matriz de inercia del cuerpo, la gravedad que le afecta, el tipo de material del que está hecho, etc. Finalmente, el robot queda estructurado tal y como aparece en la Figura 4.4(a) y, tras ponerle las pieles, queda como en la Figura 4.4(b). Y finalmente, como ha quedado tras la mejora realizada en este proyecto Figura 4.4(c)



(a) (b) (c)

Figura 4.4: (a) Nao simulado sin pieles, (b) Nao simulado con pieles y (c) Nao simulado mejorado.

Como se explicó en el Capítulo 2, todas las partes del cuerpo del robot tienen unas dimensiones y pesos iguales a los del robot real. Para ello hace falta especificar dichos

parámetros en el modelo. La dinámica que sigue la creación de un modelo es sencilla. Al tratarse de un formato de archivo basado en etiquetas como *SDF* resulta bastante intuitivo relacionar cada parte del cuerpo con su correspondencia en el modelo. De esta forma, a continuación se describen a modo de ejemplo algunas partes del modelo desarrollado y mejorado.

Pie

Hemos elegido describir esta parte del modelo en primer lugar porque es la que juega un papel clave a la hora de dotar de estabilidad al robot, ya que es el único punto de apoyo del robot con el suelo. El Cuadro 4.1 describe la estructura básica del modelado pie derecho del robot en formato *SDF*.

```
1   <link name='RAnkleRollLink'>
2     <!--algunas propiedades fisicas aqui -->
3     <pose>0 -0.05 -0.2879 0 -0 0</pose>
4     <inertial>
5       <!-- propiedades como la masa y las matrices de inercia van aqui -->
6     </inertial>
7     <collision name='RAnkleRollLink_collision'>
8       <pose>0 0 0 0 -0 0</pose>
9       <surface>
10        <!-- propieadades como los rozamientos entre superfies aqui -->
11      </surface>
12      <geometry>
13      <!-- mallas de colision aqui -->
14      </geometry>
15    </collision>
16    <visual name='RAnkleRollLink_visual'>
17      <pose>0 0 0 0 -0 0</pose>
18      <geometry>
19        <!-- mallas visuales aqui -->
20      </geometry>
21    </visual>
22    </link>
```

Cuadro 4.1: Modelo del pie derecho en formato SDF.

Esta estructura básica está compuesta por varias etiquetas que determinan los diferentes

elementos del modelo que utilizarán tanto Gazebo como los *plugins* que desarrollemos para nuestro robot. Más adelante se entrará más en detalle con respecto a las propiedades físicas del modelo.

- Para comenzar, tenemos la etiqueta *link* en la que se determina el nombre de la parte del cuerpo que se está modelando. Este nombre debe ser identificativo y único, ya que servirá como identificador para nuestro *plugin* a la hora de realizar acciones sobre esta parte del cuerpo.

- Justo después se define la posición relativa de esa parte del cuerpo con respecto a la posición global del robot en el mundo. Esta posición se establece con la etiqueta *pose* que tiene el siguiente formato: *x*, *y*, *z*, *roll*, *pitch* y *yaw*. Como se puede observar, estos seis valores definen tanto la posición como la orientación del cuerpo que se está modelando. Las unidades de medida para estos valores son metros y radianes para posición y orientación respectivamente.

- Acto seguido son definidos los parámetros relativos a la inercia del cuerpo utilizando la etiqueta *inertial*. Esta etiqueta alojará propiedades como un *pose* para indicar el centro de masas, una etiqueta *mass* para definir la masa del cuerpo y otra etiqueta *inertia* que describe la matriz de inercia de ese cuerpo.

- Le sigue las propiedades geométricas del cuerpo bajo la etiqueta *collision*. En esta etiqueta se define otra posición, algunas propiedades de la superficie del cuerpo como rozamientos y coeficientes varios (etiqueta *surface*), y por último su geometría, la cual puede variar desde primitivas geométricas básicas (*box*, *cylinder*, *sphere*, ...) hasta mallas complejas, como es nuestro caso, definidas con la etiqueta *mesh*.

- Para concluir, se define el apartado visual de la parte que se está modelando. Los primeros modelos desarrollados tenían como aspecto visual primitivas básicas, como puede verse en la Figura 4.4(a). Más tarde se agregaron las primeras mallas visuales al modelo, como se puede apreciar en la Figura 4.4(b) y la novedad que incluye este proyecto es la mejora de las mallas tanto visuales como de colisión para hacer el modelo más acorde con la realidad, como se puede ver en la figura 4.4(c). Esta apariencia del robot se define con la etiqueta *visual*, en la que se podrá poner una posición (con los mismos parámetros explicados anteriormente) y un mallado (con la etiqueta *mesh*, dentro de *geometry*). Este mallado está realizado con Blender[2], un

[2] `http://www.blender.org`

programa dedicado al modelado, animación y creación de gráficos tridimensionales, y tiene formato de COLLADA (.dae).

Articulación del pie

El modelado de articulaciones en Gazebo es parecido al de los cuerpos, pero teniendo diferentes parámetros de configuración. Dentro de Gazebo hay diferentes tipos de articulación:

- **Revolute**: articulación tipo bisagra que gira sobre un eje único, ya sea con un rango fijo o continuo de movimiento.
- **Revolute2**: dos articulaciones *revolute* conectadas en serie.
- **Prismatic**: articulación deslizante que se desliza a lo largo de un eje con un rango limitado especificado por los límites superior e inferior.
- **Ball**: articulación tipo rótula pero sin limitación de movimiento.
- **Universal**: como la articulación anterior, pero con restricción de un grado de libertad.
- **Piston**: parecida a una articulación *prismatic*, excepto que la rotación alrededor del eje de traslación es posible.

En el caso de articulaciones como las de un robot humanoide Nao, los tipos que se deben implementar son *revolute* o *revolute2*. Aunque *revolute2* ofrece un modelado sencillo de articulación con dos grados de libertad, todavía no está plenamente soportado en el API de Gazebo, por lo que no ha sido posible integrar este tipo de articulaciones al robot Nao. Tal como está modelado el Nao actualmente, para conseguir esos grados de libertad faltante hay que dividir el modelo por partes, de tal forma que lo que podría ser la composición formada por el pie, tobillo (como articulación) y tibia, hay que desglosarlo de la siguiente forma: dado que el pie tiene dos grados de libertad *pitch* y *roll*, hay que generar dos enlaces o *links*, uno para cada grado de libertad; de esta forma lo que podrían ser dos elementos (pie y tibia) más una articulación (tobillo) se convierte en: (1) *roll* del pie, (2) *pitch* del pie, (3) articulación que une el *roll* y el *pitch* del pie de tipo *revolute*, (4) tibia y (5) articulación que une el *pitch* del pie con la tibia de tipo *revolute* también. Este hecho aumenta considerablemente el tamaño del modelo, lo que lo hace engorroso de refactorizar. No obstante, esto supone un mayor control del modelo al completo, al tener dividido cada elemento en una etiqueta independiente.

Una vez visto cómo se modelan los cuerpos que componen el robot, se define la estructura básica de una articulación o *joint* en el Cuadro 4.2.

```
<joint name='RAnkleRoll' type='revolute'>
    <pose>0 0 0 0 0 0</pose>
    <child>RAnkleRollLink</child>
    <parent>RAnklePitchLink</parent>
    <axis>
        <xyz>1 0 0</xyz>
        <limit>
            <lower>−0.768992</lower>
            <upper>0.397935</upper>
            <effort>3.2</effort>
            <velocity>4.16174</velocity>
        </limit>
        <dynamics>
            <damping>0</damping>
        </dynamics>
    </axis>
</joint>
```

Cuadro 4.2: Modelo de una parte del tobillo derecho en formato SDF.

En el caso de las articulaciones pasa algo similar a lo que pasa con los cuerpos. Se define el tipo de elemento con la etiqueta *joint*, que en este caso será una articulación seguido de un nombre identificativo y único (*name*) y del tipo de articulación de la que se trata (*type*). Acto seguido se especifica la posición relativa de dicha articulación (etiqueta *pose*), seguido por dos elementos importantes que definen las dependencias de la articulación en el robot: *parent* y *child*. Estas dos etiquetas indican la dependencia existente entre los cuerpos modelados y cómo la articulación las conecta. En el caso que nos ocupa, se puede apreciar que el elemento padre es el superior (*AnklePitchLink*) y el elemento hijo es el inferior (*AnkleRollLink*), de tal forma que los movimientos aplicados al padre durante una simulación afectarán a todos sus hijos. En el ejemplo que estamos tratando, si se comanda a la tibia que se levante del suelo, ésta arrastrará al pie (su elemento hijo) consigo, dado que así se especifica en la articulación que las conecta. Justo después está la etiqueta *axis*, la cual incluye la definición del eje de giro de la articulación (etiqueta *xyz*) además de los límites de posición de la misma (etiquetas *upper* y *lower*) y dos propiedades físicas que se comentarán más adelante. Por último la etiqueta *dynamics* ofrece la posibilidad de

aumentar la información del modelo con más parámetros físicos.

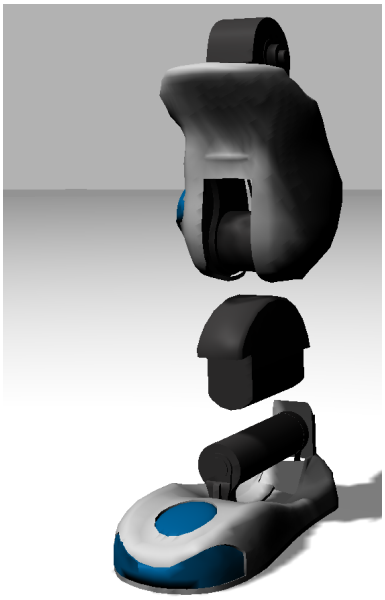


Figura 4.5: Conformación del modelo de la pierna derecha con todas sus partes en Gazebo 5.

En el Cuadro 4.3 se puede ver la otra conexión de tipo articulación que hay entre el segundo cuerpo que compone el pie (*AnklePitchLink*) y la tibia (*KneePitchLink*).

El ejemplo que se ha seguido hasta ahora es extrapolable al resto de cuerpos y articulaciones del modelo del robot completo. Hay que tener en cuenta que el número de grados de libertad puede ser distinto de dos. Por ejemplo, en el caso de la cadera el número de grados de libertad es de tres, por lo que en lugar de añadir un solo cuerpo entre el torso y el muslo, se añaden dos. En cambio, en el caso de la rodilla, que sólo cuenta con un grado de libertad, no se añaden cuerpos intermedios, simplemente en la articulación se une el muslo con la tibia.

4.2.2. Propiedades del modelo

En la sección anterior se ha descrito de forma general la estructura básica de un archivo *SDF*, que describe a los robots simulados. En esta sección estudiaremos más en profunidad las propiedades físicas del modelo que estamos tratando y cómo dichas propiedades afectan a la simulación.

```
<joint name='RAnklePitch' type='revolute'>
    <pose>0 0 0 0 0 0</pose>
    <child>RAnklePitchLink</child>
    <parent>RKneePitchLink</parent>
    <axis>
        <xyz>0 1 0</xyz>
        <limit>
            <lower>-1.1863</lower>
            <upper>0.932006</upper>
            <effort>3.2</effort>
            <velocity>6.40239</velocity>
        </limit>
        <dynamics>
            <damping>0</damping>
        </dynamics>
    </axis>
</joint>
```

Cuadro 4.3: Modelo de articulación del tobillo con la tibia en formato SDF.

Inercias

En el Cuadro 4.1, se puede observar la etiqueta *inertial* al principio de la definición. Esta etiqueta, como ya se ha apuntado, define las propiedades físicas relacionadas con las fuerzas que afectan al robot en la simulación. Para comprender esto es necesaria la definición de tensor o matriz de inercia: *Al calcular las ecuaciones del movimiento del sólido rígido, la distribución de masas se traduce en seis números que aparecen como una matriz simétrica; según el modelo de sólido que se use.* Esta matriz se conoce como tensor de inercia y en términos generales determina el comportamiento de los cuerpos sólidos rígidos en movimiento. Este tensor de inercia depende directamente de la masa y la distribución del objeto para el que se calcula. En el modelo del Nao realizado en este proyecto, y, siguiendo el ejemplo del pie, tenemos el Cuadro 4.4 en el que podemos apreciar las diferentes etiquetas que definen las propiedades inerciales del cuerpo.

En primer lugar tenemos una etiqueta *pose* que determina el centro de masas del cuerpo. Este factor es muy importante tenerlo bien definido, ya que de él depende la distribución de masas del modelo en cuestión.

Seguidamente tenemos la etiqueta *mass*, la cual determina la masa del cuerpo. Como

acabamos de apuntar, este es un factor crítico, al igual que el resto, ya que lo que se está calculando es la distribución de la masa del cuerpo, por lo que si los valores no son coherentes con el modelo geométrico del robot, la simulación no será realista.

Por último, pero no menos importante, se defie el tensor de inercia como tal. Al tratarse de una matriz simétrica basta con especificar los valores para uno de los lados de la misma para que quede totalmente definida. Tenemos que tener muy bien definidos los valores para esta matriz, ya que de ellos depende en gran medida el realismo a la hora de poner nuestro modelo en movimiento. Afortunadamente los valores para los tensores de inercia de todos los cuerpos que componen el modelo los provee Aldebaran, por lo que no hay que calcularlos por nuestra cuenta. Existe, no obstante, software específico para calcular diversidad de propiedades físicas de cuerpos sólidos tales como *MeshLab*, del cual se ha hecho uso en este proyecto debido a problemas surgidos por la incompatibilidad de las mallas usadas al principio y los valores de inercia ofrecidos por el fabricante.

Algunos de los problemas encontrados con la definición de las inercias fue que el robot, estando parado en el simulador, sin aplicarle ninguna fuerza externa ni interna, salía despedido por los aires sin motivo aparente. Ajustando manualmente los tensores de inercia uno a uno para que fueran coherentes con el modelo usado en aquel momento, conseguimos erradicar aquel comportamiento. Sin embargo, conseguir la estabilidad del Nao en estático fue prácticamente imposible. Esto se solucionó utilizando los valores oficiales ofrecidos por Aldebaran y cambiando las mallas a unas más actualizadas y compatibles con dichos valores. El esfuerzo que supuso conseguir estabilidad en el Nao fue considerable y ha sido la tarea que más tiempo ha copado en el desarrollo. Otro problema que surgió, debido a la misma causa, fue la rotación de las mallas de colisión. Dado que las mallas no estaban bien definidas, el robot atravesaba el suelo del mundo simulado con algunas partes del cuerpo (incluídas algunas zonas de los pies), por lo que una de las soluciones propuestas fue girar manualmente las mallas para hacerlas coincidir con el modelo, lo cual solucionó el problema hasta la llegada de las nuevas mallas.

Rozamientos

Otro de los factores físicos, no tan críticos para el movimiento de articulaciones por separado, es el rozamiento. En el Cuadro 4.5 se detallan las etiquetas que componen todos los parámetros de inercia del modelo. Estos se verán de forma somera al no considerarse tan importantes como los anteriores.

En primer lugar tenemos la etiqueta *surface*, la cual dispone de algunas subetiquetas

```
<link name='RAnkleRollLink'>
    <self_collide>false</self_collide>
    <gravity>true</gravity>
    <pose>0 −0.05 −0.2879 0 −0 0</pose>
    <inertial>
        <pose>0.02542 −0.0033 −0.03239 0 −0 0</pose>
        <mass>0.16184</mass>
        <inertia>
            <ixx>0.000269302</ixx>
            <ixy>5.87505e−06</ixy>
            <ixz>0.000139133</ixz>
            <iyy>0.000643474</iyy>
            <iyz>−1.88492e−05</iyz>
            <izz>0.000525035</izz>
        </inertia>
    </inertial>
    ...
</link>
```

Cuadro 4.4: Modelo de tensor de inercia del pie en formato SDF.

de las que sólo nos interesa *contact* para el modelo actual. Dado que utilizamos el motor de físicas ODE que provee Gazebo para la simulación de comportamientos de los objetos, utilizamos la etiqueta *ode*, para indicar este hecho. (Si se quisiera utilizar el otro motor de físicas del que dispone gazebo: BULLET, sólo habría que cambiar el nombre de la etiqueta por *bullet*). ODE nos ofrece algunos parámetros para la correcta simulación de contactos en el modelo:

- **kp** y **kd**: definen la solidez del modelo.
- **max vel**: velocidad máxima de corrección.
- **min depth**: profundidad mínima permitida para el contacto antes de que se envíe un impulso de corrección.

Por otra parte disponemos de los siguientes valores que definen la fricción del modelo bajo la etiqueta *friction*:

- **mu** y **mu2**: primer y segundo coeficiente de fricción del modelo.
- **fdir1**: 3-tupla que indica la dirección en la que se aplica la fuerza de rozamiento *mu1*.

Ajustando estos valores se puede lograr un gran realismo en la simulación, siempre y cuando éstos sean coherentes con el modelo que se está desarrollando.

4.2.3. Apariencia del modelo

Es importante conseguir una simulación realista a partir de las propiedades físicas del modelo robótico, dado que sin esto, no podríamos desarrollar *software* que exportar a los robots reales que tratamos de programar. Otra parte importante de la simulación es el aspecto visual, ya que es en esto, en la observación del comportamiento del robot en lo que vamos a basar nuestras concluisones. Por ello, tener un modelo con un aspecto visual muy cercano a la realidad, nos permitirá estudiar con más detalle el comportamiento de nuestros robots en simulación.

Cuando se desarrolló el primer Nao para JdeRobot, este estaba compuesto por primitivas geométricas tales como hexaedros, cilindros y esferas. Como se puede apreciar en la Figura 4.4(a). Poco después se empezaron a aplicar mallas visuales al modelo, y fruto del trabajo de Borja Menéndez surgió el modelo con el que comenzamos este proyecto:

```
<link name='RAnkleRollLink'>
    ...
    <collision name='RAnkleRollLink_collision'>
            <pose>0 0 0 0 −0 0</pose>
            <surface>
                <contact>
                    <ode>
                        <kp>1000000.0</kp>
                        <kd>100.0</kd>
                        <max_vel>0.1</max_vel>
                        <min_depth>0.003</min_depth>
                    </ode>
                </contact>
                <friction>
                    <ode>
                        <mu>0.5</mu>
                        <mu2>0.5</mu2>
                        <fdir1>1 0 0</fdir1>
                    </ode>
                </friction>
            </surface>
            ...
</link>
```

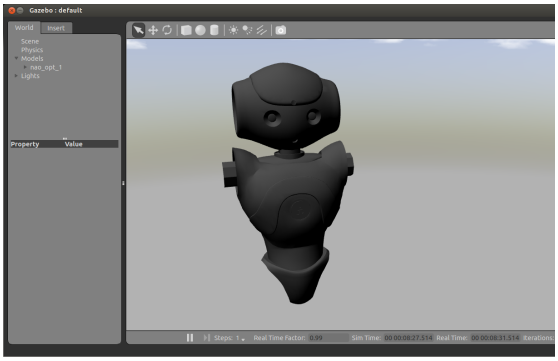
Cuadro 4.5: Modelo de fricción del pie en formato SDF.

Figura 4.4(b). Sin embargo, puede apreciarse que este modelo no concuerda exactamente con el robot real (Figura 4.6), por lo que uno de los objetivos que se propuso a la hora de llevar a cabo este proyecto fue conseguir que el modelo se acercara lo más posible a la realidad, como se puede observar en la Figura 4.7.

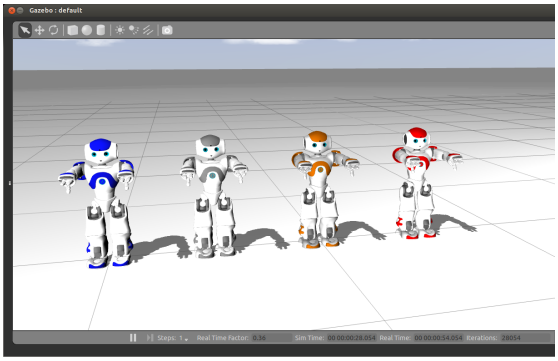
Figura 4.6: Nao real.

La consecución de este aspecto visual no fue sencilla al principio debido a la falta de información para conseguir las mallas oficiales, por lo que tuvimos que coger un modelo completo del Nao y dividir las diferentes partes del cuerpo con el programa de edición 3D *Blender*. En la figura 4.7(a) puede verse una de las etapas del proceso de obtención del modelo de forma manual. Finalmente conseguimos obtener un resultado bastante realista en cuanto a apariencia habiendo realizado un coloreado manual de varios modelos Figura 4.7(b), pero nada realista en cuanto a funcionalidad, debido a la disparidad de tamaños de mallas que obtuvimos, que no tenían correspondencia con los valores físicos aplicados al modelo. No fue hasta que se consiguieron las mallas oficiales con los tamaños apropiados, que no se pudo realizar una simulación precisa y que permitiera estabilidad al modelo. A pesar de que los colores no son tan vivos como los modelos coloreados a mano, el resultado visual es bastante aceptable: Figura 4.7(c).

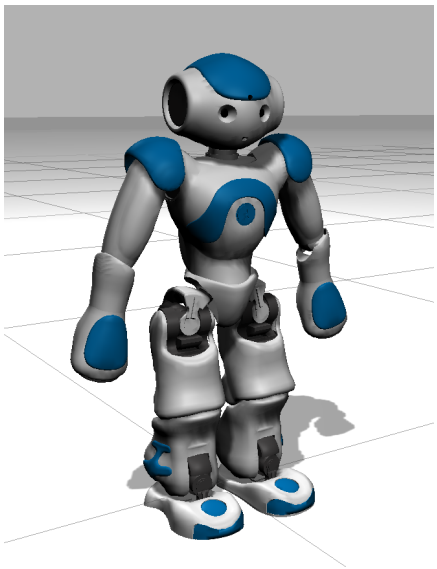
Actualmente, es muy sencillo dotar de color al modelo, ya que junto a la especificación del mismo y de sus mallas, hay que incluir un archivo de textura con 4 colores que se deseen, asociados a las mallas visuales del robot.



(a)



(b)



(c)

Figura 4.7: (a) Parte del Nao obtenido manualmente, (b) Varios Nao's coloreados con blender y (c) Nao actual.

4.3. *Plugin* básico

Una vez el modelo ha sido creado, los desarrolladores pueden programar aplicaciones que extraigan información de los sensores y comanden movimientos a sus actuadores. El simulador ofrece una API con diferentes módulos que tienen métodos para tener una simulación realista, ofreciendo acceso software a los modelos simulados.

El soporte existente del Nao incluía un *plugin* que controlaba cada articulación y cada sensor del robot simulado. Estos *plugins* se ejecutan en paralelo al arrancar Gazebo con el modelo permitiendo la comanda de movimientos simultáneos a cada articulación para generar movimientos complejos, como el caminar. Gracias a este conjunto de *plugins*, un componente de JdeRobot se podrá comunicar con el robot simulado.

Cada uno de estos *plugins* consta de dos partes diferenciadas: la parte de Gazebo, que se conecta con el API de bajo nivel que ofrece el simulador para poder obtener datos sobre el robot y hacerle actuar; y la parte de la interfaz de comunicaciones ICE, que ofrece un API de alto nivel para que cualquier componente de JdeRobot pueda comunicarse con él; y es en esencia en los dos subapartados que se va a dividir esta sección. Todo en su conjunto es el "*driver* JdeRobot" para el Nao simulado (Figura 4.1).

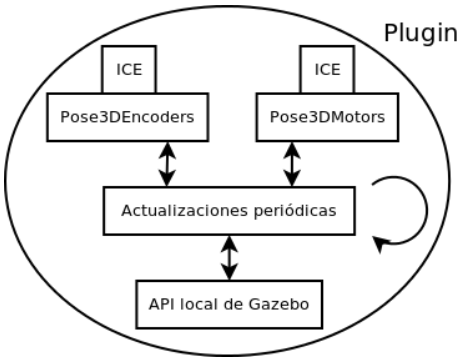


Figura 4.8: Esquema de actuación de los *plugins* de las articulaciones.

En la Figura 4.8 se puede ver un esquema de cómo actúan los *plugins*. Por un lado, son capaces de recibir peticiones asíncronas a través de sus interfaces de comunicación ICE. Tras recibirlas, el método encargado de actualizar la posición de la articulación realiza operaciones con el API de bajo nivel que proporciona Gazebo para mover adecuadamente dicha articulación.

Cada *plugin* trabaja de forma independiente a los demás al tratarse de hilos de ejecución diferenciados. Además, cada *plugin* tiene a su vez un hilo propio que permite escuchar

peticiones de los componentes de JdeRobot gracias a sus interfaces de comunicación ICE, dedicadas para cada uno de los *plugins* existentes.

Los *plugins* desarrollados para dar soporte al Nao tienen la misma estructura interna, ya que se componen de las mismas estructuras de datos (ajustándolas a cada articulación en concreto) y de las mismas funciones, al tratarse todos ellos de controladores para los mismos tipos de elementos: las articulaciones. Por lo tanto, exceptuando aquellos *plugins* dedicados a los sensores (ya que estos sólo reciben información, no comandan movimientos ni acciones), los *plugins* que nos interesan son los encargados de la locomoción del robot, por lo que en este capítulo obviaremos los *plugins* dedicados a los sensores (cámaras, sensor de ultrasonidos, sensor infrarrojo, etc.).

En esencia un *plugin* está compuesto por una estructura de datos que almacena variables claves para el envío de órdenes a los actuadores. En este caso, los tipos de datos almacenados son las propias articulaciones (de tipo *JointPtr*, estructuras que almacenan los datos que se transmiten por las interfaces ICE (como el *Pose3DEncoders* o el *Pose3DMotors*), entre otras variables necesarias para la implementación del *plugin* en cuestión. Las variables que identifican las articulaciones, son variables del API de bajo nivel de Gazebo y se utilizan para obtener la posición de los cuerpos que unen dichas articulaciones y para dotarles de movimiento a través de instrucciones específicas ofrecidas por el API del simulador. Como ejemplo, en el Cuadro 4.6 se muestra la estructura de datos principal del *plugin* del pie derecho.

```
1 struct rightankle_t {
2     physics::JointPtr joint_pitch, joint_roll;
3     encoders_t encoders;
4     motorsdata_t motorsdata;
5     motorsparams_t motorsparams;
6 };
```

Cuadro 4.6: Estructura de datos del pie derecho.

Cabe destacar que el acceso a estas estructuras de datos se hace de forma segura, utilizando el mecanismo de exclusión mútua o *mutex*. Dado el carácter desacoplado de los encoders y de los motores, se decidió en su momento utilizar un *mutex* independiente para cada uno de ellos.

Como novedad de este proyecto, se ha incluído un controlador PID en cada uno de los *plugins* existentes para traducir las posiciones deseadas de las articulaciones específicas,

en comandas de velocidad para dichas articulaciones, que hemos considerado que dotan de más realismo a la simulación.

Como dato importante y para justificar la presencia de un controlador PID, hemos mencionado que el control de todas las articulaciones se realiza en velocidad. Esta decisión de diseño viene por la experiencia no del todo realista del control en posición que ofrece Gazebo para nuestro modelo ya definido. Se observó que el control en posición generaba comportamientos extraños en el robot a la hora de realizar movimientos, como que el robot pivotaba en el aire sin razón aparente, o que las articulaciones no adquirían la posición exacta que se les comandaba. Por todo ello, se decidió mantener el control en velocidad que existía, añadiendo un controlador PID para compensar el tiempo que tardaba la articulación en llegar al destino marcado en posición. A grandes rasgos, convertimos una posición deseada para una articulación en una instrucción de velocidad en Gazebo compensando el consumo de tiempo que conlleva un control basado en velocidad.

4.3.1. API de bajo nivel con Gazebo

Gazebo tiene un motor de ejecución que actualiza periódicamente la información de los cuerpos que tiene en ese momento en simulación, de forma que las variables internas que llevan el control de todas las propiedades de los cuerpos se actualizan continuamente.

La parte programática de los *plugins* más básica de Gazebo pasa por la sobreescitura de tres métodos principales que se detallan a continuación. Para situar el contexto de bajo nivel en el que nos hemos movido, se van a ilustrar estos métodos detallando el *plugin* desarrollado para el pie derecho con la refactorización incluyendo el controlador PID. La Figura 4.9 resume la ejecución de cada *plugin*: primero el *plugin* llama a ciertos métodos para inicializar todo lo necesario y luego el método de actualización va modificando los valores de los cuerpos en función de las peticiones ICE que reciba.

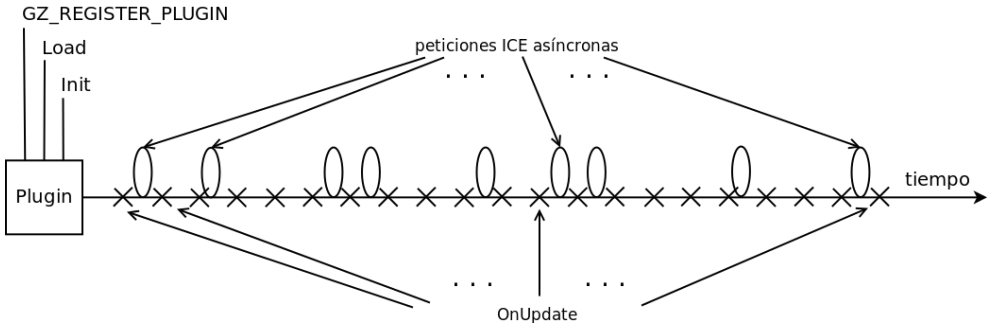


Figura 4.9: Diagrama de ejecución de los métodos más importantes en cada *plugin*.

1. `GZ_REGISTER_PLUGIN`. Esta macro predefinida por Gazebo especifica qué *plugin* se quiere registrar. Para ello necesita el nombre de la clase que va a implementar el *plugin*.

2. `Load`. Este método es la puerta de acceso a los atributos del modelo definido anteriormente. Recibe como parámetros: el modelo sobre el que se hace la llamada del *plugin* y el fichero descriptor del robot (*SDF*) que contiene la información necesaria para el funcionamiento del controlador. Los valores que nos interesan del modelo para el ejemplo que estamos tratando son: el *pitch* y el *roll* del pie derecho. El Cuadro 4.7 muestra la estructura del método *Load* para el pie derecho. Además de obtener los anteriores valores, este método se encarga de lanzar el hilo de ejecución de las interfaces ICE de esta articulación. Una vez lanzado este hilo de ejecución se lanza también el método *OnUpdate*.

3. `Init`. Este método es invocado antes de comenzar el *mainloop* de control de la simulación del que se encarga la función *OnUpdate*. Hace las veces de una función de inicialización de variables para mantener en todo momento el control sobre los valores que se le asignan a las articulaciones. Para nuestro ejemplo, utilizamos el método *Init* para inicializar a cero los valores tanto del *pitch* como del *roll* del pie derecho; además de los valores de error acumulado necesarios para el controlador PID, como se puede apreciar en el Cuadro 4.8.

4. `OnUpdate`. Gazebo ejecuta este método iterativamente hasta que se elimine al robot de la simulación o el simulador se cierre, después de haber sido llamado con el método `ConnectWorldUpdateBegin` desde `Load`.

En cada iteración se actualizan los valores del *encoder*, es decir, la posición en la que están los ejes de la articulación, con dos propósitos: por si un componente externo de JdeRobot lo pide y para calcular la distancia que hay entre la posición deseada y la posición actual. Esta actualización se realiza gracias a los métodos de la clase *JointPtr* de Gazebo, variable interna que tiene la estructura del Cuadro 4.6.

La estructura de éste método es prácticamente la misma para todos los *plugins*. Primero se obtienen los datos de posición de los cuerpos que gestiona el *plugin* en cuestión con la instrucción que ofrece el API de Gazebo: *GetAngle();*. Acto seguido se ha de especificar la fuerza máxima o par con el que la articulación se va a mover. Este es uno de los cambios principales de la refactorización de los *plugins*, ya que debido a los cambios realizados en el motor de físicas de Gazebo: ODE; la instrucción que se utilizaba hasta ahora para este fin (*SetMaxForce()*) ha quedado obsoleta. En su lugar se ha implementado esta funcionalidad con una instrucción más genérica a la que se le indica el tipo de parámetro que se quiere definir, *SetParam(key, index, value)*. Esta instrucción, en nuestro caso tomará como *key* el valor *"fmax"* para indicar a Gazebo que el parámetro que se está definiendo es la fuerza

```
1  void PoseRightAnkle::Load ( physics::ModelPtr _model, sdf::ElementPtr _sdf ) {
2          if (!_sdf->HasElement(this->modelPitch))
3              gzerr << "PoseRightAnkle plugin missing <" << this->modelPitch << "> element\n";
4          if (!_sdf->HasElement(this->modelRoll))
5              gzerr << "PoseRightAnkle plugin missing <" << this->modelRoll << "> element\n";
6
7          std::string elemPitch = std::string(_sdf->GetElement(this->modelPitch)->Get<std::string>());
8          std::string elemRoll = std::string(_sdf->GetElement(this->modelRoll)->Get<std::string>());
9
10         this->rightankle.joint_pitch = _model->GetJoint(elemPitch);
11         this->rightankle.joint_roll = _model->GetJoint(elemRoll);
12
13         this->maxPitch = (float) this->rightankle.joint_pitch->GetUpperLimit(0).Radian();
14         this->minPitch = (float) this->rightankle.joint_pitch->GetLowerLimit(0).Radian();
15         this->maxRoll = (float) this->rightankle.joint_roll->GetUpperLimit(0).Radian();
16         this->minRoll = (float) this->rightankle.joint_roll->GetLowerLimit(0).Radian();
17
18         // Load torque
19         if (_sdf->HasElement("torque"))
20             this->stiffness = _sdf->GetElement("torque")->Get<double>();
21         else {
22             gzwarn << "No torque value set for the left ankle plugin.\n";
23             this->stiffness = 5.0;
24         }
25
26         pthread_t thr_ice;
27         pthread_create(&thr_ice, NULL, &thread_RightAnkleICE, (void*) this);
28
29         // Load OnUpdate method
30         this->updateConnection = event::Events::ConnectWorldUpdateBegin(
31                                      boost::bind(&PoseRightAnkle::OnUpdate, this));
32 }
```

Cuadro 4.7: Método *Load* del *plugin* del pie derecho en Gazebo.

```
1  void PoseRightAnkle::Init () {
2          this->rightankle.encoders.tilt = 0.0;
3          this->rightankle.encoders.roll = 0.0;
4
5          this->rightankle.motorsdata.tilt = 0.0;
6          this->rightankle.motorsdata.roll = 0.0;
7
8          this->error_tilt = 0.0;
9          this->error_roll = 0.0;
10         this->error_tilt_ant = 0.0;
11         this->error_roll_ant = 0.0;
12 }
```

Cuadro 4.8: Método *Init* del *plugin* del pie derecho en Gazebo.

máxima a la que el *plugin* va a funcionar (la antigua *SetMaxForce()*). Justo después, y a partir de los datos almacenados previamente obtenidos por los encoders, se calcula la velocidad que se va a aplicar a la articulación para llegar a la posición deseada por el componente externo. Para ello, basta con aplicar el control PID a la diferencia entre la posición a la que se desea llegar y la posición actual de la articulación (obtenida de los encoders) en cada iteración. La constante proporcional la hemos puesto con un valor de 10, tras realizar y evaluar varias pruebas. Además hemos incluído un valor pequeño a la constante derivada para evitar oscilaciones en el movimiento de las articulaciones. Dado que este cálculo se realiza con una frecuencia de unos 20Hz, no ha sido necesario poner un valor excesivamente grande. Como seguridad y para dar completitud y realismo a la funcionalidad de los *plugins*, se ha incluído una comprobación de seguridad para evitar que las articulaciones excedan el límite máximo de rotación definido por el fabricante. Por último, se comanda esa velocidad calculada a la articulación mediante una instrucción específica y que también ha sido refactorizada con respecto a la versión anterior del *plugin*. En este caso concreto, la instrucción que se utilizaba era *SetVelocity*, no obstante, debido a los cambios en los motores de físicas comentados antes, esta instrucción ha quedado obsoleta y ha sido reemplazada por la misma instrucción que utilizamos para definir la fuerza máxima: *SetParam(key, index, value)*. De este modo, los *plugins* quedan totalmente operativos, actualizados y funcionales para la nueva versión de Gazebo. El cuadro 4.9 ilustra toda la funcionalidad de este método

```
1  void PoseRightAnkle::OnUpdate () {
2
3          // ----------ENCODERS----------
4          pthread_mutex_lock(&this->mutex_rightankleencoders);
5          this->rightankle.encoders.tilt = this->rightankle.joint_pitch->GetAngle(0).Radian();
6          this->rightankle.encoders.roll = this->rightankle.joint_roll->GetAngle(0).Radian();
7          pthread_mutex_unlock(&this->mutex_rightankleencoders);
8
9          // ----------MOTORS----------
10         this->rightankle.joint_pitch->SetParam("fmax",0, this->stiffness);
11         this->rightankle.joint_roll->SetParam("fmax",0, this->stiffness);
12
13         pthread_mutex_lock(&this->mutex_rightanklemotors);
14         this->error_tilt = this->rightankle.motorsdata.tilt - this->rightankle.encoders.tilt;
15         this->error_roll = this->rightankle.motorsdata.roll - this->rightankle.encoders.roll;
16
17         double pitchSpeed = 10*(this->rightankle.motorsdata.tilt - this->rightankle.encoders.tilt) +
18          0.1*(this->error_tilt - this->error_tilt_ant);
19         double rollSpeed = 10*(this->rightankle.motorsdata.roll - this->rightankle.encoders.roll) +
20          0.1*(this->error_roll - this->error_roll_ant);
21
22         // Checking joint limits, avoiding weird behaviours
23         if ((this->rightankle.motorsdata.roll >= maxRoll) ||
24             (this->rightankle.motorsdata.roll <= minRoll))
25             rollSpeed = 0.0000000;
26         if ((this->rightankle.motorsdata.tilt >= maxPitch) ||
27             (this->rightankle.motorsdata.tilt <= minPitch))
28             pitchSpeed = 0.000000;
29
30         this->rightankle.joint_pitch->SetParam("vel",0, pitchSpeed);
31         this->rightankle.joint_roll->SetParam("vel",0, rollSpeed);
32
33         this->error_tilt_ant = this->error_tilt;
34         this->error_roll_ant = this->error_roll;
35         pthread_mutex_unlock(&this->mutex_rightanklemotors);
36
37         ...
38     }
39 }
```

Cuadro 4.9: Método *OnUpdate* del *plugin* del codo derecho en Gazebo.

4.3.2. API con interfaces ICE

El API de alto nivel ofrece un interfaz de comunicaciones ICE para que otro componente JdeRobot pueda hacer uso de la simulación del robot. Este interfaz de comunicaciones, en el caso de las articulaciones, es doble, ofrece dos interfaces ICE: *Pose3DEncoders* y *Pose3DMotors*. Tanto el *Pose3DEncoders* como el *Pose3DMotors* se utilizan típicamente en cuellos mecánicos, de ahí su uso en cualquier tipo de articulación. El interfaz *Pose3DMotors* ofrece la posibilidad de comandar acciones sobre posición (x, y y z), orientación (*pan*, *tilt* y *roll*) y velocidad (*panSpeed* y *tiltSpeed*), pero para el caso de las articulaciones sólo se hace uso del control sobre la orientación.

Las clases *Pose3DEncoders* y *Pose3DMotors* quedan implementadas en cada uno de los *plugins*, uno por cada articulación del humanoide Nao, heredando de las clases ICE e implementando sus métodos. En concreto, la clase *Pose3DEncoders* implementa un método para obtener la posición de la articulación (`getPose3DEncodersData`). La clase *Pose3DMotors* implementa tres métodos: `getPose3DMotorsData`, que da información sobre la última actualización de la posición de los motores; `getPose3DMotorsParams`, que proporciona información acerca de los valores máximos y mínimos que puede tomar tanto la posición como la velocidad de la articulación; y `setPose3DMotorsData`, que recibe como parámetro un puntero a la clase *Pose3DMotorsData* y comanda el movimiento requerido a la articulación. Todos y cada uno de los métodos hacen uso, bien para ofrecer información, bien para modificar valores, del atributo que tiene asignado la estructura de datos comentada en el Cuadro 4.6, protegiendo su acceso debidamente con los mutex anteriormente comentados.

Ambas clases reciben como parámetro del constructor un puntero a la clase desarrollada por el *plugin*, de manera que se puedan obtener y modificar los valores necesarios para el movimiento de la articulación fácilmente. Una vez creados estos interfaces, se delega a ICE el compromiso de mantener dichos interfaces abiertos para que otros componentes de JdeRobot se puedan conectar a ellos asíncronamente.

Configuración.

Por último, para que se pueda ofrecer un interfaz de comunicaciones con ICE hace falta una mínima configuración. Dicha configuración se puede encontrar en cada uno de los ficheros *.cfg*, uno por cada *plugin*. En ellos se definen una serie de parámetros como qué tipo de conexión se va a realizar (TCP o UDP, por defecto será esta última), la IP en la que se ofrecerá la interfaz y el puerto al que estará atado. Se puede observar un ejemplo en el Cuadro 4.10.

```
1 PoseRightElbowEncoders.Endpoints=default -h localhost -p 9070
2 PoseRightElbowMotors.Endpoints=default -h localhost -p 9071
```

Cuadro 4.10: Fichero de configuración de la interfaz del codo derecho.

Los valores por defecto de dicha configuración siguen un convenio para todas las articulaciones, de forma que sea fácil identificarla: si acaba en 0 ofrecerá un interfaz de la clase *Pose3DEncoders* y si acaba en 1 ofrecerá un interfaz de la clase *Pose3DMotors*. En la Tabla 4.2 se puede ver cómo queda la configuración de las diferentes articulaciones:

Interfaz ICE	Puerto Pose3DEncoders	Puerto Pose3DMotors
LeftAnkle	9000	9001
RightAnkle	9010	9011
LeftKnee	9020	9021
RightKnee	9030	9031
LeftHip	9040	9041
RightHip	9050	9051
LeftElbow	9060	9061
RightElbow	9070	9071
LeftShoulder	9080	9081
RightShoulder	9090	9091
NeckSpeed	9100	9101
Neck	9110	9111

Tabla 4.2: Puertos utilizados para las diferentes interfaces de cada articulación.

4.4. *Plugin* de caminata

Además de la refactorizaciíon y actualización de todo lo anterior; y como novedad en este proyecto, se ha desarrollado un *plugin* específico de alto nivel que permite, a través de un componente JdeRobot, generar caminatas compactadas y parametrizadas para el robot humanoide Nao. En esta sección se describe tanto el *plugin* en sí mismo, como el acceso de bajo nivel a las articulaciones por separado.

4.4.1. *Plugin* de alto nivel *Walking*

A diferencia del resto de *plugins*, éste nuevo *plugin* aumenta las posibilidades de desarrollo para robots humanoides en JdeRobot, ya que sirve como base para generar otros *plugins* similares que acepten tipos de movimientos complejos, y no articulación por articulación como se venía haciendo hasta ahora en JdeRobot para el simulador Gazebo La Figura 4.10 refleja la arquitectura de este nuevo *plugin* en el entorno JdeRobot + Gazebo.

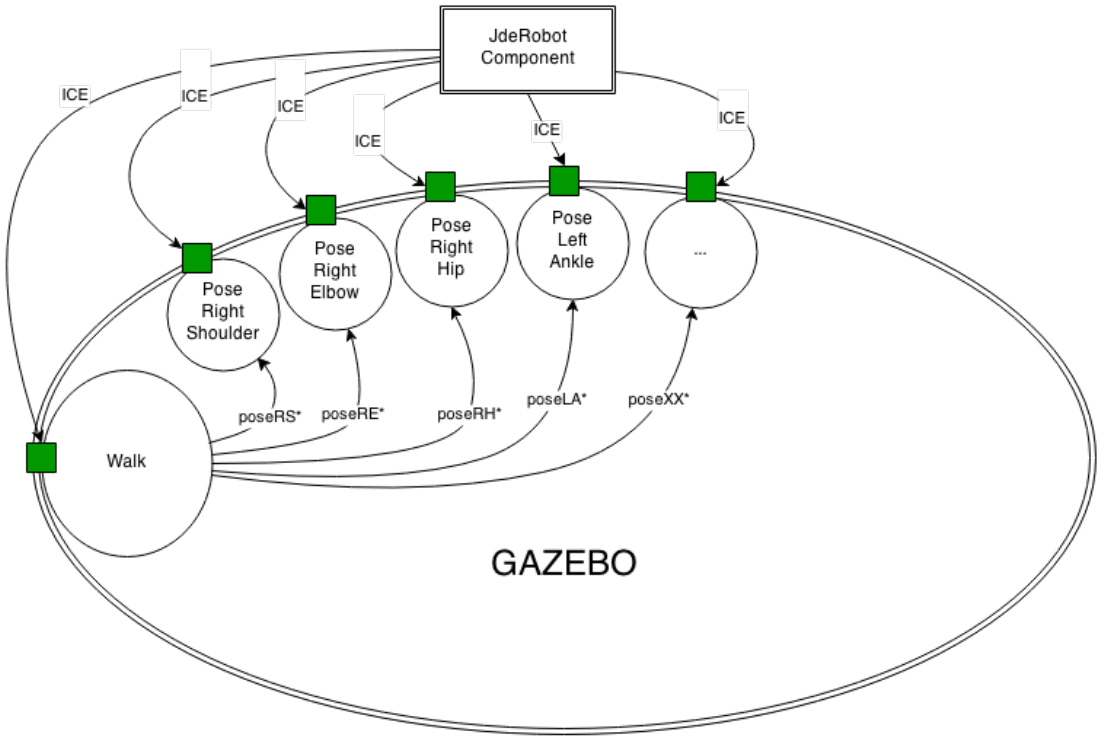


Figura 4.10: Esquema de actuación de los *plugins* de las articulaciones.

Para el desarrollo de este *plugin* nos hemos basado en una funcionalidad similar existente y realizada por Francisco Miguel Rivas en su trabajo de fin de carrera [Rivas, 2011], en el cual se modelaba una caminata para el robot humanoide Nao utilizando el simulador de código cerrado *Webots*. Para entender lo expuesto en este capítulo, vamos a describir de forma muy somera este modelo parametrizado, el cual se desarrollará con más detalle en el capítulo siguiente.

Hacer caminar a un robot bípedo no es una tarea sencilla, hay multitud de frentes de investigación abiertos estudiando este mismo tema, sin que se haya conseguido hasta la fecha un movimiento de caminar tan natural como el que poseemos los humanos. Hay muy buenas aproximaciones, como el robot ASIMO de Honda, el cual es el robot humanoide más avanzado a día de hoy. En la caminata intervienen un gran número de parámetros

que, si bien están muy definidos, son muy numerosos; por lo que si queremos conseguir un movimiento fluido y natural, debemos tener en cuenta gran cantidad de factores. En el proyecto que nos ocupa, la caminata no se va a definir de forma manual, sino que se va a encontrar mediante una búsqueda en un espacio finito y delimitado por nosotros. Se calcula que alrededor de 120 parámetros intervienen en el modelado de una caminata para un robot, teniendo en cuenta todos los grados de libertad de cada una de sus articulaciones. Francisco Rivas consiguió reducir esos 120 parámetros a 10, que permiten modelizar una caminata completa.

Nosotros hemos desarrollado un *plugin* que, a partir de esos 10 parámetros; y basándonos en el modelo matemático de generación de ondas propuesto por él, permite realizar un desenrrollamiento de esos 10 parámetros a todos los necesarios para realizar un movimiento de caminata acompasado. A modo de resumen, la entrada del *plugin* serán los 10 parámetros que modelizan la caminata que este convertirá en vectores de posiciones determinadas en el tiempo para cada articulación.

El *plugin*

El *plugin* desarrollado sigue la estructura básica del resto de *plugins* implementados para las articulaciones individuales, sobreescribiendo los métodos *Load*, *Init* y *OnUpdate*. No obstante, a parte del método *Load*, que nos sirve como puerta de acceso al modelo del Nao, los otros dos métodos están vacíos. En su lugar, y para tener un mayor control sobre los períodos de actualización del controlador, se ha implementado una clase interna (que será ofrecida a través de ICE como ocurre con las clases *Pose3DMotors* o *Pose3DEncoders*) que encapsula toda la funcionalidad.

Esta clase, a la que hemos denominado *Walker* dispone de todos los métodos necesarios para convertir los 10 parámetros que definen una caminata, y que acepta mediante ICE, en un movimiento determinado del robot mediante el cálculo de valores aplicados a funciones determinadas para cada articulación. Teniendo definida una base de movimiento analítica, la cual se basa en funciones específicas que generan ondas que definen el movimiento de cada articulación en el tiempo, basta con transformar esos 10 parámetros entrantes en vectores con valores de posición para cada una de las articulaciones presentes en el movimiento.

Para este fin, se ha incluído en este *plugin* la generación de las ondas acopladas, así como la obtención de valores para cada una de las articulaciones, las cuales dispondrán de un vector independiente para cada una. De esta forma, al acabar el procesamiento de los parámetros de entrada dispondremos de un vector para cada uno de los grados de libertad

de las articulaciones que influyen en la caminata (en nuestro caso concreto disponemos de 18 vectores) de igual tamaño y rellenos con los datos de posición de cada una de ellas en cada instante. Cabe decir que, a diferentes parámetros de entrada, diferetes ondas se generarán, provocando multitud de movimientos posibles.

El de estos vectores se realiza con funciones específicas para cada uno, que aplicando los parámetros de entrada a la base analítica del movimiento definida en el *plugin*, como se puede apreciar en el Cuadro 4.11. No vamos a profundizar en temas de implementación ya que eso se desarrollará en el capítulo siguiente.

Una vez que se han generado los valores para cada una de las articulaciones y están almacenados en sendos vectores, este *plugin* ofrece un método que comienza la simulación comandando a todas las articulaciones del Nao los valores calculados y almacenados previamente. El resultado de invocar a este método con diferentes parámetros de caminata genera un entorno de pruebas ideal para la búsqueda de una caminata óptima para el Nao en Gazebo 5, donde poder evaluar los posibles candidatos a caminata óptima de forma repetible y segura.

Se han escogido dos criterios a la hora de determinar cuándo una simulación ha terminado. Dado que se va a realizar una búsqueda de parámetros en un espacio amplio, van a surgir infinidad de candidatos a evaluar, por lo que el tiempo de simulación es crucial en este caso, para poder estudiar el mayor número de caminatas en un tiempo razonable. Los criterios escogidos han sido: (1) la simulación acabará si han pasado más de 6 segundos desde que comenzó el movimiento; y (2) la simulación acabará inmediatamente si se detecta que el robot se ha caído realizando el movimiento generado. En cuanto al primer criterio, se estima que se puede obtener una caminata de calidad en menos de 10 segundos, por lo que se han elegido esos 6 segundos como límite; en cuanto al segundo criterio, dado que lo que buscamos es que el robot camine, no tiene sentido continuar con la simulación si el robot se ha caído, por lo que la calidad de un candidato a caminata que provoca la caída del robot será, en cualquier caso, nula.

La simulación de una caminata aporta multitud de datos útiles que se pueden evaluar para determinar la calidad de la misma. Así, una vez se ha terminado la simulación de una caminata se generan estadísticas que la describen, ofreciendo todos los elementos necesarios para evaluarla. Los datos que se han seleccionado como relevantes para el estudio de las caminatas son:

- posición inicial del robot antes de comenzar el movimiento. Se ofrecen los tres valores de posición: x, y y z (en centímetros).

```
1  void ankle_roll_value(float frequency, float balance, int pos) {
2
3              float amplitude, phase, shift;
4              float time, moment, freq;
5
6              float result;
7
8              amplitude = ANKLE_ROLL_A*balance;
9              shift = ANKLE_ROLL_S;
10             phase = BASE_PHASE - ANKLE_ROLL_P;
11
12             time = 0;
13             moment = frequency/pos;
14             freq = (float)2*M_PI/(frequency);
15
16             for (int i = 0; i < pos; i++) {
17                 time = time + moment;
18                 result = amplitude*sin((freq*(time))+phase)+shift;
19                 positions_right_ankle_roll.push_back(result);
20             }
21             time = 0;
22             shift = shift + 5;
23             for (int i = 0; i < pos; i++) {
24                 time = time + moment;
25                 result = amplitude*sin((freq*(time))+phase)+shift;
26                 positions_left_ankle_roll.push_back(result);
27             }
28 }
```

Cuadro 4.11: Método que calcula los valores de posición para el pie derecho.

- posición final del robot una vez terminado el movimiento. se ofrecen también los tres valores de posición: x, y y z (en centímetros).
- tiempo de simulación (en segundos).
- distancia recorrida por el robot (en centímetros).
- distancia recorrida en el eje X del movimiento (en centímetros).
- distancia recorrida en el eje Y del movimiento (en centímetros).
- error cuadrático medio de los movimientos laterales del robot.
- si se ha caido o no
- función salud o *fitness* que evalúa el movimiento.

Tanto la posición inicial como la posición final del robot sirven para calcular distancias recorridas en ambos ejes, como la distancia total conseguida. Dado que nuestro algoritmo de *fitness* premia la distancia recorrida ante todo, estos valores son esenciales para la evaluación del movimiento. El tiempo de simulación también es un factor importante en nuestra evaluación, ya que la velocidad de la caminata también se premia. Tendrá más calidad una caminata que recorra X cm en 4 segundos que en 6 segundos. La distancia se calcula a partir de los valores de posición, siendo el eje principal de la evaluación de los movimientos, al ser lo que más valor tiene en nuestra evaluación. Las distancias separadas por ejes, si bien no son tan importantes para el cálculo de la calidad de la caminata, son una buena forma de complementar la información obtenida de la caminata. La desviación, o error cuadrático medio mide la fluctuación existente en el movimiento. Si un robot oscila mucho sobre el eje sobre el que está realizando la caminata, este valor se incrementará reduciendo la calidad de la caminata consigo. Sin embargo, si el robot consigue una trayectoria lo más recta posible, este valor se decrementará no reduciendo así la calidad de la misma. Por último, se contempla una variable que dictamina si el robot se ha caído intentando realizar el movimiento, lo cual es útil para descartar candidatos de forma directa. El valor de salud o *fitness* se ha incluído en esta estructura de datos por completitud, sin embargo, no es el *plugin* el que calcula este valor, sino el componente JdeRobot encargado de generar los candidatos a caminata y enviar las órdenes al *plugin*

Una vez que se disponen de todos los datos necesarios para evaluar la caminata, el componente JdeRobot encargado de generar los candidatos y enviar las órdenes al *plugin*, pide los datos a éste para realizar los cálculos necesarios (que se detallan en el capítulo siguiente) y almacenando los resultados en ficheros en disco, para su posterior consulta.

Interfaz ICE específica

Para hacer posible la comunicación entre el *plugin* de caminatas y el componente JdeRobot que hace uso de él ha sido necesaria la creación de un interfaz ICE específico para dicho *plugin*. Como sucede con las interfaces *Pose3DMotors* o *Pose3DEncoders* del resto de *plugins*, necesitábamos una interfaz que nos permitiera transportar la estructura de datos creada, con los diez parámetros que definen la caminata, entre el componente y el *plugin*. Para ello se definió la interfaz *walker.ice* cuyo contenido se puede apreciar en el Cuadro 4.12.

Como se puede observar, este interfaz ICE se compone de dos estructuras de datos correspondientes a la entrada del *plugin*: *WalkerData* y a la salida del mismo: *StadisticsData*, de tal forma que un componente que incluya la cabecera de este interfaz, será capaz de comunicarse con el *plugin* evaluador de caminatas mediante ICE. Además, este interfaz declara cuatro funciones que se tendrán que implementar en el *plugin*. Estas funciones son:

- **StartWalk()**: función que empieza a evaluar una caminata con los valores para las articulaciones calculados previamente.
- **StopWalk()**: función que detiene una caminata.
- **setParams(WalkerData)**: esta función recibe como parámetro la estructura de datos definida en el interfaz y sirve para enviar al *plugin* la información con los diez parámetros que definen la caminata.
- **getStadistics()**: función que entrega las estadísticas almacenadas en el tipo de dato definido en la interfaz ICE, de la caminata que acaba de evaluar.

Esta es, en resumen, la estructura de este *plugin* de alto nivel que se ha desarrollado como novedad. En el capítulo siguiente estudiaremos más en profundidad cómo se calculan los valores de posición para cada una de las articulaciones, así como cómo se evaluúa la calidad de una caminata simulada a partir de las estadísticas que nos ofrece el simulador.

4.4.2. Acceso a las articulaciones

Por último, para concluir esta sección, se detalla a continuación la comunicación existente entre el *plugin* de alto nivel desarrollado y los *plugins* de bajo nivel existentes y refactorizados para cada articulación.

```
module jderobot{
    class WalkerData
    {
        float param1;
        float param2;
        float param3;
        float param4;
        float param5;
        float param6;
        float param7;
        float param8;
        float param9;
        float param10;
    };
    class StadisticsData
    {
        int id;
        float x0;
        float y0;
        float z0;
        float x1;
        float y1;
        float z1;
        float simTime;
        float distance;
        float distanceX;
        float distanceY;
        double desviation;
        int fallen;
        float fitness;
    };
    interface Walker
    {
        int startWalk();
        int stopWalk();
        int setParams(WalkerData data);
        StadisticsData getStadistics();
    };
}; //module
```

Cuadro 4.12: Interfaz específico ICE para el *plugin* de caminatas.

La forma en la que se conectan los distintos componentes de JdeRobot con los *plugins* de los robots que se simulan en Gazebo es mediante interfaces de comunicación ICE, como se puede ver en la Figura 4.10. No obstante realizar este tipo de comunicación entre los *plugins* (que están en la misma capa de abstracción todos y cada uno de ellos no resulta eficiente debido a las latencias innecesarias que se incluirían si salgo del entorno de gazebo mediante una comunicación con ICE, para después volver a entrar. Esta latencia puede ser despreciable en un ámbito local (si solo trabajamos con un equipo), sin embargo es tremendamente ineficiente si queremos distribuir la funcionalidad de nuestro sitema. Por ello, se llegó a la solución de mantener punteros de cada uno de los *plugins* existentes a los cuales podría acceder desde el *plugin* de alto nivel de caminatas.

Para resumir el esquema de comunicación que se ilustra al principio de esta sección, los componentes de JdeRobot podrán conectarse a los *plugins* de forma individual mediante interfaces de comunicación ICE, como se explica en el apartado 4.3.2, mientras que los *plugins* podrán comunicarse entre sí a traves de punteros hacia sus clases que se instanciarán una vez por ejecución. Este modo de implementar la comunicación elimina toda latencia posible asociada a una comunicación en red, lo que permite borrar cualquier ruido que afecte a nuestra simulación.

4.5. Herramientas para experimentación

Una vez descrito todo el soporte para el robot Nao tanto real como simulado, se van a describir a continuación las herramientas desarrolladas para la validación de este soporte. Las herramientas desarrolladas son básicamente dos: una que engloba un conjunto de herramientas que controlan cada articulación por separado; y otra que permite realizar movimientos algo más complejos incluyendo varias articulaciones a la vez en el movimiento. Todas las herramientas desarrolladas carecen de interfaz, ya que se han usado principalmente como herramientas de depuración de los *plugins* refactorizados, por lo que se han utilizado *callbacks* asociadas a eventos de teclado para controlar las articulaciones asignando teclas específicas para cada movimiento posible de cada articulación.

4.5.1. Componentes individuales

Para comenzar, una vez completados tanto el modelo del robot, como los *plugins* asociados al modelo, se han desarrollado una serie de componentes JdeRobot muy simples para el control de las articulaciones por separado. En concreto se ha implementado

un componente por articulación. Las articulaciones que nos interesan del Nao son 11: cuello, hombro izquierdo, hombro derecho, codo izquierdo, codo derecho, cadera izquierda, cadera derecha, rodilla izquierda, rodilla derecha, tobillo izquierdo y tobillo derecho. Cada componente desarrollado controla las articulaciones tanto del lado izquierdo del robot, como del lado derecho, por lo que en total han resultado seis componentes:

1. componente que controla el movimiento del cuello.
2. componente que controla el movimiento de los hombros.
3. componente que controla el movimiento de los codos.
4. componente que controla el movimiento de las caderas.
5. componente que controla el movimiento de las rodillas.
6. componente que controla el movimiento de los tobillos.

Todos estos componentes se conectan a las articulaciones del robot simulado a través de las interfaces ICE definidas en cada *plugin*, de modo que, por ejemplo, el componente controlador del tobillo genera dos conexiones a sendos tobillos (el izquierdo y el derecho) pudiendo así mandar órdenes de posición a los mismos.

Además, estas herramientas carecen de interfaz de usuario, debido a que el fin para el que estaban diseñadas era esencialmente depurativo. Bastaba con comprobar que las órdenes que estos componentes comandaban a las articulaciones simuladas llegaban a Gazebo y se aplicaban de forma correcta. Por ello, el control de estos componentes básicos está basado en *callbacks* generados por eventos del teclado. De tal forma que, se escogieron un conjunto de teclas específicas del teclado y se les dotó de un sentido en el componente. En el caso del controlador del tobillo, por ejemplo, estas teclas son: w y s para aumentar y reducir el valor del pitch del tobillo derecho respectivamente, a y d para aumentar y reducir el valor del roll del tobillo derecho respectivamente, y de forma análoga i y k para el control del pitch del tobillo izquierdo, y j y l para el control del roll del tobillo izquierdo. Por supuesto, cada componente tiene mapeadas sus propias *hotkeys* en función de los grados de libertad de la articulación que controlen.

La Figura 4.11 ilustra la arquitectura de comunicaciones de estas herramientas *ad-hoc*.

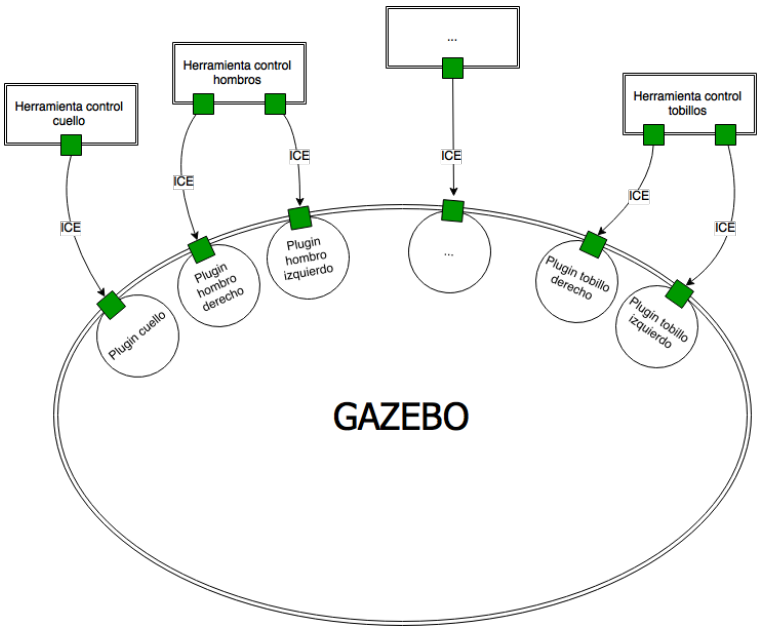


Figura 4.11: Esquema de actuación de los *plugins* de las articulaciones y las herramientas desarrolladas.

4.5.2. Componente avanzado de movimientos combinados

Una vez depurados y probados todos los *plugins* individualmente, la idea era comenzar a generar movimientos más complejos que involucraran varias articulaciones a la vez, con vistas a la generación de un movimiento mucho más complejo, y objetivo de este proyecto: las caminatas. Dado que el salto entre el control de articulaciones individuales y la evaluación de caminatas era demasiado grande, optamos por desarrollar otra herramienta con carácter depurativo que permitiera comandar unos movimientos en un punto medio de dificultad, tales como saludar con los brazos, con la cabeza, realizar un pequeño y muy simple baile, etc. Para este fin se desarrolló una herramienta capaz de conectarse a varios *plugins* de las articulaciones a la vez a través de las interfaces ICE que estoss ofrecen y ordenar acciones relativamente sencillas a los actuadores simultáneamente.

A diferencia de los componentes desarrollados *ad-hoc* para depurar y probar el movimiento de las articulaciones en simulación por separado, este componente es capaz de conectarse a todas las articulaciones del robot simulado al mismo tiempo. Por lo tanto, este componente genera 11 conexiones ICE, una para cada *plugin* asociado a cada articulación que nos interesa. Además de no tener interfaz, esta herramienta no es controlada en tiempo real por el usuario, como sí pasaba con las anteriores. No disponemos de teclas para comandar órdenes complejas ya que la abstracción de que un movimiento como mover el

brazo de arriba a abajo asociada a una tecla no era intuitiva. Por ello se decidió almacenar movimientos prefijados en ficheros de texto que contenían las diferentes posiciones de cada articulación involucrada en el movimiento en cada instante. Estas acciones eran secuencias fijas de movimientos, que a modo de fotogramas se definían en el ficher de texto con el siguiente formato:

```
time
val_neck_pitch
val_nec_yaw
val_right_shoulder_pitch
val_right_shoulder_roll
val_left_shoulder_pitch
val_left_shoulder_roll
val_right_elbow_yaw
val_right_elbow_roll
val_left_elbow_yaw
val_left_elbow_roll
...
val_left_ankle_pitch
val_left_ankle_roll
```

De tal forma que se especificaba para cada momento en el tiempo un bloque, indicando los valores que debía tomar cada articulación en el instante determinado.

Esta forma de definir movimientos, si bien es sencilla, no es práctica ni flexible si queremos conseguir movimientos complejos como el de caminar, debido a la gran cantidad de parámetros a definir. No obstante, es muy útil para el tipo de movimientos sencillos que se querían probar y que implicaban varias articulaciones al mismo tiempo, para comprobar el comportamiento de éstas ante acciones simultáneas. Los movimientos que se probaron fueron: mover un brazo de arriba a abajo (articulaciones implicadas: hombro y codo), mover ambos brazos en modo espejo (articulaciones implicadas: hombro y codo) y balancear el robot de un lado a otro usando las caderas. Esto nos daba pie a estudiar el movimiento de las articulaciones de la cadera para el siguiente paso en nuestro proyecto.

Capítulo 5

Caminata

La consecución del modelo simulado estable y más robusto para el humanoide Nao en el simulador Gazebo, abre la puerta a posibles desarrollos comportamientos con locomoción y automatización de movimiento. En este capítulo se va a describir el desarrollo llevado a cabo para dotar de movimiento al Nao simulado. Estudiaremos cómo se han modelizado las formas de andar del robot simulado, basándonos en las ideas y los desarrollos realizados por Francisco Rivas [Rivas, 2011]. Posteriormente se detallará cómo son evaluadas esas caminatas generadas, mediante una función salud o *fitness*, para concluir con el componente software desarrollado para dar soporte a la generación de caminatas en el Nao simulado.

Partíamos de la base de tener un modelo de simulación preciso y coherente conseguido en las fases anteriores de este proyecto sobre el que podíamos investigar. Además, gracias a la refactorización aplicada al soporte anterior a este proyecto disponíamos de las últimas tecnologías en materia de simulación al estar desarrollando todo el proyecto sobre las versiones más recientes de Gazebo y JdeRobot. Este era el contexto en esta fase del desarrollo, por lo que teniendo cerrados los objetivos previos, este capítulo sólo se centrará en los aspectos relacionados con la caminata del robot simulado, y no tanto en la simulación en sí misma, vista en el Capťulo 4.

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 una: cadera, rodilla y tobillo. Esta marcha se considera periódica debido a que encadena pasos. Encadenando pasos se consigue un movimiento que llamamos caminata. Si nos desplazamos a una velocidad constante, estos pasos que componen la marcha se vuelven completamente iguales, lo que nos ofrece un punto de entrada para abordar la generación de caminatas, al poder simplificar la marcha del Nao en una sucesión de pasos iguales.

Basándonos en este hecho, tenemos como caracterísitica principal de una caminata que la sucesión de pasos se realiza de forma simétrica entre las dos piernas, con un cierto retraso en los actuadores de una y otra pierna. Esta sincronización es crucial para la estabilidad del robot, ya que si no es precisa, el movimiento hará que el robot se caiga.

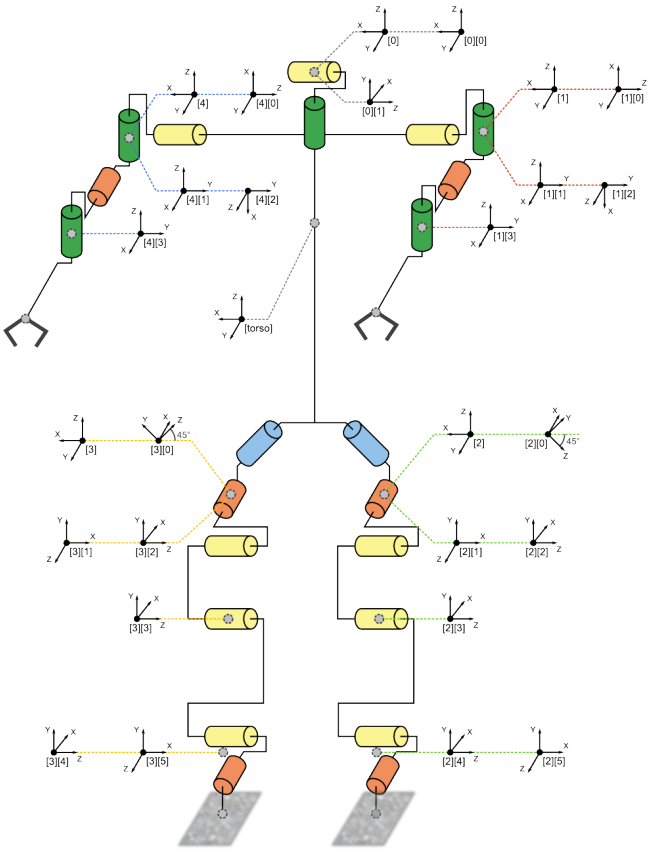


Figura 5.1: Esquema Denavit-Hartenberg del robot Nao.

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, las cuales nos ofrecen 6 grados de libertad repartidos como sigue y como se puede ver en la Figura 5.1.

- **cadera** con 3 grados de libertad: *pitch*, *yawpitch* y *roll*.
- **rodilla** con 1 grado de libertad: *pitch*.
- **tobillo** con 2 grados de libertad: *pitch* y *roll*.

Para generar ondas distintas de caminar tendríamos que modificar el valor de los actuadores en cada fotograma del movimiento, lo cual es demasiado complejo y costoso,

teniendo en cuenta que si intervienen 12 actuadores en la marcha y definimos el movimientos en 10 fotogramas, serían 120 parámetros (los valores concretos de todos los actuadores en cada fotograma) que fijar para conseguir una nueva forma de caminar. Con lo expuesto anteriormente, esta tarea carece de la seguridad que ofrecen otras alternativas como, por ejemplo, la que vamos a estudiar a continuación: la generación de ondas acopladas que definen el movimiento.

5.1. Modelo de caminata basado en ondas acopladas

Como la paramtrización de un paso con 120 parámetros es excesiva si se desea buscar en el espacio de las posibles caminatas, tenemos que encontrar la forma de reducir ese número hasta conseguir un espacio de búsqueda que sea factible de explorar tanto en tiempo como en dimensión. Una posible solución a esto, que ya desarrolló Francisco Rivas [Rivas, 2011], y que aquí vamos a explicar sólo la superficie, es generar para cada actuador que interviene en la marcha una función que defina el movimiento. De esta forma los parámetros ya no dependerían del núnero de fotogramas que componen el movimiento, sino de los parámetros de la función característica. El objetivo final de esta modelización de la caminata es conseguir un número de parámetros lo suficientemente reducido para poder realizar una búsqueda en un espacio menor y obtener resultados de forma más inmediata.

Francisco Rivas generó una base de movimiento explícita basada en para cada articulación presente en el movimiento. Tras un estudio exhaustivo, llegó a la solución que se muestra en la Figura 5.2, la cual muestra las ondas obtenidas para cada articulación individual y que sirven como base del movimiento de caminar. Como se puede ver, en la mayoría de los casos estas ondas no corresponden con ondas sinusoidales, sin embargo las funciones propuestas por Francisco modelizan la onda con los mismos parámetros que una onda sinusoidal: amplitud, frecuencia, fase y desplazamiento.

No vamos a entrar en detalles del modelo desarrollado por Francisco, ya que se puede encontrar toda la información en su proyecto de fin de carrera, sino que vamos a ofrecer un contexto general de cómo se ha llegado a la reducción de parámetros de la caminata para el humanoide Nao.

Se parte de la idea de que tenemos un movimiento definido por 10 fotogramas en el cual intervienen 12 actuadores. Lo que hace un total de 120 parámetros que fijar para conseguir un modo de caminar. Como hemos explicado, esto no es factible, por lo que

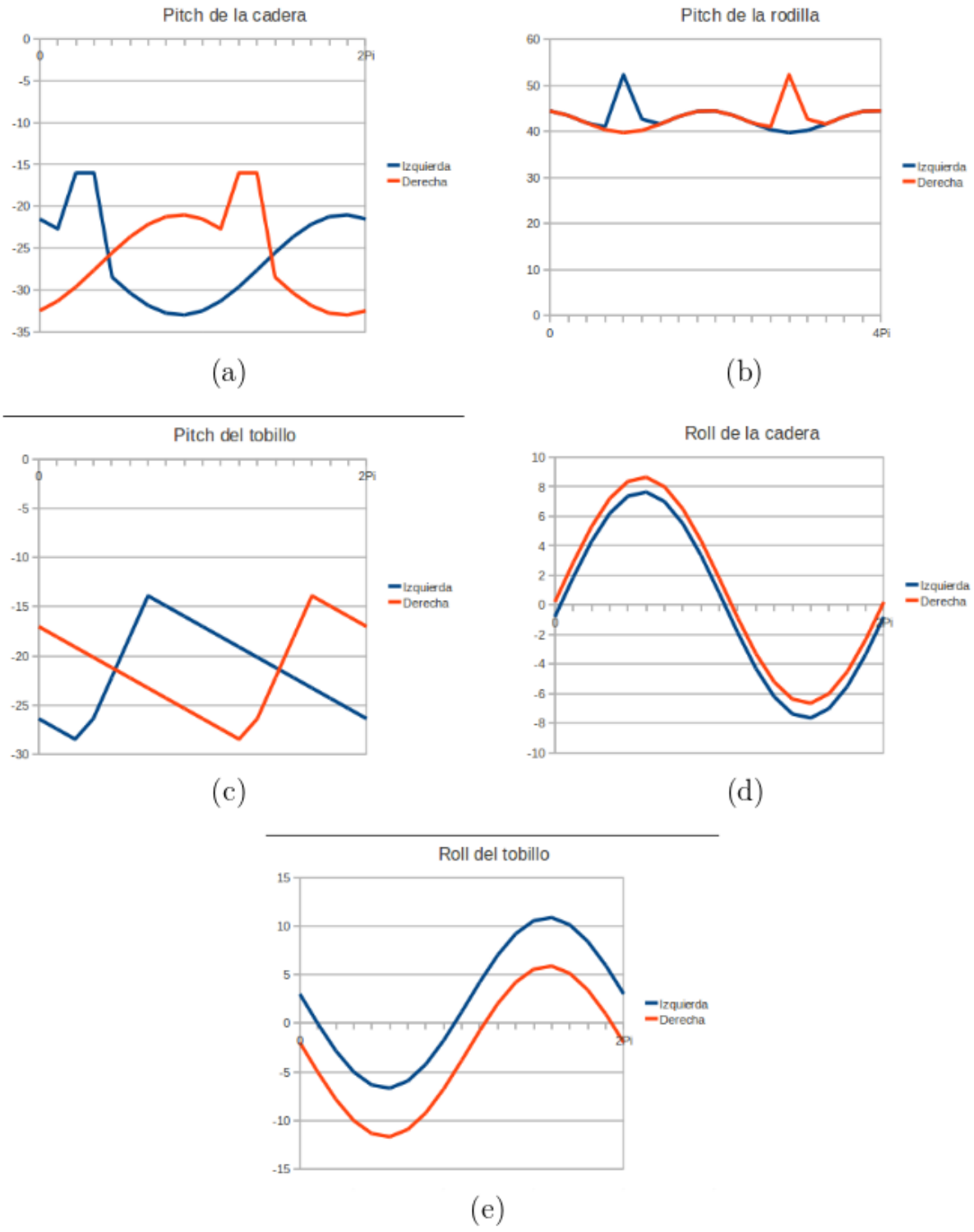


Figura 5.2: Graficas de las ondas que definen el movimiento de los actuadores involucrados en la caminata.

hay que reducirlos Para empezar, y dado que nuestro objetivo es conseguir un movimiento en línea recta, obviaremos los *yaw* de la cadera, que son los que controlan la apertura de las piernas en la caminata, y dado que queremos una caminata estable y apuntando hacia adelante fijaremos los valores de los *yaw* de ambas caderas a cero, por lo que hemos reducido los actuadores que intervienen en la marcha de 12 a 10.

Por otra parte, conseguida la base del movimiento basada en ondas moduladas éstas por 4 parámetros: amplitud, frecuencia, fase y desplazamiento; podemos afirmar que en el movimiento sólo influirán 4 parámetros por actuador, lo cual, al tener 10 actuadores que intervienen en la marcha generan 40 parámetros en total. Con lo que hemos reducido la cantidad de parámetros de 120 a 40. No obstante, 40 sigue siendo una gran cantidad de parámetros, por lo que se estudió el seguir reduciéndolos.

Como hemos apuntado ya, la sincronización de todas los actuadores es crucial, por lo que todos ellos deben funcionar a la misma frecuencia. Con esto, tenemos que de los 40 parámetros hemos eliminado 10 frecuencias de todos los actuadores para dejar 1 sola: la *frecuencia fundamental*. Por lo tanto tenemos que *40 - 10 + 1 = 31* parámetros.

También se acordó que los movimientos del robot a la hora de caminar a una velocidad constante son simétricos con un cierto retardo (el tiempo que dura medio paso exactamente), con lo cual, podemos modelar el comportamiento para una sola pierna, haciendo que la otra dependa de ésta aplicando un cierto desfase a las funciones que definen el movimiento. Por ello, ahora tenemos 5 actuadores con 3 parámetros cada uno más la frecuencia fundamental, que hacen un total de 16 parámetros.

Aunque parezcan pocos, 16 parámetros siguen siendo demasiados para realizar una búsqueda, teniendo en cuenta los rangos de valores en los que se mueven dichos parámetros. Por lo tanto se deben crear ciertas dependencias entre ellos para reducir aún más su número. Para esto, se decidió hacer que las fases de todas las articulaciones dependieran de una sóla que sería la que marcaría la sincronización del movimiento. La fase elegida fue la del *pitch* de la cadera que define la posición en la que va a comenzar el robot. Es por ello que este valor puede dejarse fijo y eliminarse de los parámetros de búsqueda quedando así 15 parámetros.

Por último se incluyó un nuevo parámetro llamado *amplitud del balanceo* que afecta directamente a los actuadores modelizados con ondas sinusioidales (los *roll* de la cadera y el tobillo); y que consegía eliminar 5 parámetros más al establecer dependencias directas entre la fase de la cadera y los *roll* de la cadera y el tobillo. Con la inclusión de la amplitud del balanceo se consigue eliminar:

- las amplitudes del *roll* de la cadera y el tobillo, que dependerán del nuevo parámetro.
- las dos fases del *roll* de la cadera y el tobillo, que serán directametamente dependiente de la fase del *pitch* de la cadera
- los dos desplazamientos del *roll* de la cadera y el tobillo, ya que al igual que las amplitudes dependerán del nuevo parámetro.

Con todo esto se logró reducir el número de parámetros que intervienen en una caminata de 120 a 10, que son razonables para comenzar a realizar una búsqueda en el espacio de parámetros, que es el de posibles movimientos.

Para la adaptación a este proyecto, se implementaron todas las funciones que definen la base del movimiento en el plugin explicado en el Capitulo 4, dando lugar a la generación de ondas acompasadas dentro del mismo plugin y a los valores para las articulaciones en cada instante. De modo que toda la expansión de esos 10 parámetros y la traducción a valores concretos para todos los actuadores en los diferentes instantes del periodo que dura el paso queda centralizada en el propio plugin de caminatas. Esto permite a las aplicaciones de JdeRobot pensar en caminatas en vez de actuadores individuales y permite organizar la búsqueda de parámetros en un componente externo al plugin.

5.2. Evaluación de las caminatas

Una vez adaptado el modelo de paso a nuestras necesidades en este proyecto, habiéndolo implementado dentro del plugin de caminatas desarrollado para el mismo, necesitamos una forma de evaluar la calidad de una caminata tentativa para elegir la óptima.

Para este fin disponemos de un simulador para el cual hemos definido un modelo del Nao robusto y realista del que se tiene control total y con el que se pueden realizar pruebas de caminatas en entornos controlados y seguros. Dado que una de las ventajas de los simuladores es la posibilidad de repetitr una prueba bajo las mismas condiciones una y otra vez, podemos establecer una evaluación de movimientos a partir de la información estadística que nos ofrece Gazebo.

En nuestro caso premiamos la distancia recorrida junto con la velocidad a la que se efectúa la caminata, la estabilidad y 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 existen variaciones en la linealidad del movimiento, el

robot deberá compensar dichas variaciones con giros y otros movimientos que no se han implementado para este proyecto, por lo que este parámetro será fundamental a la hora de decidir la calidad de la caminata.

Además de todas estas consideraciones, han sido descartadas todas aquellas caminatas que acaban con la caída del robot, de modo que aunque este avanzara una distancia considerable en un corto período de tiempo, si al final se caía, se establece que la calidad de esa caminata es nula.

Todos estos parámetros forman parte de la ecuación salud que define la calidad de una caminata determinada. La ecuación que se ha utilizado para este proyecto es la misma que definió Francisco Rivas en su proyecto de fin de carrera, y que se detalla a continuación:

Distancia. La distancia se obtiene a partir de la posición inicial y la posición final del robot con la siguiente ecuación:

$$distancia = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}$$

Error cuadrático medio. La estabilidad se calcula iterativamente a medida que avanza la simulación a partir de las variaciones en x e $\mathbf{y}$ de la posición del robot en el mundo simulado. Cuanto menor sea este valor de error más alta será la calidad de la caminata debido a que la estabilidad ha sido buena. La ecuación que calcula este error es:

$$error = \frac{\sum_{x=0}^{n} \sqrt{(x_{actual} - x_{anterior})^2 + (y_{actual} - y_{anterior})^2}}{n}$$

Linealidad. El cálculo de la linealidad se realiza a partir de la posición inicial y la posición final del robot calculando el desplazamiento total e indicando cuán en línea recta ha ido el robot. La linealidad se calcula como:

$$linealidad = \begin{cases} \frac{y_0}{y_0+(y_1-y_0)}, & si(y_0 - y_1 < 0) \\ \frac{y_0}{y_0+(y_0-y_1)}, & si(y_0 - y_1 >= 0) \end{cases}$$

Con estos tres factores definidos, que pueden verse de forma gráfica en la Figura 5.3, con sus respectivas ecuaciones, la función salud empleada ha sido la siguiente:

$$salud = \sqrt{\frac{distancia^3}{tiempo}} * \sqrt{\frac{1}{error}} * linealidad$$

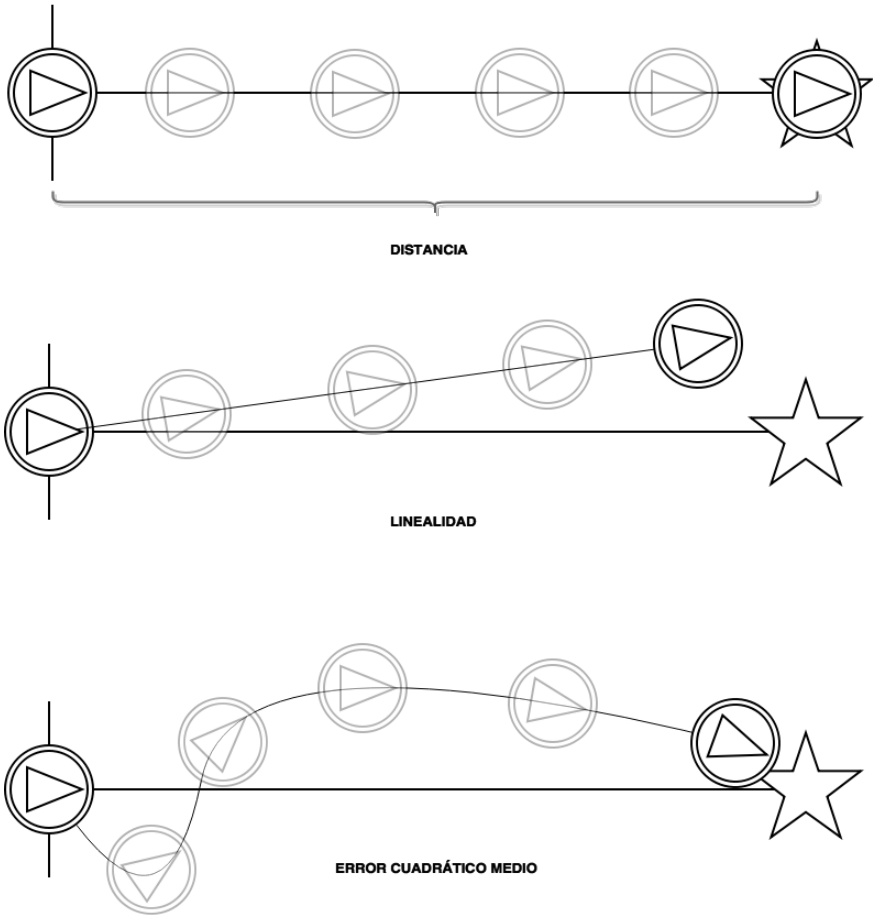


Figura 5.3: Factores que intervienen en la función salud.

Esta función salud, como puede comprobarse, resulta bastante discriminante a la hora de evaluar caminatas dándole el mayor peso a la distancia recorrida, debido a que conseguir una forma de caminar en la que el robot sea capaz de moverse de forma rápida y estable es complicado.

5.3. Busqueda rastrillada

En este estado del proyecto se disponía de un soprte totalmente operativo para el robot simulado Nao, con unos controladores actualizados que proporcionaba una simulación robusta, además de un modelo realista del humanoide y un modelo matemático que define y evalúa los movimientos del robot. Todos estos ingredientes permitían experimentar con el robot en simulación en la búsqueda de caminatas.

Para realizar esto, se ha desarrollado un componente en JdeRobot llamado *Nao Walking Component*,(Figura 5.4) cuyo objetivo es dar soporte a todo lo desarrollado anteriormente en cuanto a caminatas se refiere. Este componente, en esencia, permite realizar búsquedas sistemáticas de los parámetros en los que se ha definido la caminata del Nao y mandarlos al simulador para poder evaluarlos. Para ello ofrece un interfaz de usuario liviano y sencillo que permite realizar estas acciones.

El espacio de búsqueda de los parámetros que definen los modos de caminar de nuestro humanoide es inmensamente grande. De tal modo que, sería imposible encontrar una caminata óptima, o incluso una válida, si no se reduce ese espacio. Para reducir este espacio de búsqueda se ha diseñado e implementado un algoritmo que acota todos los parámetros que intervienen en el movimiento a unos rangos que se han considerado adecuados. Además, y para reducir aún más el espacio de búsqueda, a cada parámetro se le ha asignado un incremento determinado que influye en la granularidad (número de elementos por parámetro) de la búsqueda. Los límites se han fijado como sigue:

- 18 niveles para la frecuencia del movimiento.
- 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.

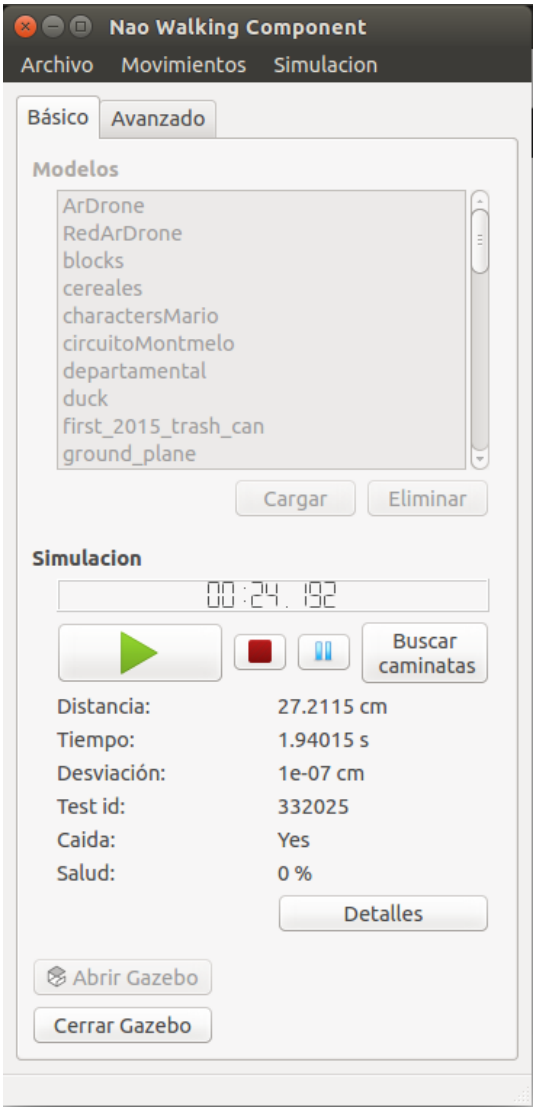


Figura 5.4: Interfaz gráfica del componente *Nao Walking Component*.

- 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.

Con los parámetros acotados y los incrementos definidos, el espacio de búsqueda se reduce considerablemente a unas magnitudes razonables. Aún así haría falta mucho tiempo (meses) para recorrer todo ese espacio. Una de las soluciones propuestas por Francisco Rivas en su proyecto de fin de carrera era dejar algunos de los parámetros que definen la caminata fijos y realizar la búsqueda sobre el resto, reduciendo así las posibilidades de búsqueda. No obstante, esto tiene la implicación negativa de que muchos de los candidatos posibles de ese espacio de búsqueda no serán evaluados. Por lo tanto, la propuesta alternativa es realizar una evaluación aleatoria de todos los candidatos a caminata generados en la búsqueda, de tal modo que, en lugar de recorrer el espacio de búsqueda secuencialmente, este recorrido se haga aleatoriamente, lo que aumenta las probabilidades de encontrar antes el mejor candidato a caminata óptima.

El componente *Nao Walking Component* es el encargado de dar las órdenes que ponen todo el sistema en funcionamiento. Como se puede ver en la Figura 5.4, dispone de 3 botones de control para la simulación, y un cuarto para realizar la búsqueda que generara los candidatos a caminata. Además, con un cronómetro que cuenta el tiempo total invertido desde que se empezó con la búsqueda y evaluación de las caminatas. Por otro lado tiene elementos para controlar la apertura y el cierre del simulador Gazebo desde el mismo componente, sin tener que realizar este paso en un terminal separado, centralizando toda la funcionalidad y controlando todos los aspectos de la ejecución, además de una lista de todos los modelos que hay en el directorio de modelos de Gazebo, para cargar los robots en el mundo de forma cómoda. El mismo componente genera un mundo sencillo en el que incluye el robot que se le ha indicado mediante el interfaz y conecta sus interfaces ICE haciendo uso de los ficheros de configuración de cada articulación.

Por otro lado, el botón de *Buscar caminatas* ejecuta el algoritmo de búsqueda rastrillada diseñado, de tal forma que sea robusto ante paradas de simulación tanto voluntarias como involuntarias asegurando la persistencia de los datos en ficheros de texto destinados tanto a los candidatos generados, como a los resultados obtenidos de las diferentes evaluaciones.

Cada vez que se genera un nuevo candidato a caminata se genera una entrada en el archivo de texto de las búsquedas con el siguiente formato:

```
Search#4129
param1:0.6
param2:0
param3:-100
param4:30
param5:2
param6:40
param7:20
param8:-3
param9:-75
param10:0
```

Donde se almacena, en la primera línea el identificador de caminata único, y en el resto los valores de cada uno de los parámetros sobre los que se ha realizado la búsqueda. Por otra parte, en un fichero análogo se almacenan los resultados de las diferentes evaluaciones de las caminatas, siempre que estas pasen la criba impuesta: que el robot no haya caído durante el desarrollo del movimiento y que sobrepase cierto valor de calidad la evaluación final de la caminata, en este caso el 70 %. Estas estadísticas, además, se muestran en tiempo real en el interfaz de usuario una vez se ha evaluado una caminata. El fichero de las estadísticas tiene el siguiente formato:

```
Search_id#354717
Simulation time: 6.0 s
Origin: (0 cm, 0 cm, 33.5 cm)
End: (200 cm, 0.00 cm, 33.5 cm)
Distance: 200.0 cm
Desviation: 0.0001 cm
Fallen: 0
Fitness: 100.0%
```

De donde:

- *Search_id* es el identificador único de la caminata evaluada.
- *Simulation Time* es el tiempo empleado para la simulaciíon de una determinada caminata.
- *Origin* es el punto de origen antes de empezar el movimiento.
- *End* es el punto donde termina el Nao al finalizar el movimiento.
- *Distance* es la distancia recorrida con la caminata evaluada.
- *Deviation* indica el error del movimiento. Fluctuaciones en la trayectoria.
- *Fallen* indica si el robot se ha caído o no. Utiliza los valores de verdad de C++ (0 = falso, !0 = verdadero).
- *Fitness* es el valor de la función salud para esa caminata.

La inclusión de esta funcionalidad permite detener las simulaciones completamente, y retomarlas por el punto en el que se quedaron, ya que todos los datos son almacenados en ficheros. De esta forma, se da la posibilidad al usuario de detener el proceso de búsqueda y evaluación (muy costoso en cuanto a carga de CPU), para que pueda tener a su disposición toda la potencia de su máquina si la requiriera para otra cosa y luego retomar la búsqueda.

En cuanto al funcionamiento del sistema, se ha encontrado un problema crucial relacionado con Gazebo y del cual se desconoce su causa. Mientras pasan las iteraciones, el proceso de Gazebo va consumiendo espacio en disco que no se libera hasta que no se cierra el proceso. Una posible causa es que algún archivo de log esté creciendo descontroladamente sin ser visto y que provoque este problema. Para esquivarlo,se cierra Gazebo cada cierto tiempo, de modo que ese espacio es liberado, pudiendo rearrancar gazebo y seguir con la simulación normalmenete. El modo en el que se arranca y detiene el simulador a través del componente es mediante llamadas al sistema operativo con la abstracción de alto nivel que ofrece C++ para este propósito, la instrucción *system(" comando ")*. De este modo, ejecutando la instrucción *system* con el comando *"gzserver tmp/nao.world"* se abriría un hilo en el que se ejecutará Gazebo sin interfaz gráfico cargando el mundo que hemos generado anteriormente con el propio interfaz de usuario al cargar el modelo; mientras que con el comando *"killall gzserver > /dev/null"* detendríamos cualquier proceso en el sistema con el nombre *gzserver* y redireccionando la salida de ese comando a /dev/null para que no muestre nada por la salida estándar.

Capítulo 6

Conclusiones y trabajos futuros

Como último capítulo para cerrar este proyecto, a continuación se presentan las conclusiones resultantes de la realización de este proyecto, además de evaluar los resultados obtenidos en función de los hitos marcados al principio del mismo. Por último se dedicarán un último punto a desatacar las posibles líneas de investigación que este proyecto abre consigo.

6.1. Conclusiones

Antes de la realización de este proyecto, se acordó en que el objetivo principal era la actualización del soporte que ofrece JdeRobot en cuanto al robot humanoide simulado, con vistas a abrir nuevos frentes de desarrollo para los robots humanoides en la plataforma. Como objetivo adicional, se optó por la refactorización de los modos de caminar existentes en JdeRobot, operativo para el robot real (el cual llama a funciones de NaoQi) y para el simulador Webots, de tal forma que fuera posible evaluar caminatas utilizando la última versión del simulador Gazebo.

Tras este proyecto se dispone en JdeRobot de un soporte para el robot humanoide Nao, estable, robusto y muy flexible a cambios. Este es uno de los puntos más importantes que se han conseguido, dado que el simulador con el que trabajamos no ofrece soporte para robots de este tipo, por lo que a partir de ahora se puede utilizar el modelo actualizado para realizar multitud de desarrollos en un entorno controlado y de forma repetible. El modelo anterior ofrecía control total sobre todas las articulaciones del robot humanoide por separado, no obstante, con la nueva versión del simulador, este modelo quedó obsoleto y poco estable en cuanto a capacidad de movimientos se refiere, dado que las propiedades físicas de las

que disponía el modelo eran las estándar de Gazebo. Elementos tan importantes como los tensores de inercia del modelo han sido modificados con los valores reales ofrecidos por el fabricante, de modo que la simulación ha ganado en realismo, precisión y posibilidades de desarrollo.

Este objetivo ha sido el más complejo de conseguir con diferencia, ya que la refactorización del modelo conllevaba el estudio tanto del formato de archivo descriptor del fichero, así como el impacto que tenían las diferentes propiedades físicas aportadas al modelo. Nos econtramos con infinidad de problemas para dotar de estabilidad al Nao en estático debido a la gran cantidad de factores de los que depende que se mantenga en pie sin fuerzas externas actuándo sobre él. Por otra parte, no sólo bastaba con tener un modelo fiable y cercano a la realidad, sino que también se propuso como uno de los objetivos, conseguir que las mallas del Nao (su aspecto visual) fueran aún más cercanas a la realidad, para lo cual hubo que, en primera instancia, conseguir las mallas oficiales del fabricante y procesarlas para que se ajustaran al modelo desarrollado. Esta tarea no fue trivial, dado que por temas de licencias, las mallas no podían ser de acceso público, por lo que tuvimos que apoyarnos sobre la comunidad de desarrollo de ROS para conseguirlas.

Por otro lado, había que refactorizar los controladores o plugins del Nao para hacerlos compatibles con la nueva versión de Gazebo. Esto supuso el estudio de la API de Gazebo para determinar cómo las funciones que se habán quedado obsoletas afectaban al modelo ya desarrollado del robot. Esto tiene implicaciones serias en la estabilidad del robot, ya que son los plugins los encargados de dictaminar qué valores de fuerza o velocidad le llegan a cada actuador en cada momento. Es por este motivo que la refactorización se tuvo que hacer articulación por articulación, ya que una sola articulación inestable podría comprometer la estabilidad del robot, haciendo que la simulación no fuera, en absoluto, realista. Una vez estudiada la API de Gazebo y localizadas las funciones obsoletas los cambios no fueron drásticos y se pudo reutilizar la mayor parte de los plugins existentes, sustituyendo sólo aquellas funcionalidades que se habían quedado obsoletas y afectaban al funcionamiento final del robot simulado.

Por lo que respecta al movimiento del robot, la inclusión de un *plugin* que permita el control de alto nivel de todas las articulaciones del robot implica que la escalabilidad del proyecto aumente en gran medida. Con este *plugin* no sólo se pueden generar modos de caminar, sino que se puede generar una amplia variedad de tipos de movimiento basados tanto en la parametrización de los mismos, como en valores prefijados para articulación (secuencias fijas) a modo de fotogramas en cada instante. Esto es posible al alto grado de desacoplamiento existente entre los componentes de JdeRobot y los robots simulados

y reales a los que da soporte la plataforma. La exportación del modelo de generación de caminatas realizado por Francisco Rivas a las nuevas versiones de Gazebo y JdeRobot no ha supuesto una gran complicación, ya que dicho modelo es independiente de la plataforma, por lo que bastó con implementar dicho modelo en el *plugin* específicamente diseñado para las caminatas sin demasiada complicación. Lo realmente complicado de esta tarea fue el conseguir unos valores coherentes para realizar un movimiento natural y coordinado de todas las articulaciones que permitieran caminar al robot, desarrollando, para este fin, un algoritmo de búsqueda rastrillada.

En cuanto al algoritmo de búsqueda, se ha implementado desde cero definiendo unos rangos de valores y unos pasos para cada parámetro presente en la caminata. Esta decisión se tomó debido a que el espacio de búsqueda sin acotar es infinito, por lo que para poder reducir el tiempo de búsqueda de una caminata óptima cada parámetro está acotado según un criterio determinado. Con la configuración actual se generan más de un millón de candidatos a caminata, lo cual, teniendo en cuenta que a cada simulación se dedican de media unos 7 segundos hace que el tiempo que debería emplearse en evaluar todos los candidatos sea, al menos aceptable. Además de todo lo expuesto, se ha dado un carácter aleatorio a la selección de candidatos generados, permitiendo que el espacio de búsqueda se explore de forma más diversa y no tan lineal. Con esto se consigue que, si por ejemplo la caminata óptima es la número 15.000 las posibilidades de encontrarla antes de haber evaluado 14.999 candidatos aumente, eliminando ese factor secuencial a la hora de recorrer una estructura de datos.

Como valor añadido a este componente, se le ha dotado de la capacidad de parada y reanudación, de tal modo que permita parar la simulación en cualquier momento y retomarla cuando se desee. Debido a la gran carga de CPU que supone el evaluar las caminatas, incluímos esta funcionalidad para darle la posibilidad al usuario de utilizar su PC si lo requiriera, en cualquier instante. El desarrollo de la búsqueda con todas las funcionalidades añadidas que se han citado no ha sido especialmente un reto, ya que el número de parámetros a los que se redujo el movimiento de caminar fue considerable, dejando así un espacio de búsqueda muchísimo más reducido y teniendo que considerar menos parámetros a la hora de codificar el componente.

Sobre el interfaz gráfico, la idea principal es que fuera amigable para el usuario, y liviano además de ofrecer toda la información relevante en cada momento de la simulación. Por ello se han realizado bastantes cambios desde el prototipo inicial hasta lo que es hoy día habiéndose estudiado de forma concreta cada componente del interfaz. Además, también se ha desarrollado pensando en el futuro, ya que ofrece elementos que, a pesar de no

estar implementados en la version actual, puede completar la herramienta con muchas funcionalidades extra.

Por último, para el desarrollo de todos los componentes software presentes en este proyecto se ha tenido en cuenta un factor importante en la plataforma, y es la posibilidad de exportar este modelo a otros robots bípedos que se pudieran incorporar en un futuro, realizando el menor número de cambios posible, por lo tanto la inclusión de nuevos robots será posible siempre que este nuevo modelo utilice las mismas interfaces de comunicación que se utilizan actualmente.

Como novedades aportadas en este proyecto, ahora disponemos de un modelo de humaoide corregido y mejorado con el que se pueden desarrollar multitud de desarrollos relacionados con la locomoción robótica en robots bípedos. Dicho soporte, como se ha apuntado anteriormente, no era del todo preciso al principio de este proyecto, por lo que las simulaciones en temas de locomoción en las que se veían involucradas varias articulaciones al mismo tiempo no eran del todo realistas. Además, se ha dotado a este robot humanoide de un comportamiento específico, en concreto la posibilidad de caminar de forma autónoma a partir de unos pocos valores especificados previamente.

Para finalizar esta sección, dejaremos constancia de lo aprendido durante el desarrollo de este proyecto. Este tipo de proyectos implican el conocimiento en profundidad de diversas tecnologías muy diferentes entre sí. Este proyecto incluye algunos desarrollos previos de familiarización del entorno JdeRobot que han abarcado las siguientes tecnologías: lenguaje de programación C++, librerías gráficas de Qt, algunos elementos de Python, el middleware de comunicaciones ICE, la librería de OpenGL, entre otras. A parte de los conocimientos inherentes que se derivan del desarrollo de un proyecto como este, además se han estudiado temas tales como la parametrización de movimientos en robots, acoplamiento de ondas, programación de controladores orientados a la simulación entre muchos otros.

6.2. Trabajos futuros

Puesto que no podemos abarcar todas las posibilidades que ofrece un robot humanoide en cuanto a desarrollos y aplicaciones posibles, este proyecto se ha centrado en mejorar lo ya existente para disponer de un soporte más robusto, flexible y actualizado del robot Nao en simulación. Con los objetivos conseguidos, se abren infinidad de posibilidades de desarrollo y líneas de investigación, con lo que utilizaremos esta breve sección para comentarlos.

En cuanto al robot simulado en Gazebo, la refactorización llevada a cabo ha dejado

algunos elementos interesantes por explorar. Por ejemplo, la inclusión de algunos de los sensores más completos del Nao, como el sensor inercial, el de ultrasonidos o el de contacto de los pies; que han sido obviados en este proyecto. Se pueden realizar desarrollos muy interesantes con estos sensores, por ejemplo, se puede generar un comportamiento que dote de equilibrio al Nao en situaciones en las que se apliquen grandes fuerzas al modelo (empujándolo) para que caiga y recupere la posición en la que estaba. Además, se pueden añadir elementos en simulación como los dedos, los cuales hemos obviado al no ser relevantes en este proyecto, y sus *plugins* asociados. La inclusión de una mano simulada puede dar lugar a comportamientos como los de manipular objetos del entorno, e incluso transportar pequeños objetos de un lado a otro.

El comportamiento que se ha definido en este proyecto ha sido el de caminar. No obstante, la infraestructura resultante de este desarrollo deja las puertas abiertas a la generación de otro tipo de comportamientos complejos y compuestos que se pueden utilizar para diversas tareas. Un ejemplo sería generar secuencias de movimientos de yoga, utilizando al Nao como monitor. Dado que dispone de micrófono, podrá dar las instrucciones mientras va realizando los movimientos. Otra posibilidad, utilizando la funcionalidad desarrollada de las caminatas, sería el utilizar al Nao como robot guía.

En cuanto al modo de caminar desarrollado, éste solo realiza trayectorias en línea recta, de modo que esta funcionalidad se puede ampliar para que el robot realice movimientos laterales y cambios de trayectoria sobre la marcha, permitiendo realizar recorridos heterogéneos e incluso esquivar obstáculos utilizando los diferentes sensores de los que dispone.

Por último, y en la línea de optimización del proyecto, la búsqueda de parámetros para generar candidatos a caminata se ha hecho por "fuerza bruta". Se ha implementado un algoritmo de búsqueda rastrillada que recorre un espacio de búsqueda finito entre multitud de posibles valores. Este algoritmo se puede mejorar implementando algoritmos de búsqueda más avanzados como los genéticos o los evolutivos, optimizando así esta labor.

Bibliografía

[Agüero *et al.*, 2010] Carlos E. Agüero, José M. Cañas, Francisco Martín, and Eduardo Perdices. Behavior-based iterative component architecture for soccer applications with the nao humanoid. In *5th Workshop on Humanoids Soccer Robots. Nashville, TN, USA*, 2010.

[Bermejo and Cañas, 2012] J. Bermejo and J. Cañas. Soporte del robot humanoide nao en el simulador 3d gazebo. In *Proceedings of XIII Workshop on Physical Agents, WAF 2012*, pages 73–80, Santiago de Compostela, September 2012.

[Bermejo, 2010] Jorge Bermejo. Soporte del robot humanoide nao en simulador gazebo para aplicaciones en jderobot. Trabajo fin de carrera, ingeniería técnica en informática de sistemas, Universidad Rey Juan Carlos, 2010.

[Brooks *et al.*, 2007] Alex Brooks, Tobias Kaupp, Alexei Makarenko, Stefan Williams, and Anders Orebäck. Orca: a component model and repository. In *Software engineering for experimental robotics*, pages 231–251. Springer, 2007.

[Brugali, 2007] Davide Brugali. *Software engineering for experimental robotics*, volume 30. Springer, 2007.

[Cañas Plaza, 2003] Jose María Cañas Plaza. Jerarquía dinámica de esquemas para la generación de comportamiento autónomo. *Tesis Doctoral - Universidad Politécnica de Madrid*, Diciembre 2003.

[Echeverria *et al.*, 2011] G. Echeverria, N. Lassabe, A. Degroote, and S. Lemaignan. Modular openrobots simulation engine: Morse. In *Proceedings of the IEEE ICRA*, 2011.

[Gerkey *et al.*, 2003] Brian P. Gerkey, Richard T. Vaughan, and Andrew Howard. The player/stage project: tools for multi-robot and distributed sensor systems. In *Proceedings of the 11th International Conference on Advanced Robotics ICAR-2003, pp. 317-323*, Coimbra (Portugal), 2003.

[Jackson, 2007] J. Jackson. Microsoft robotics studio: a technical introduction. *IEEE Robotics & Automation Magazine*, 14(4):82–87, 2007.

[Koening and Howard, 2004] N. Koening and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *Proceedings of 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Sendai, Japan, 2004.

[Makarenko *et al.*, 2006] Alexei Makarenko, Alex Brooks, and Tobias Kaupp. Orca: Components for robotics. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 163–168, 2006.

[Martín *et al.*, 2013] F. Martín, C.E. Agüero, J.M: Cañas, M. Valenti, and Pablo Martínez-Martín. Robotherapy with dementia patients. *Int. J. of Advanced Robotic Systems*, 10(10), 2013.

[Martín, 2010] Carlos Iván Martín. Herramienta de programación visual de autómatas en jderobot. Trabajo fin de carrera, ingeniería técnica en informática de sistemas, Universidad Rey Juan Carlos, 2010.

[Menéndez *et al.*, 2013] Borja Menéndez, José M. Cañas, and Rubén Salamanqués. Programming of a nao humanoid in gazebo using hierachical fsm. In *Proceedings of the XIV Workshop en Agentes Físicos (WAF-2013)*, pages 15–22, Madrid, 2013.

[Menéndez, 2013] Borja Menéndez. Visualhsfm 4: Programación de humanoides con autómatas de estado finito usando jderobot. Trabajo fin de máster, sistemas telemáticos e informáticos, Universidad Rey Juan Carlos, 2013.

[Quigley *et al.*, 2009] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, 2009.

[Rivas *et al.*, 2011] Francisco Rivas, José M. Cañas, and Juan González. Aprendizaje automático de modos de caminar para un robot humanoide. In *Proceedings of Robot2011 III Workshop de Robótica: Robótica experimental*, pages 120–127, Sevilla, 2011.

[Rivas *et al.*, 2013] Francisco Rivas, José M. Cañas, Borja Menéndez, Julio Vega, Eduardo Perdices, Rubén Salamanqés, and Francisco Martin. Programming a humanoid social robot using the jderobot framework. In *Proceedings of RoboCity2030 11th Workshop, Robots sociales*, pages 71–94, U.Carlos III, Madrid, 2013.

[Rivas, 2011] Francisco Miguel Rivas. Caminata basada en ondas acopladas para el robot humanoide nao. Trabajo fin de carrera, ingeniería técnica en informática de sistemas, Universidad Rey Juan Carlos, 2011.

[Salamanqués, 2012] Rubén Salamanqués. Herramienta de programación visual de autómatas de estado finito jerárquicos para aplicaciones robóticas. Trabajo fin de carrera, ingeniería técnica en informática de sistemas, Universidad Rey Juan Carlos, 2012.

[Vaughan and Gerkey, 2007] Richard T. Vaughan and Brian P. Gerkey. Reusable robot software and the player/stage project. In D. Brugali, editor, *Software Engineering for Experimental Robotics*, pages 267–289. Springer, Berlin / Heidelberg, 2007.

[Yunta and Cañas, 2012] D. Yunta and J. Cañas. Programación visual de autómatas para comportamientos en robots. In *Proceedings of XIII Workshop on Physical Agents, WAF 2012*, pages 65–71, Santiago de Compostela, September 2012.

[Yunta, 2011] David Yunta. Herramienta de programación visual de autómatas de estado finito para aplicaciones robóticas. Trabajo fin de carrera, ingeniería técnica en informática de sistemas, Universidad Rey Juan Carlos, 2011.